Análisis e Implementación de Técnicas para el Descubrimiento de Reglas de Asociación Temporales Generalizadas

Ana Belén Blazquez – María Adriana Colareda

Mayo de 2004

Índice General

Abstract

Con la búsqueda de reglas de asociación se intenta hallar posibles asociaciones entre ítems. Existen ciertas restricciones, como el soporte mínimo y la confidencia para obtener asociaciones interesantes. Un supuesto implícito en el modelo tradicional de reglas de asociación es que los ítems se distribuyen de manera uniforme en el tiempo. Evidentemente, este supuesto diverge de la realidad, ya que permanentemente están apareciendo ítems nuevos y otros dejan de comprarse o producirse. Por otra parte, existen ítems que aparecen en transacciones sólo en determinados momentos o períodos del año. De la idea del refinamiento del modelo tradicional, con la incorporación de la variable *tiempo*, han surgido diversos modelos temporales de reglas de asociación. En este trabajo implementamos el algoritmo TDIC, diseñado dentro del Modelo de Reglas de Asociación Temporales Basado en LifeSpans, analizamos su performance y escalabilidad comparándolo con DIC y Apriori. Para esto implementamos un generador de transacciones sintéticas de amplio uso y lo extendemos con tiempos obtenidos por un simulador de arribos y despacho de colas.

Agradecimientos

(en orden alfabético)

Virginia Blazquez

Soledad Escobar

Abraham Ovando

Natalia Romero

Esther Sarasua

Silvia Gordillo, nuestra co-directora y Patricia Bazán, coordinadora de Trabajos de Grado, por darnos una mano con los trámites administrativos.

Y especialmente a *Juan María Ale*, nuestro director, por su invalorable apoyo y dedicación.

Capítulo 1

Introducción

En una de sus cortas historias, La biblioteca de Babel, Jorge Luis Borges describe una biblioteca infinita. Una red infinita de habitaciones con anaqueles que tienen innumerable cantidad de libros, la mayoría de ellos sin significado y con títulos tan inentendibles como "Axaxaxas mlö". La gente vaga por esta biblioteca hasta que muere y los sabios desarrollan hipótesis sobre ella, tales como: "en algún lugar debe existir un catálogo"; "todos los libros que uno necesita deben estar en algún lugar de la biblioteca"; "la biblioteca tiene una estructura infinita que se repite indefinidamente". Ninguna de estas hipótesis puede verificarse - la biblioteca contiene una cantidad infinita de datos, pero no contiene información al alcance de la mano.

La biblioteca de Babel puede interpretarse como una metáfora interesante, pero cruel, de la situación en la cual se encuentra la humanidad actualmente: vive en un universo de datos que se expande constante y vertiginosamente; generando demasiados datos y demasiada poca información.

Varios estudios han revelado que la cantidad de datos en el mundo se duplica anualmente, y como consecuencia sorprendente, la cantidad de información decrece rápidamente. Así es como la mayoría de las organizaciones produce más información en una semana de lo que la mayoría de la gente puede consumir en una vida, poseen grandes bases de datos que contienen mucha riqueza en información potencialmente accesible. Sin embargo, generalmente es muy difícil acceder a esta información.

Por lo mencionado, el desarrollo de nuevas técnicas para encontrar información en grandes cantidades de datos es uno de los principales desafíos de los actuales desarrolladores de software. Es en este contexto donde aparece uno de los aspectos más fascinantes de la ciencia de la computación: la utilización de la computadora para descubrir información nueva y significativa. La producción mecánica de datos ha creado la necesidad de consumo mecánico de los datos. Esto lleva a adoptar estrategias para desarrollar métodos mecánicos para filtrar, seleccionar e interpretar los datos para así encontrar información escondida y sumamente útil.

El análisis de los datos puede proveer mayor conocimiento sobre un negocio dado. Es decir, al introducirse en los datos se puede derivar conocimiento sobre el negocio. Aquí es donde Data Mining y Knowledge Discovery in Databases (KDD) tienen beneficios para cualquier empresa.

Se puede ver a Data Mining como el proceso de analizar filas de información en bases de datos, como las mencionadas anteriormente, y sintetizarlas en información que sea útil para realizar decisiones efectivas. Apunta a utilizar información existente para derivar nuevos hechos y descubrir nuevas relaciones previamente desconocidas, aún para expertos que estén familiarizados con los datos.

Data Mining, también, es reconocido como un área nueva en investigaciones de base de datos. El área puede definirse como descubrir eficientemente reglas interesantes en largas colecciones de datos.

La necesidad de procesar grandes volúmenes de datos diferencia Data Mining en bases de datos de su estudio en el contexto de la Inteligencia Artificial. La clasificación es una aproximación a tratar de desarrollar tuplas de datos juntas basadas en ciertas características comunes. Esto ha sido estudiado en el contexto de las bases de datos y la Inteligencia Artificial. Otra fuente de Data Mining son los datos ordenados, tales como stock de supermercado y puntos de venta. Aspectos interesantes incluyen búsquedas de secuencias similares, por ejemplo: stocks con movimientos similares en precios, y patrones secuenciales, por ejemplo: ítems de verdulería comprados en un conjunto de visitas en secuencia.

En varios tipos de aplicaciones de Data Mining, se considera importante el análisis de los datos transaccionales. Se asume que las bases de datos mantienen información sobre transacciones de usuarios, donde cada transacción es una colección de ítems de datos. La noción de *regla de asociación* fue propuesta para capturar la co-ocurrencia de ítems en las transacciones. Por ejemplo, dada una base de datos de órdenes de un restaurante (transacciones), se puede obtener una regla de asociación de la forma:

Medialunas ➔ café (Soporte: 3%, confidencia: 80%)

Lo que significa que el 3% de todas las transacciones contienen los ítems *medialuna* y *café*, y el 80% de las transacciones que tiene el ítem *medialuna* también tienen el ítem *café*. Los dos parámetros de porcentaje se conocen normalmente como *soporte* y *confidencia*, respectivamente.

Una extensión interesante a las reglas de asociación, como las mencionadas, es incluir dimensiones temporales. Por ejemplo, *medialunas* y *café* pueden pedirse juntos principalmente entre las 7:00 Hs. y las 11:00 Hs. Así, se puede ver que la *regla de asociación* tiene un soporte de 40% en las transacciones que suceden entre las 7:00 y las 11:00 y un soporte de 0,005% en el resto de las transacciones.

Esto sugiere que se pueden descubrir diferentes reglas de asociación si se consideran diferentes intervalos de tiempo. Algunas *reglas de asociación* pueden valer durante ciertos

intervalos de tiempo, pero no durante otros. Descubrir los intervalos temporales y las *reglas de asociación* que valen en esos intervalos puede llevar a descubrir información útil. Informalmente, las reglas de asociación durante sus intervalos temporales serán denominadas *reglas de asociación temporal*.

1.1 Objetivo de este Trabajo

El objetivo de este trabajo es implementar y verificar algoritmos que contemplen la temporalidad de los datos, partiendo de la especificación Apriori según [3]. Para la implementación hemos realizado una extensión al sistema WEKA [12] desarrollado en la Universidad de Waikato, en Nueva Zelanda, que nos permite aprovechar las funciones comunes y abocarnos fundamentalmente a los aspectos particulares del algoritmo temporal de [1] y la definición e implementación de Dynamic ItemSet Counting [7].

La base de información a utilizar será una base de datos sintética que intenta representar el comportamiento de las transacciones comerciales de datos llamados *basket* en el mundo real.

Se mostrará el resultado de las comparaciones de los distintos mecanismos de búsqueda de *reglas de asociación*, logrando así demostrar que se puede llegar a obtener información significativa a través de distintos caminos. "Sólo hay que buscar el camino correcto para lo que se desea encontrar o simplemente buscar".

1.2 Organización

Capítulos

El presente trabajo está organizado a lo largo de los capítulos que siguen de esta manera:

Capítulo 2, *Reglas de Asociación* - Presenta un resumen de la disciplina Data Mining e introduce al tema de las reglas de asociación.

Capítulo 3, *Reglas de Asociación Temporales* - Introduce en el tema de la temporalidad en la búsqueda de información.

Capítulo 4, *Experimentación* - Define el proceso de generación de datos sintéticos, el escenario de ejecución de pruebas y presenta la información obtenida.

Capítulo 5, *Análisis y conclusiones de los resultados obtenidos*.

Capítulo 6, *Conclusión* – Resume las conclusiones de este trabajo.

Apéndices

Se presentan los siguientes apéndices, que exhiben los detalles de los algoritmos estudiados y desarrollados.

Apéndice A, *WEKA, DIC y TDIC* - Presenta en detalle el desarrollo base de la investigación, y las modificaciones incorporadas.

Apéndice B, *Generador de datos sintéticos* - Presenta en detalle el desarrollo para generar el Input a los algoritmos a testear.

Apéndice C, *Generador de cola de llegadas o clientes* - Presenta en detalle el desarrollo para generar el Input complementario a los algoritmos a testear.

Capítulo 2

Reglas de Asociación

2.1 Introducción a Data Mining

Data Mining, or Knowledge Discovery in Databases (KDD) as it is also known, is the nontrivial extraction of implicit, previously unknown, and potentially useful information from data. This encompasses a number of different technical approaches, such as clustering, data summarization, learning classification rules, finding dependency net works, analysing changes, and detecting anomalies.

William J. Frawley, Gregory Piatetsky-Shapiro y Christopher J Matheus.

Data mining is the search for relationships and global patterns that exist in large databases but are `hidden' among the vast amount of data, such as a relationship between patient data and their medical diagnosis. These relationships represent valuable knowledge about the database and the objects in the database and, if the database is a faithful mirror, of the real world registered by the database.

Marcel Holshemier & Arno Siebes (1994)

Data mining refers to "using a variety of techniques to identify nuggets of information or decision-making knowledge in bodies of data, and extracting these in such a way that they can be put to use in the areas such as decision support, prediction, forecasting and estimation. The data is often voluminous, but as it stands of low value as no direct use can be made of it; it is the hidden information in the data that is useful"

Clementine User Guide, a data mining toolkit.

2.1.1 Sistemas que Pueden Aprender

La habilidad de aprender es inherente a los seres humanos; aunque organismos relativamente simples, como las plantas o las amebas tienen esta capacidad. Por un lado, las plantas "aprenden" a maximizar la cantidad de luz que reciben al rotar sus hojas al sol. Por otro lado, los humanos aprendieron a explotar una estructura extremadamente complicada y delicada como es el lenguaje, encontrando un mecanismo para explorar las leyes del universo. Esta capacidad de aprender parece ser una característica esencial de la vida. La teoría de la evolución muestra que las especies que sobreviven son aquellas que mejor se adaptan a sus ambientes. Dado que el aprendizaje es una forma de adaptación, podemos concluir que es un aspecto central en la aparición de la vida en nuestro planeta.

Por esto, existe una motivación filosófica en la búsqueda de sistemas de computación que puedan aprender. Pocos especialistas, en la actualidad, dudan que un programa de computación inteligente sea también un programa con capacidad de aprender. No puede haber Inteligencia Artificial (IA) sin "Aprendizaje Artificial" (o machine learning, ML). Por ende los programas que pueden aprender han sido el foco de la Inteligencia Artificial desde el principio de la tecnología de computación. Con el entusiasmo inicial, allá por los años 50, los primeros científicos exageraron los resultados de los casos, y cuando inevitablemente fallaron, los fondos de investigación fueron recortados al final de los 60. Como resultado, machine learning fue condenado a llevar una vida oculta en las universidades y centros de investigación durante dos décadas.

Los comienzos de los 80 trajeron cambios. Una renovada generación de investigadores hicieron nuevos descubrimientos: algoritmos simples para crear árboles de decisión para la clasificación de clases arbitrarias de objetos; nuevas arquitecturas para redes neuronales y otros planteos como algoritmos genéticos modelados sobre la teoría de la evolución. El avance de la tecnología les ofreció a los investigadores máquinas más poderosas, permitiéndoles testear sus algoritmos con datos reales de forma más rápida y eficiente. A esta altura, era claro que muchas tareas que a simple vista parecían fáciles de resolver con una computadora, eran de hecho lo opuesto, dada su complejidad. Tareas simples en el área de planeamiento son extremadamente difíciles de resolver con una computadora; aunque en estos casos, una persona puede resolver el problema relativamente más fácil. La persona que realiza una planificación parece poder aprender cómo manejar la complejidad desde la experiencia. La capacidad de aprendizaje jugó un rol importante en la Inteligencia Artificial. Como resultado, la atención giró a la construcción de algoritmos que aprendieran y al área de los sistemas expertos.

2.1.2 ¿Qué es Data Mining?

Existe otra motivación para el estudio de machine learning, la explosión de datos en la sociedad moderna y la necesidad de interpretar grandes volúmenes de información. Una de las principales razones del éxito limitado de los sistemas de bases de datos actuales es que no proveen la funcionalidad necesaria para obtener información a través de sus contenidos.

Fundamentalmente, Data Mining se refiere al análisis de datos y al uso de técnicas de software para encontrar patrones y regularidades en conjuntos de datos. La computadora es la responsable de encontrar los patrones al identificar las reglas subyacentes y características de los datos. Es posible descubrir oro en lugares inesperados ya que el

software de Data Mining extrae patrones no perceptibles previamente o tan obvios que nadie antes los advirtió.

Interceptando Data Mining con las disciplinas de machine learning, estadísticas y bases de datos se logran encontrar relaciones y patrones globales que existen en grandes volúmenes de datos pero que están ocultos a primera vista.

El análisis de Data Mining tiende a trabajar desde los datos y las mejores técnicas son aquellas desarrolladas con orientación a grandes volúmenes de información, utilizando la mayor cantidad de los datos recolectados como sea posible para llegar a conclusiones y decisiones confiables. El proceso de análisis comienza con un conjunto de datos, utiliza una metodología para desarrollar una representación óptima de la estructura de datos durante el tiempo de adquisición de conocimiento. Una vez que se obtuvo el conocimiento, esto puede extenderse a conjuntos de datos más grandes, trabajando sobre la base que los datos más grandes tienen una estructura similar al conjunto original. Nuevamente, esto es análogo a la operación de mining, donde grandes cantidades de datos de bajo impacto se juntan para encontrar algo de valor.

Respecto del significado exacto de los términos "Data Mining" y "KDD (Knowledge Discovery in Databases)", varios autores los consideran sinónimos, aunque en la primer conferencia internacional de KDD en 1995 se propuso que el término KDD sea empleado para describir el proceso completo de extracción de conocimiento desde los datos. En este contexto, conocimiento significa relaciones y patrones entre los datos. Se propuso, también, que el término "Data Mining" sea más específico y se utilice exclusivamente para la etapa del descubrimiento del proceso de KDD. Una definición menos oficial de KDD es "la extracción no trivial de conocimiento previamente desconocido y potencialmente útil desde los datos". Entonces, el conocimiento debe ser nuevo, no obvio y debe poder utilizarse. KDD no es una técnica nueva, sino un campo multi-disciplinario de la investigación: machine learning, estadísticas, tecnología de bases de datos, sistemas expertos y visualización de datos. Todos hacen su contribución.

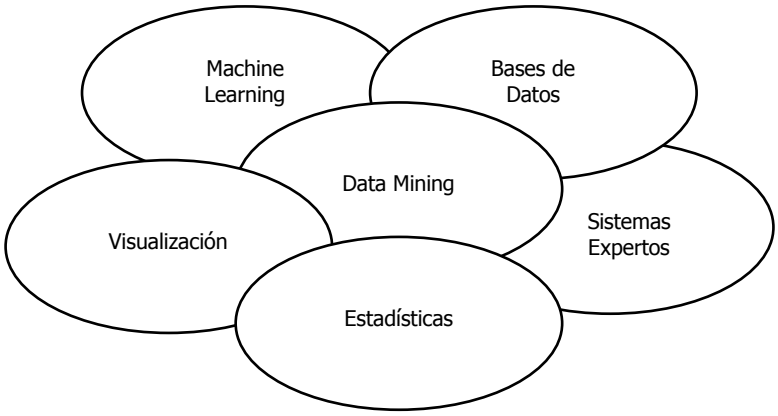


Figura 2.1: KDD es un área de investigación multi-disciplinario

En el proceso de KDD se pueden distinguir seis etapas:

- Selección de datos
- Preprocesamiento: limpieza y enriquecimiento
- Transformación o codificación
- Data Mining, fase de descubrimiento
- Interpretación, Evaluación y reporte de resultados

Aunque la metodología da la impresión de una trayectoria lineal, no es así. En cada etapa, se puede volver atrás una o más fases. Por ejemplo, en la etapa de transformación o de Data Mining, puede descubrirse que la limpieza está incompleta y regresar a limpiar nuevamente.

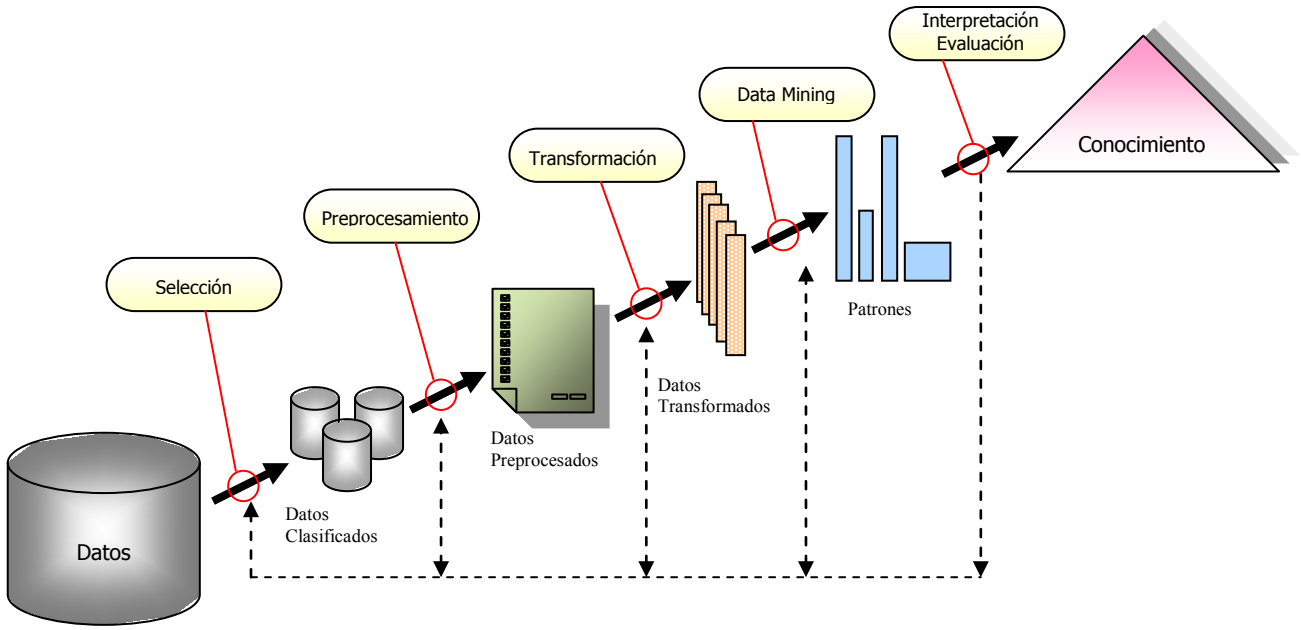


Figura 2.2: Proceso de KDD

2.1.3 Ejemplos de Vida Real

En un supermercado existe una gran colección de ítems. Las decisiones típicas de negocio que los gerentes del supermercado tienen que hacer incluyen qué poner a la venta, cómo diseñar los cupos, cómo poner la mercadería en las góndolas para maximizar las ganancias, etc. El análisis de las transacciones ocurridas en el pasado se utiliza comúnmente para mejorar la calidad de estas decisiones. Hasta hace poco, solamente estaba disponible información sobre ventas en algunos períodos de tiempo (un día, una semana, un mes, etc.). Pero afortunadamente los progresos en la tecnología, como por ejemplo: los códigos de barras, han posibilitado almacenar en una base de datos por transacción los llamados datos *basket*. Las transacciones de tipo *basket* no necesariamente consisten en ítems comprados juntos en el mismo momento. Pueden consistir en ítems comprados por un cliente en un período de tiempo. Ejemplos de estas transacciones pueden incluir compras mensuales realizadas por miembros de un club de libros o un club de música. Las compañías de venta por catálogo también suelen coleccionar datos de venta de las órdenes que reciben. Un registro de tales datos típicamente consiste en la fecha de transacción, el ítem o ítems comprados, y posiblemente el número de cliente si la transacción se hace por tarjeta de crédito o alguna tarjeta de cliente.

El análisis de los datos de transacciones pasadas puede proveer información importante sobre el comportamiento de los clientes y así mejorar la calidad de las decisiones de negocio, tales como: qué poner a la venta, qué mercaderías poner en la misma góndola, cómo diseñar los programas de marketing, para nombrar algunos. Es esencial coleccionar una cantidad suficiente de datos de venta (digamos un mes) antes de poder realizar alguna conclusión. Como resultado, la cantidad de información tiende a ser muy grande y aquí es donde comienza la desinformación en la gran información. Por eso, es importante desarrollar algoritmos eficientes para realizar el Data Mining sobre estos datos.

El interés actual en Data Mining puede explicarse por los siguientes factores:

1.- En los 80 las grandes organizaciones crearon bases de datos conteniendo información de clientes, competidores y productos. Estas bases de datos contienen Gbytes de datos con "mucha" información que no puede obtenerse fácilmente utilizando SQL. Los algoritmos de Data Mining pueden encontrar "agrupaciones", o regularidades interesantes en una base de datos. SQL es un lenguaje de querys que sólo ayuda a encontrar datos bajo ciertas restricciones que se conocen. Los algoritmos de Data Mining típicamente sub-dividen la base de datos.

2.- Como el uso de las redes continúa creciendo, se torna muy fácil conectar bases de datos.

3.- En los últimos años, las técnicas de machine learning se han expandido enormemente. Redes neuronales, algoritmos genéticos y otras técnicas de aprendizaje hacen fácil encontrar conexiones interesantes en las bases de datos.

4.- La revolución cliente/servidor permite acceder a sistemas de información central. Los especialistas de marketing también quieren disponer de estas técnicas.

2.2 Introducción a Reglas

2.2.1 Reglas de Asociación

Los gerentes de marketing buscan reglas tales como: el 90% de las mujeres con autos deportivos rojos y perros pequeños, usan Channel Nº 5. Este tipo de descripción les da un perfil claro de los clientes a los cuales dirigir sus acciones de marketing. La pregunta es si este tipo de reglas pueden encontrarse con herramientas de Data Mining. La respuesta es "si" y en Data Mining este tipo de relación se la conoce como regla de asociación.

Supongamos que tenemos una base de datos que contiene información del género del cliente, el color y tipo de su auto, el tipo de mascota que tiene, y los productos que compra. Entonces, la regla mencionada anteriormente, se reflejaría en la base de la forma: el 90% de los registros donde género es femenino, auto es deportivo, el color del auto es rojo, y la mascota es perro pequeño, entonces el perfume es Channel Nº 5.

El problema con las reglas de asociación es que se encuentran tantas asociaciones que es muy difícil separar información valiosa del "ruido", y por eso es necesario introducir medidas para distinguir las asociaciones interesantes de las no-interesantes.

Las reglas de asociación siempre se definen con atributos binarios y se representan de la forma:

REV_MUSICA, REV_HOGAR ➔ REV_AUTOS

Esto significa que alguien que lee revistas de música y de hogar, es muy probable que lea revistas de autos.

¿Qué reglas son interesantes?. En principio, se deben buscar reglas que estén presentes un gran número de veces en la base de datos, y esto define el *soporte de una regla de asociación*. En el ejemplo anterior el soporte de la regla es el porcentaje de registros para los cuales valen REV_MUSICA, REV_HOGAR y REV_AUTOS. Es decir, todas las personas que leen las tres revistas.

Sin embargo, el soporte en sí mismo no es suficiente. Puede suceder que un grupo considerable de personas lean las tres revistas y un grupo mayor lea REV_MUSICA y

REV_HOGAR pero no lean REV_AUTOS. En este caso la asociación es débil, aunque el soporte sea relativamente alto. Se necesita una medida adicional que se denominará ***confidencia***, que en el ejemplo es el porcentaje de registros para los cuales vale REV_AUTOS, dentro del grupo de registros para los cuales vale REV_MUSICA y REV_HOGAR.

Se puede concluir que las ***Reglas de Asociación*** son un algoritmo bien definido, cuyo significado se mide por factores de soporte y confidencia. Se componen de una condición y un resultado. Generalmente se representan de la forma: "If *condición* then *resultado*". Es una implicación de la forma X ➔ Y que significa que en las tuplas de datos para las cuales valen los atributos de X, también valen los atributos de Y.

En las bases de datos de un supermercado las reglas tratan de identificar relaciones del tipo "Un cliente que compra el ítem X a veces también compra el ítem Y".

Aplicar Reglas de Asociación a una base de datos de transacciones de venta significa descubrir todas las asociaciones entre ítems tal que la presencia de algunos ítems en una transacción implicará la presencia de otros ítems en la misma transacción.

También pueden considerarse como un paso para mejorar las bases de datos con funcionalidades para procesar búsquedas tales como (se omite la especificación del factor de confidencia):

- Encontrar todas las reglas que tienen "Coca Light" en el consecuente. Estas reglas pueden ayudar a planear lo que debería hacer el mercado para mejorar la venta de Coca Light.
- Encontrar todas las reglas que tengan "Cerealitas" en el antecedente. Estas reglas pueden ayudar a determinar qué productos podrían verse impactados si el mercado decide discontinuar la venta de Cerealitas.
- Encontrar todas las reglas que tengan "salchichas" en el antecedente y "mostaza" en el consecuente. Esta búsqueda puede utilizarse alternativamente como pedido para conocer los ítems adicionales que tengan que venderse junto con salchichas para aumentar la probabilidad que también se venda la mostaza.
- Encontrar todas las reglas que relacionen los ítems de las góndolas A y B. Estas reglas pueden ayudar a planear la organización de las góndolas al determinar si la venta de ítems de la góndola A está relacionada con la venta de ítems de la góndola B.
- Encontrar las "mejores k reglas" que tengan Cerealitas en el consecuente. Donde "mejores k reglas" puede formularse en términos de los factores de confidencia de

las reglas, o en términos de su soporte; es decir, las fracciones de transacciones que satisfacen la regla.

2.2.2 Modelo Formal

Sea $I = i_1, i_2, ..., i_m$ un conjunto de literales llamados ítems. Sea D un conjunto de transacciones, donde cada transacción T es un conjunto de ítems tal que $T \subseteq I$. Asociado a cada transacción existe un identificador único llamado *TID*. Se dice que una transacción T contiene a X, un conjunto de algunos ítems de I, si $X \subseteq T$.

Una *regla de asociación* es una implicación de la forma $X \Rightarrow Y$, donde $X \subset I$, $Y \subset I$, $X \cap Y = \varnothing$. La regla $X \Rightarrow Y$ se satisface en el conjunto de transacciones D con factor de *confidencia* $0 \leq c \leq 1$, si y sólo si al menos el c% de las transacciones en D que contienen X también contienen Y. La regla $X \Rightarrow Y$ tiene *soporte* s en el conjunto de transacciones D si el s% de las transacciones en D contienen $X \cup Y$.

Dado un conjunto de transacciones D, el problema de encontrar reglas de asociación es generar todas las reglas que tengan soporte y confidencia mayor que un soporte mínimo (llamado *minsup*) y confidencia mínima (llamada *minconf*) respectivamente, especificados por el usuario.

La representación de la información no debería afectar al mecanismo de búsqueda. D podría ser un archivo de datos, una tabla relacional, o el resultado de una expresión relacional.

Las reglas a generar desde el conjunto de transacciones mencionado deberán satisfacer ciertas restricciones adicionales:

1. Restricciones sintácticas: estas restricciones se aplican sobre los ítems que pueden aparecer en la regla. Por ejemplo, pueden ser interesantes sólo reglas que tengan un ítem específico X en el consecuente, o reglas que tengan un ítem específico Y en el antecedente. También son posibles combinaciones de las restricciones anteriores, pueden pedirse reglas que tengan ítems de algún conjunto predefinido X en el consecuente, e ítems de algún otro itemset Y en el consecuente.
2. Restricciones de soporte: Estas restricciones involucran el número de transacciones en D que soportan una regla. El soporte de una regla se define como la fracción de transacciones en D que satisfacen la unión de ítems en el consecuente y antecedente de la regla.

El soporte no debe confundirse con la confidencia de una regla. La confidencia es una medida de fuerza de la regla, mientras que el soporte se corresponde con un significado

estadístico. Con esto, el problema de descubrir reglas de asociación puede descomponerse en dos subproblemas:

1. Encontrar todos los conjuntos de ítems (denominados *itemset*) que tengan soporte de transacción sobre el soporte mínimo, llamado *minsup*. El soporte para un itemset es el número de transacciones que contiene el itemset. Los itemsets con soporte mínimo son llamados ***Itemsets Frecuentes*** y todos los otros ***Itemsets no Frecuentes***.
2. Utilizar los Itemsets Frecuentes para generar las reglas deseadas. La idea general es que si X e Y son Itemset Frecuentes, entonces se puede determinar si la regla $X \Rightarrow Y$ tiene factor de confidencia c al calcular *conf* = soporte(X) / soporte(Y). Si *conf* es mayor que c entonces la regla se satisface con factor de confidencia c; de otra manera, la regla no satisface la confidencia.

Si se tiene que el itemset X es frecuente, entonces todo subconjunto de X también será frecuente, y debemos tener disponible su soporte, como resultado de la solución del primer subproblema. Además, todas las reglas derivadas de X deben satisfacer la restricción de soporte, dado que X satisface el soporte mínimo y X es la unión de ítems en el consecuente y antecedente de tales reglas.

Para lograr la mayor eficacia con respecto a performance en la búsqueda de reglas de asociación se debe hacer foco en la performance del primer paso.

Se puede pensar que la búsqueda de reglas de asociación tiene su analogía en el problema de descubrir Itemsets Frecuentes, donde un Itemset Frecuente es un grupo de ítems que aparece en un número suficiente de transacciones de la base de datos.

2.2.3 Búsqueda de Itemsets Frecuentes

Los algoritmos para descubrir Itemsets Frecuentes hacen múltiples pasadas sobre los datos. En la primera pasada, se cuenta el soporte de ítems individuales y se determina cuáles de ellos son frecuentes, es decir, que tienen soporte mínimo. En cada pasada posterior, se comienza con el conjunto inicial de itemsets encontrados como frecuentes en la pasada anterior. Se usa este conjunto inicial para generar nuevos itemsets potencialmente frecuentes, llamados itemsets Candidatos, y se cuenta el soporte actual para estos itemset Candidatos durante la pasada sobre los datos. Al final de la pasada, se determina nuevamente cuál de los itemsets son realmente frecuentes, y éste será el conjunto de inicio de la próxima pasada. Este proceso continúa hasta que no se encuentran nuevos Itemsets Frecuentes.

El problema de descubrir Itemsets Frecuentes puede solucionarse al construir primero un conjunto candidato de itemsets y luego identificar, dentro de este conjunto candidato, aquellos itemsets que cumplen los requerimientos para ser un Itemset Frecuente. Generalmente, esto se hace de forma iterativa para cada *k*-itemset frecuente con un orden creciente de *k*, donde un *k*-itemset frecuente es un Itemset Frecuente con *k* ítems.

En la sección siguiente se describirán algoritmos para descubrir Itemsets Frecuentes, sus orientaciones y evoluciones.

2.2.4 Algoritmos para Descubrir Itemsets Frecuentes

2.2.4.1 Historia de los Algoritmos

Varios Algoritmos con distintas características, algunos mejores que otros, han abierto paso al descubrimiento de información relevante a lo largo del tiempo. Las Reglas de Asociación descriptas anteriormente pueden ser halladas con estos algoritmos. Podemos enumerar algunos: AIS, SETM, Apriori, AprioriTid y DIC entre los modelos tradicionales.

Los algoritmos, por lo general, difieren fundamentalmente en término de cuales itemsets candidatos son contados en una pasada y en la forma en que esos candidatos son generados. Como así también en performance y escalabilidad.

AIS, por ejemplo, genera los itemsets candidados on-the-fly mientras recorre la base de información, luego lee las transacciones y determina cuáles de los itemsets encontrados son frecuentes. Los candidatos son tomados para la próxima pasada y así sucesivamente. Esto lleva a generar demasiados candidatos pequeños que más tarde serán descartados generando desgaste innecesario de procesamiento.

La creación del algoritmo SETM se motivó en el uso de SQL para encontrar Itemsets Frecuentes. Como AIS, genera candidatos on-the-fly leyendo transacciones de una base y por lo tanto también encuentra información que será descartada.

Apriori y su derivado AprioriTid difieren de AIS y SETM en que generan los itemsets candidatos tomando en cuenta los itemsets identificados frecuentes en la pasada anterior y no en las transacciones. Esto mejora la performance, dado que la cantidad de procesamiento se incrementa con el tamaño del problema en rangos que van desde un factor 3 para pequeños problemas a más de un orden de magnitud para grandes problemas. Esta escalabilidad abre un abanico de posibilidades de Mining sobre bases de datos de gran volumen de información y marca también dos factores importantes en el desarrollo de cualquier algoritmo para que sea performante: el número de pasadas sobre los datos y la eficiencia de esas pasadas.

Viendo las desventajas de Apriori es que nació Dynamic Itemset Counting, DIC. Este algoritmo reduce el número de pasadas que se realizan sobre los datos. Se podría decir que DIC trabaja como un tren sobre los datos, con paradas a intervalos de M, los "pasajeros" del tren son itemsets que se mantienen en viaje hasta que cumplieron una vuelta y ya no pueden agregar más información.

Se describen a continuación los principales algoritmos que son la base de esta Investigación: Apriori y Dynamic Itemset Counting.

2.2.4.2 Algoritmo Apriori

Lo primero que cualquier algoritmo debe hacer es descubrir Itemsets Frecuentes, que son los itemsets con soporte mínimo. Luego el algoritmo debe usar los Itemsets Frecuentes para generar las reglas deseadas. Un ejemplo de esto es que si ABCD y AB son Itemsets Frecuentes, entonces se puede determinar si la regla AB $\Rightarrow$ CD se cumple al calcular *conf* = soporte(ABCD) / soporte(AB). Si *conf* $\geq$ *minconf* entonces la regla se cumple. La regla seguramente tendrá mínimo soporte porque ABCD es frecuente.

En el algoritmo Apriori se asume que los ítems en cada transacción se encuentran ordenados en orden lexicográfico. También que la base D de transacciones está normalizada y cada registro es un par <TID, ítem>, dónde TID es el identificador de la transacción correspondiente. El número de ítems en un itemset refleja su *tamaño* y llamamos a un itemset de tamaño *k* un *k*-itemset. La notación c[1]·c[2]·.....·c[k] representa un *k*-itemset c que contiene los ítems c[1],c[2],.....,c[k] donde c[1]<c[2]<.....<c[k]. Si c = X·Y e Y es un *m*-itemset, entonces Y es una *m-extensión* de X. Asociado a cada itemset se tiene un campo contador para almacenar el soporte del itemset que es inicializado en 0 cuando es creado.

La intuición básica lleva a pensar que cualquier subconjunto de un Itemset Frecuente debe ser frecuente. Por lo tanto, los itemsets candidatos que tengan k ítems (*k*-itemset) pueden ser generados al hacer join sobre Itemsets Frecuentes que tengan k-1 ítems (*k-1*-itemset) y dejar de lado aquellos que contengan cualquier subconjunto que no es frecuente. Este procedimiento da como resultado la generación de un número mucho más pequeño de itemsets candidatos que si se realizaran todas las posibilidades de join de todos los itemsets y es el fundamento principal del algoritmo Apriori.

Para descubrir Itemsets Frecuentes, Apriori hace múltiples pasadas sobre los datos. En la primera pasada, cuenta el soporte de ítems individuales y determina cuáles de ellos son frecuentes, es decir que tienen soporte mínimo.

En cada pasada siguiente, comienza con un conjunto inicial de itemsets que fueron encontrados como frecuentes en la pasada anterior. Con este conjunto inicial, genera nuevos potenciales Itemsets Frecuentes, llamados itemsets Candidatos. Cuenta el soporte para estos itemsets candidatos durante la pasada sobre los datos. Al final de la pasada, determina cuáles de los itemsets son realmente frecuentes y serán el conjunto inicio de la próxima pasada. Este proceso continúa hasta que no se encuentran nuevos Itemsets Frecuentes.

Por lo detallado, el proceso básico del algoritmo Apriori es:

La primera pasada del algoritmo simplemente cuenta ocurrencias de ítems para determinar los 1-itemset frecuentes.

Las siguientes pasadas, por ejemplo pasada *k,* consisten en dos etapas: Primero, los Itemsets Frecuentes L_{k-1} encontrados en la (k-1) pasada se usan para generar los itemsets candidatos, denominados C_k, utilizando la función apriori-gen que se detalla posteriormente. Luego, se cuenta el soporte de los candidatos de C_k en la base de datos.

Se necesita para el último paso un conteo rápido y eficiente de los candidatos en C_k que están contenidos en una transacción *t*. Para esto el algoritmo utiliza una función subset y un manejo especial del buffer de almacenamiento de los datos que se detallan más adelante.

2.2.4.2.1 Pseudocódigo

```
1)   L1 = {1-itemsets frecuentes};
2)   for ( k = 2; Lk-1 ≠ ∅; k++ ) do begin
3)       Ck = apriori-gen(Lk-1); // Nuevos candidatos - Ver Función apriori-gen
4)       forall transacciones t ∈ D do begin
5)           Ct = subset (Ck, t); // Candidatos contenidos en t - Ver Función Subset
6)           forall candidatos c ∈ Ct do
7)               c.count++;
8)       end
9)       Lk = {c ∈ Ck | c.count ≥ minsup}
10)  end
11)  Respuesta = ∪k Lk;
```

2.2.4.2.2 Función apriori-gen

Esta función se diseñó para devolver los itemsets candidatos que se pueden generar a partir de un conjunto dado de entrada. Si el argumento de entrada es L_{k-1}, el conjunto de todos los (k-1) Itemsets frecuentes, retorna un superconjunto de los k-itemset frecuentes.

El pseudocódigo de la función se divide en dos partes:

Primero, se hace join de L_{k-1} con L_{k-1}.

```
Insert into Ck
Select p.item1, p.item2,....., p.itemk-1, q.itemk-1
From Lk-1 p, Lk-1 q
Where p.item1 = q.item1,.....,p.itemk-2 = q.itemk-2,p.itemk-1 < q.itemk-1;
```

Segundo, se borran todos los itemset $c \in C_k$ tal que algún (k-1)-subconjunto de c no está en L_{k-1}.

```
Forall itemsets c ∈ Ck do
        Forall (k-1)-subconjuntos s de c do
                If (s ∉ Lk-1) then
                        Delete c de Ck;
```

Este procedimiento tiene como base lo propuesto en [20] donde los pasos son similares a:

$C'_k = \{X \cup X' \mid X, X' \in L_{k-1}, |X \cap X'| = k\text{-}2\}$

$C_k = \{X \in C'_k \mid X \text{ contiene } k \text{ miembros de } L_{k-1}\}$

Un ejemplo para comprender el mecanismo es:

Sea L_3 = {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}, después del paso join, C_4 = {{1 2 3 4},{1 3 4 5}}. El segundo paso borra el itemset {1 3 4 5} dado que no está {1 4 5} en L_3. En C_4 solo quedará {1 2 3 4}.

Esta generación de candidatos, en contraste con la utilizada en los algoritmos AIS y SETM, es más eficiente dado que genera y cuenta menos itemsets (en el caso del ejemplo solo {1 3 4 5}) porque concluye que las otras combinaciones no alcanzan a tener soporte mínimo. En cambio AIS y SETM, en el paso *k,* leen una transacción *t* y determinan cuál de los Itemsets Frecuentes en L_{k-1} están presentes en *t*. Cada uno de estos Itemsets Frecuentes *l* se extiende con todos aquellos ítems frecuentes que están presentes en *t* y ocurren posteriormente en el orden lexicográfico que cualquiera de los ítems en *l* (en el ejemplo anterior si se considera la transacción {1 2 3 4 5}, en la cuarta pasada se generan dos candidatos, {1 2 3 4} y {1 2 3 5}, al extender el Itemset Frecuente {1 2 3}. De la misma manera, un itemset adicional de tres candidatos será generado al extender los otros Itemsets Frecuentes de L_3, llevando a un total de cinco candidatos en consideración en la cuarta pasada. Luego, los que no cumplan las condiciones, serán descartados.

Para ver la correctitud del algoritmo se debe verificar que $C_k \supseteq L_k$. Claramente, cualquier subconjunto de un Itemset Frecuente debe tener soporte mínimo. Por lo tanto, si se extiende cada itemset de L_{k-1} con todos los ítems posibles y después se borran aquellos (k-1)-subconjuntos que no están en L_{k-1}, se obtiene un superconjunto de los itemsets de L_k.

El join es equivalente a extender L_{k-1} con cada ítem de la base y después borrar aquellos itemsets para los cuales el (k-1)-itemset obtenido no está en L_{k-1}. La condición p.item$_{k-1}$ < q.item$_{k-1}$ simplemente asegura que no se generen duplicados. Así, después del paso join, $C_k \supseteq L_k$. Por razonamiento similar, se borran de C_k todos los itemsets cuyos (k-1)-subconjuntos no están en L_{k-1}.

2.2.4.2.3 Función subset

El algoritmo Apriori sugiere almacenar los itemsets candidatos C_k en un hash-tree como sigue: Un nodo del hash-tree contiene una lista de itemsets si es nodo hoja o una tabla de hash si es un nodo interior. En un nodo interior, cada entrada de la tabla de hash apunta a otra tabla de hash. La raíz del hash-tree se define con profundidad 1. Un nodo interior de profundidad d apunta a un nodo de profundidad d+1. Los itemsets son almacenados en las hojas. Para agregar un itemset c, se comienza desde la raíz y baja en el árbol hasta alcanzar una hoja. En un nodo interior de profundidad d se decide qué camino seguir al aplicar una función de hash al d-ésimo ítem del itemset. Todos los nodos son creados inicialmente como nodos hoja. Cuando el número de itemset en una hoja excede un límite específico, el nodo hoja se convierte en un nodo interior.

La función subset encuentra todos los candidatos contenidos en una transacción t como sigue: comienza desde el nodo raíz, si está en una hoja encuentra cuál de los itemsets en la hoja están contenidos en t y agrega las referencias a ellos en el conjunto respuesta. Si está en un nodo interior y lo ha alcanzado al hacer hash sobre el ítem i, hace hash en cada ítem que viene después de i en t y en forma recursiva aplica este procedimiento al nodo en la entrada correspondiente. Para el nodo raíz, hace hash en cada ítem de t.

Para ver por qué la función subset retorna el subconjunto deseado de referencias se considera: para cualquier itemset c contenido en la transacción t, el primer ítem de c debe estar en t. En la raíz, al hacer hash en cada ítem de t, se asegura que sólo ignora los itemsets que comienzan con un ítem que no está en t. Un argumento similar se puede aplicar a profundidades más bajas. Dado que los ítems en un itemset están ordenados, si se alcanza el nodo actual al hacer hash en el ítem i, sólo se necesita considerar los ítems en t que ocurren después de i.

Si k es el tamaño del itemset candidato en el hash-tree, se puede encontrar en tiempo O(k) si el itemset está contenido en una transacción al usar un bitmap temporario. Cada bit del bitmap se corresponde con un ítem. El bitmap se crea una vez por la estructura de datos, y se reinicializa para cada transacción. Esta inicialización toma tiempo O(tamaño(transacción)) para cada transacción.

2.2.4.2.4 Manejo de Buffers

Es importante el manejo de la memoria donde se almacena la información que se genera en cada pasada, puesto que para la etapa de generación de candidatos de la pasada *k*, se necesita almacenar los Itemsets Frecuentes L_{k-1} y también el conjunto de candidatos C_k. En la siguiente etapa, se necesita almacenar C_k y las transacciones de la base.

2.2.4.2.5 Extensiones de Apriori

Una extensión del algoritmo Apriori es AprioriTid, que tiene la propiedad adicional que no toda la base de información se usa para contar el soporte de los itemsets candidatos después de la primer pasada. Para este propósito, se utiliza una codificación de los itemsets candidatos usados en la pasada anterior.

En pasadas posteriores el tamaño de esta forma de almacenar la información se hace mucho más pequeño que la propia base de información, evitando así mucho esfuerzo de lectura.

Apriori en las primeras pasadas es mejor que AprioriTid, sin embargo AprioriTid mejora Apriori en las pasadas posteriores. La razón es que Apriori y AprioriTid utilizan el mismo procedimiento de generación de candidatos y cuentan los mismos itemsets. En las pasadas posteriores, el número de itemsets candidatos se reduce, aunque Apriori examina todas las transacciones en la base. AprioriTid en lugar de buscar en la base de información, busca en la codificación especial de itemsets Candidatos que es menor que el tamaño de la base.

Con estas observaciones se diseñó un algoritmo híbrido, llamado AprioriHybrid que utiliza Apriori en las pasadas iniciales y cambia a AprioriTid cuando encuentra menos candidatos frecuentes en la pasada actual que en la anterior. Es decir, no se utiliza el mismo algoritmo en todas las pasadas sobre los datos.

Cambiar de un algoritmo a otro implica un costo que vale la pena cuando la base de información es muy grande.

2.2.4.3 Algoritmo DIC (Dynamic Itemset Counting)

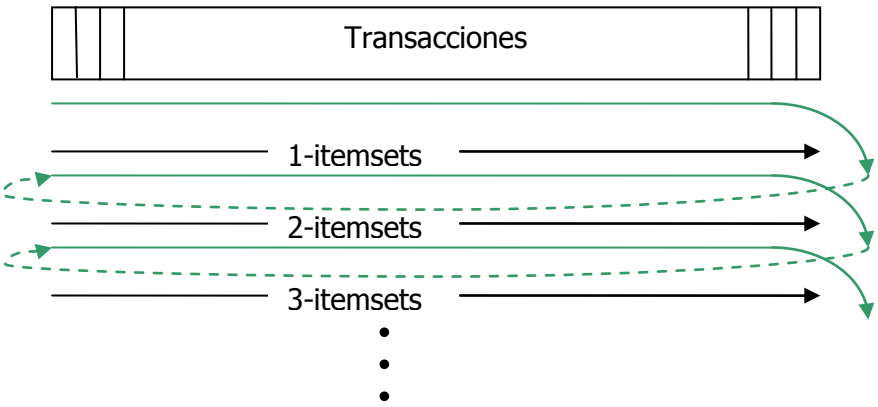
Como se mencionó anteriormente, al algoritmo DIC se lo puede asemejar a un "tren de pasajeros", donde los "pasajeros" son itemsets y se cuentan sus ocurrencias mientras permanecen a bordo, el "tren" se detiene a intervalos de M transacciones y cuando alcanza el final de las transacciones comienza nuevamente desde el principio.

DIC permite que un itemset pueda ser contado en cualquier momento del viaje y dejado de contar en ese mismo momento pero en la pasada siguiente. Por ejemplo, si tenemos una

base de 40.000 transacciones, M = 10.000, se contarán los 1-itemsets en las primeras 40.000 transacciones, los 2-itemsets se comenzarán a contar después de las 10.000 transacciones, los 3-itemsets comenzarán después de las 20.000. Al final de las transacciones se deben dejar de contar los 1-itemsets y se comenzará en el principio del archivo con los 2 y 3-itemsets. Después de 10.000 transacciones se dejarán de contar los 2-itemsets y así siguiendo. Al final se habrá realizado una pasada y media sobre los datos.

La figura 2.3 compara gráficamente los mecanismos de los algoritmos Apriori y DIC.

Apriori (3 pasadas)



DIC (1.5 pasadas)

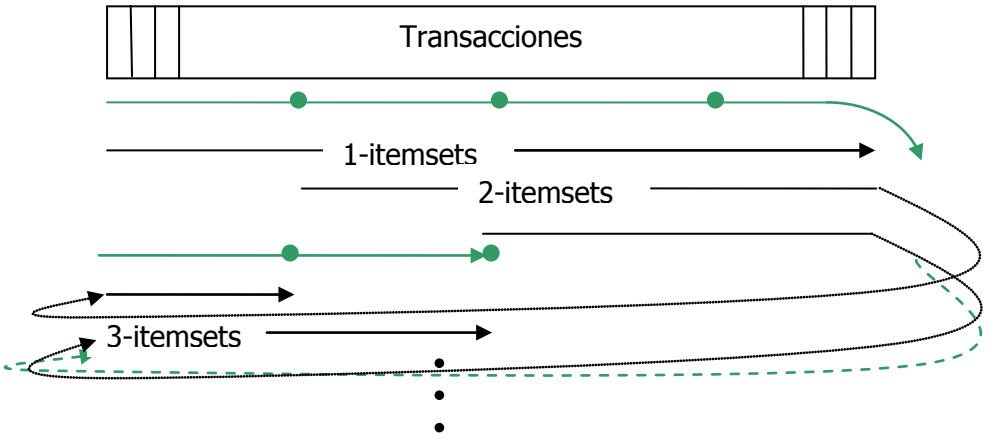


Figura 2.3: Comparación de Procesos DIC y Apriori

Al analizarse todos los Itemsets Frecuentes a la vez se deben encontrar y contar no sólo los Itemsets Frecuentes sino también los Itemsets no Frecuentes. DIC propone 4 formas distintas de marcar los itemsets para este propósito:

a. Rectángulo Sólido: Es un Itemset Frecuente, se terminó de contar y supera el soporte requerido.

b. Círculo Sólido: Es un Itemset no Frecuente, se terminó de contar y no supera el soporte requerido.

c. Rectángulo Punteado: Es un posible Itemset Frecuente, aún no se terminó de contar y supera el soporte requerido.

d. Círculo Punteado: Es un posible Itemset no Frecuente, aún no se terminó de contar y no supera el soporte requerido.

Con esta clasificación de estados de los itemsets, DIC trabaja de la siguiente manera:

1. Se comienza con un itemset vacío que es marcado como rectángulo sólido. Los 1-itemsets son marcados como círculos punteados. Los demás itemsets no se marcan. (Figura 2.4).
2. Se leen M transacciones. Por cada transacción se incrementan los contadores de los itemsets punteados.
3. Si un círculo punteado excede el soporte, se pasa al estado de rectángulo punteado. Si todos los conjuntos inferiores de un itemset están como rectángulos sólidos o punteados, entonces se marca a éste con círculo punteado y se le agrega un contador. (Figura 2.5 y 2.6)
4. Si un itemset es punteado y se terminó de contar entonces se lo marca como sólido y se lo deja de contar.
5. Si se llegó al final de las transacciones, se comienza nuevamente desde el principio (Figura 2.7).
6. Si queda algún itemset punteado, ir al paso 2.

En resumen, DIC comienza a contar 1-itemsets y rápidamente cuenta 2,3,4,…,*k*-itemsets obteniendo así en pocas pasadas los itemsets resultados.

Si la información es homogénea y el intervalo M elegido es razonablemente pequeño, DIC es más performante que Apriori dado que realizará un orden de dos pasadas mientras que Apriori debe realizar tantas pasadas como el tamaño máximo de un itemset candidato. Si la información no es homogénea el orden de pasadas es aleatorio.

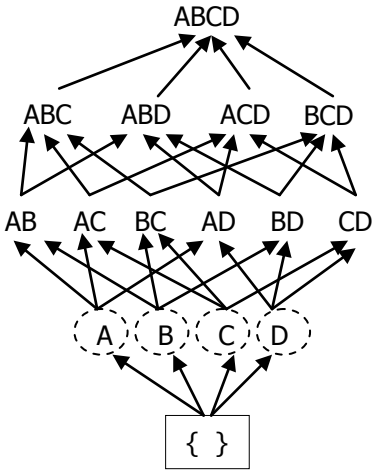


Figura 2.4: Comienzo de Alg. DIC

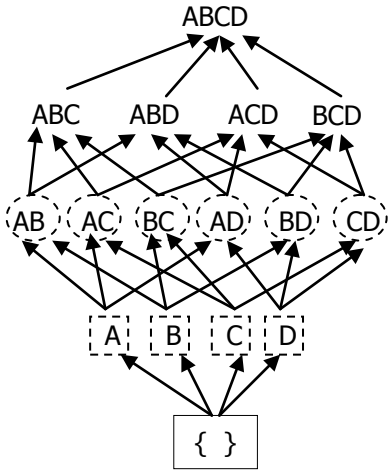


Figura 2.5: Después de M transacciones

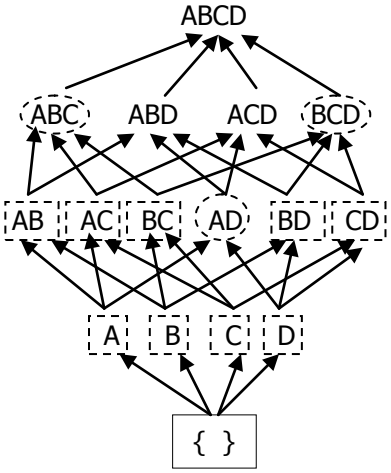


Figura 2.6: Después de 2M transacciones

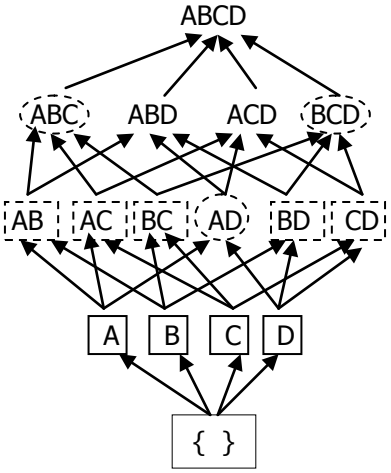


Figura 2.7: Después de 1 pasada

2.2.5 Algoritmo para Descubrir Reglas de Asociación

Para generar reglas para todos los Itemsets Frecuentes *l* se deben buscar todos los subconjuntos no vacíos de *l*. Para cada uno de tales subconjuntos *a*, se le aplica una regla de la forma $a \Rightarrow (l\text{-}a)$ si soporte(*l*)/soporte(*a*) es al menos *minconf*. Se consideran todos los subconjuntos de *l* para generar reglas con múltiples consecuentes.

Se recomienda guardar los Itemsets Frecuentes en estructuras de acceso fácil y rápido como pueden ser tablas de hash, pues el acceso a encontrar los soportes para los subconjuntos debe ser eficiente.

El procedimiento anterior se puede mejorar planteando la generación de los subconjuntos de un itemset Frecuente en forma recursiva Depth First. Por ej. dado un itemset ABCD, primero se considera el subconjunto ABC, después AB, etc. Entonces si un subconjunto *a* de un subconjunto *l* no genera una regla, los subconjuntos de *a* no necesitan ser considerados

para generar reglas que usen *l*. Por ej. si ABC ⇒ D no tiene suficiente confidencia, no se necesita chequear si vale AB ⇒ CD. Con este mecanismo no se pierde ninguna regla dado que el soporte de cualquier subconjunto *ã* de *a* debe ser tan grande como el soporte de *a*. La confidencia de la regla ã ⇒ (*l*-*ã*) no puede ser mayor que la confidencia de *a* ⇒ (*l*-*a*). Si *a* no produce una regla que involucre todos los ítems en *l* con *a* como el antecedente, tampoco *ã* lo hará. El siguiente algoritmo resume estas ideas.

```
// Algoritmo simple
forall Itemsets Frecuentes l_k, k ≥ 2 do
        call genrules(l_k, l_k);
// Genrules genera todas las reglas válidas ã ⇒ (l_k - ã) para todos ã ⊂ a_m
procedure genrules (l_k: k-itemsets frecuentes, a_m:m-itemsets frecuentes)
1)      A = {(m -1)-itemsets a_m-1 | a_m-1 ⊂ a_m };
2)      forall a_m-1 ∈ A do begin
3)              conf = soporte(l_k) / soporte(a_m-1);
4)              if ( conf ≥ minconf) then begin
7)                  output la regla a_m-1 ⇒ (l_k - a_m-1),
                        con confidencia = conf y soporte = soporte (l_k);
8)                  if (m - 1 > 1) then
9)                      call genrules(l_k,a_m-1);       // para generar reglas con subconj. de a_m-1
                                                        //como antecedentes
10)             end
11)     end
```

2.2.5.1 Un Algoritmo más Rápido

Se dijo que si *a* ⇒ (*l*-*a*) no vale, tampoco lo hará ã ⇒ (*l*-*ã*), para cualquier *ã* ⊂ *a.* De otra forma, para que valga la regla (*l*-*c*) ⇒ *c*, todas las reglas de la forma (*l*-∼*c*) ⇒ ∼c deben valer, considerando a ∼*c* como subconjunto no vacío de *c*. Por ej. si la regla AB ⇒ CD vale, entonces las reglas ABC ⇒ D y ABD ⇒ C también deben valer.

Con la propiedad anterior, para un Itemset Frecuente dado, si una regla con consecuente *c* es válida entonces también lo son las reglas con consecuentes que son subconjuntos de *c*. Esto es similar a la propiedad que si un itemset es frecuente también lo son todos sus subconjuntos. De un Itemset Frecuente *l*, primero se generan todas las reglas con un ítem en el consecuente. Luego se usan los consecuentes de estas reglas y la función apriori-gen para generar todos los posibles consecuentes con dos ítems que pueden aparecer de una regla generada desde *l*, etc. Las reglas que tienen algún ítem como consecuente generado en el paso 2 de este algoritmo pueden ser encontradas al utilizar una versión modificada de la función genrules en la cual los pasos 8 y 9 son eliminados para evitar llamados recursivos.

```
// Algoritmo más rápido
1)      forall large k-itemsets l_k, k ≥ 2 do begin
2)      H1 = {consecuentes de reglas derivadas de l_k con un ítem en el consecuente};
3)              call ap-genrules(l_k, H_1);
4)      end

procedure ap-genrules (l_k: k-itemsets large, H_m: conjunto de consecuentes de m-items)
if (k > m + 1) then begin
        H_{m+1} = apriori-gen (H_m);
        forall h_{m+1} ∈ H_{m+1} do begin
                conf = soporte(l_k) / soporte(l_k - h_{m+1});
                if ( conf ≥ minconf) then begin
                  output la regla (l_k - h_{m+1}) ⇒ h_{m+1},
                                con confidencia = conf y soporte = soporte (l_k);
                else
                  delete h_{m+1} de H_{m+1};
        end
        call ap-genrules(l_k,H_{m+1});
end
```

A modo de ejemplo sobre las ventajas de este algoritmo, se tiene un Itemset Frecuente ABCDE. Asumimos que ACDE $\Rightarrow$ B y ABCE $\Rightarrow$ D son las únicas reglas con consecuente de un ítem derivadas del mencionado itemset que tienen confidencia mínima. El algoritmo simple, genrules(ABCDE,ACDE) analizará si las reglas con consecuente de dos ítems ACD $\Rightarrow$ BE, ADE $\Rightarrow$ BC, CDE $\Rightarrow$ BA y ACE $\Rightarrow$ BD valen. La primera de estas reglas no puede valer, dado que E $\subset$ BE, y ABCD $\Rightarrow$ E no tiene confidencia mínima. La segunda y tercer regla no pueden valer por razones similares. La llamada a genrules(ABCDE,ABCE) analizará si las reglas ABC $\Rightarrow$ DE, ABE $\Rightarrow$ DC, BCE $\Rightarrow$ DA y ACE $\Rightarrow$ BD valen, y encontrará que las primeras tres de estas reglas no valen. De hecho, la única regla con consecuente de dos ítems que puede posiblemente valer es ACE $\Rightarrow$ BD, donde B y D son los consecuentes en las reglas válidas con consecuentes de un ítem. Esta es la única regla que será analizada por el algoritmo rápido.

Capítulo 3

Reglas de Asociación Temporales

3.1 Motivación

Como se mencionó anteriormente, una de las principales aplicaciones de Data Mining es el análisis de datos transaccionales, donde cada transacción está formada por una colección de ítems. Las reglas de asociación proponen capturar la co-ocurrencia de ítems en las transacciones.

Por ejemplo, si se observa la base de transacciones de un supermercado, se puede ver que *Pan Dulce y Sidra* raramente se venden juntos. Sin embargo, si sólo se miran las transacciones en la semana anterior a Navidad se puede descubrir que la mayoría de las transacciones contienen *Pan Dulce y Sidra*; es decir, *"Pan Dulce ➔ Sidra"* tienen soporte y confidencia altos en la semana anterior a Navidad.

Lo anterior sugiere que se pueden descubrir diferentes reglas de asociación si se consideran diferentes intervalos de tiempo. Algunas reglas de asociación pueden valer durante algunos intervalos, y no durante otros.

Descubrir intervalos temporales y las reglas de asociación que valen en esos momentos, puede llevar a obtener información útil. Por ejemplo: al considerar cada paquete IP (Internet Protocol) de una red como una transacción y los atributos del header IP como ítems de la transacción, se pueden utilizar las reglas de asociación temporales para representar la actividad normal de la red en diferentes períodos del día. Con esto se pueden prevenir ataques a la red cuando se detecta un comportamiento diferente del normal.

Otro ejemplo que se puede destacar, y regresando a las transacciones en un supermercado, es que "cerveza ➔ papas fritas" tiene soporte 3% y confidencia 87%. Pero, si se segmenta la información entre las 18:00 Hs. y 21:00 Hs., se puede ver que el soporte para cerveza y papas fritas sube generalmente a 50%.

De los ejemplos anteriores, se puede deducir que aunque las reglas de asociación tengan soporte y confidencia superiores al mínimo sugerido por el usuario dentro del período entero considerado, analizar los datos en detalle puede revelar que las reglas de asociación existen sólo en ciertos intervalos, y no en los restantes.

Al analizar la base de datos de un supermercado, se asume que cada transacción contiene los ítems comprados por un cliente y el momento en que se realizó la compra, lo que se

denominará *transaction time*. A veces, también se logra registrar alguna identificación del cliente.

En grandes bases de datos, también se puede encontrar información relacionada con productos que no necesariamente estuvieron disponibles durante todo el período de recolección. Se pueden encontrar algunos productos que, al momento de realizar el mining, están discontinuados. También puede haber productos que comenzaron a venderse después del comienzo de recolección. Algunos de estos productos deberían participar en las asociaciones encontradas, pero no se consideran en las reglas por las restricciones de soporte. Además, se deberían considerar algunos productos que pueden ser frecuentes sólo en subintervalos estrictamente contenidos en el período de vida, pero no en el intervalo entero. Ejemplos reales de estos productos son Reproductores de DVD, diskette 5 ¼, bolsas para freezer, televisores blanco y negro, televisores color.

Este problema se soluciona al incorporar el tiempo en el modelo de descubrimiento de reglas de asociación. Estas nuevas reglas se denominan ***Reglas de Asociación Temporales*** [1].

Una particularidad de esta idea es la posibilidad de eliminar reglas fuera de fecha, de acuerdo al criterio del usuario. Más aún, es posible borrar conjuntos de ítems obsoletos en función de su tiempo de vida, reduciendo la cantidad de trabajo a realizar en la determinación de los ítems frecuentes y las reglas.

Las reglas de asociación temporales son una extensión del modelo no temporal. Para conjuntos frecuentes de ítems o itemsets, la idea básica es limitar la búsqueda al tiempo de vida de los miembros del itemset. Por otro lado, para evitar considerar frecuente un itemset con un período de vida muy corto, por ejemplo un ítem que se vendió una sola vez, se introduce el concepto de ***soporte temporal***. Así, cada regla tiene un lapso de tiempo asociado, que se corresponde con el tiempo de vida de los ítems que participan de la regla. Si la extensión del tiempo de vida de una regla excede un mínimo estipulado por el usuario, se debe analizar si la regla es frecuente en ese período. Este concepto permite encontrar reglas que, con el punto de vista tradicional de la frecuencia, no hubiera sido posible descubrir.

3.2 Diversos Enfoques

Trabajos anteriores sobre Data Mining, que incluyen aspectos temporales, generalmente están relacionados a la secuencia de análisis de eventos [4], [6], [13]. El objetivo de estos es descubrir regularidad en la ocurrencia de ciertos eventos y relaciones temporales entre los diferentes eventos.

En particular en [13] se trata el problema de reconocer episodios frecuentes en una secuencia de eventos. Un episodio se define como una colección ordenada de eventos que ocurren uno cerca del otro en un orden dado, durante intervalos de tiempo de un tamaño específico definido por el usuario. Cada evento tiene asociado el momento en que ocurrió. Para ser considerados interesantes, los eventos de un episodio deben ocurrir en el tiempo, lo suficientemente juntos como el usuario lo defina.

Mientras tanto, [4] revisa el problema de descubrir patrones secuenciales en bases de datos transaccionales. La solución consiste en crear una secuencia para cada cliente y buscar patrones frecuentes dentro de cada secuencia. Cada transacción contiene: la fecha en que se realizó, la lista de ítems comprados y alguna identificación del cliente. Todas las transacciones de un cliente forman una secuencia, donde cada transacción contiene un conjunto de ítems, y la lista de transacciones ordenada por fecha, corresponde a una secuencia. Dada una base de datos de transacciones, Mining Sequential Patterns es encontrar las secuencias maximales que alcanzan un soporte mínimo especificado por el usuario. El soporte de una secuencia se define como la cantidad total de clientes que soportan la secuencia. Cada una de estas secuencias maximales, representa un patrón secuencial.

En [6] se consideran patrones más complejos que en los casos mencionados anteriormente, dado que no considera definido por el usuario el lapso de tiempo. En estos casos se analizarán distancias temporales con múltiples granularidades. Se introduce la noción de una estructura de evento que es esencialmente un conjunto de restricciones temporales sobre un conjunto de variables que representan los eventos. Cada restricción limita la distancia entre un par de eventos en términos de la granularidad del tiempo.

Los autores de [8], de forma totalmente diferente, utilizan el principio de longitud de mínima descripción junto con una escena de codificación, para analizar la variación de la correlación entre ítems a través del tiempo. Este análisis tiene como objetivo extraer patrones temporalmente sorpresivos, y es un intento de sustitución del rol del dominio de conocimiento en la búsqueda de patrones interesantes. Cuando el analista está familiarizado con los patrones prevalecientes en los datos, el mayor beneficio se da en los cambios en las relaciones entre la frecuencia de los ítems en el tiempo. No necesariamente un conjunto de *k* ítems se declara "interesante" porque su soporte absoluto supera el mínimo impuesto por el usuario, sino también porque las relaciones entre los ítems varían en el tiempo. Incluso si el soporte de un itemset varía en el tiempo, no se considera interesante si los cambios se producen por los cambios en el soporte de subconjuntos de ítems más pequeños.

Esta visión, comparada con los algoritmos estándar en los cuales las operaciones principales son la generación de patrones y conteo de tuplas, es más costosa en cuanto a cálculos

computacionales. Pero esta diferencia se compensa con el tiempo utilizado por los analistas de los resultados para descartar las reglas.

En [14] los autores estudian el problema de reglas de asociación que existen en ciertos intervalos de tiempo y muestran variaciones cíclicas regulares a través del tiempo. Presentan algoritmos para descubrir eficientemente lo que llaman "Reglas de Asociación Cíclicas" donde se asume que los intervalos de tiempo son especificados por el usuario. Definen una regla de asociación como cíclica si tiene soporte y confidencia mínima en intervalos regulares de tiempo. Tratan las reglas de asociación y los ciclos independientemente. Aplican alguno de los algoritmos conocidos para descubrir reglas en cada segmento de datos y posteriormente aplican algoritmos de "pattern matching" para detectar los ciclos.

En [17] los autores estudian cómo varían en el tiempo las reglas de asociación, generalizándolo en [14]. Introducen la noción de utilizar un álgebra calendario para describir el fenómeno de interés de los usuarios y presentan algoritmos para descubrir "Reglas de Asociación Calendric", esto es reglas de asociación que siguen los patrones temporales informados en las expresiones calendario provistas por el usuario. Una regla de asociación es calendric si la regla tiene el soporte y confidencia mínima durante cada unidad de tiempo contenida en un calendario y permite un límite de error para representar el hecho de que, en la vida real, las reglas de asociación se cumplen la mayoría de las veces, pero no siempre. Tratan las reglas de asociación y los ciclos independientemente. Aplican alguno de los algoritmos conocidos para descubrir reglas en cada segmento de datos y posteriormente aplican algoritmos de "pattern matching" para detectar calendarios en las reglas de asociación.

La investigación [11] sugiere expresiones de calendario para representar reglas temporales y presenta sólo ideas básicas de los algoritmos de descubrimiento de reglas temporales.

La base de este trabajo es el modelo de reglas de asociación temporales basado en lifespans ([1] y [2]), que se detalla a continuación y difiere de los otros en que no es necesario definir intervalos o calendarios porque se asume que el período de vida de un ítem, o lifespan, es intrínseco en los datos.

3.3 El Enfoque del Lifespan de los Itemsets

El período de vida de un ítem o itemset, también denominado Lifespan, está delimitado por la primera y última vez que éste aparece en las transacciones de la base.

Con este nuevo atributo, el lifespan, es necesario definir el soporte de un itemset en el intervalo de tiempo definido por su lifespan.

3.3.1 Modelo Formal

Sea T = {. . ., t_0, t_1, t_2, . . . } un conjunto de tiempos, contablemente infinito, sobre el cual se define el orden lineal $<_T$, donde $t_i <_T t_j$ significa que t_i ocurre antes o es anterior que t_j. Se asume que T es isomórfica a N (números naturales) y se restringe la atención a los intervalos cerrados $[t_i,t_j]$.

Sea $I=\{i_1,....,i_p\}$, donde los i_j's son ítems, D es una colección de subconjuntos de I denominada base de transacciones. Cada transacción S en D es un conjunto de ítems tal que $S \subseteq I$. Asociado a S tenemos el timestamp t_s, que representa el momento o tiempo de la transacción S.

Cada ítem tiene un período de vida o lifespan en la base de datos, que representa explícitamente la duración temporal de información del ítem, o sea el tiempo en el cual el ítem es relevante para el usuario. El lifespan de un ítem i_j está dado por el intervalo $[t_1,t_2]$, con $t_1 <_T t_2$, donde t_1 es el timestamp de la primer transacción en D que contiene i_j y t_2 es el timestamp de la última transacción en D que contiene i_j.

Sea i_j un ítem de I. Con cada ítem i_j y la base de datos D, se asocia un lifespan definido por un intervalo de tiempo $[i_j.t_1,i_j.t_2]$ o simplemente $[t_1,t_2]$ si i_j es conocido. l: $i_j \rightarrow 2^T$ es una función que asigna un lifespan a cada ítem i_j en I que se denominará como li_j. Entonces, l_d, es el lifespan de D, tal que

$l_d = \cup li_j$, $\forall$ j.

Como se mencionó anteriormente un itemset es un conjunto de ítems. Formalmente, sea X $\subseteq I$ un conjunto de ítems, S contiene a X, o X se verifica en S, si X $\subseteq$ S. El conjunto de transacciones en D que contienen X es indicado por V(X,D) = { S | S $\in$ D $\wedge$ X $\subseteq$ S} (se omite D por claridad). Si la cardinalidad de X es *k*, X es llamado un *k*-itemset.

Se puede estimar el lifespan de un k-itemset X, para k>1, con [t,t'] donde

t = max {t_1 | $[t_1,t_2]$ es el lifespan de un ítem i_j en X} y

t' = min {t_2 | $[t_1,t_2]$ es el lifespan de un ítem i_j en X}.

Sea X $\subseteq$ I un conjunto de ítems y l_X su lifespan. Si D es el conjunto de transacciones de una base de datos, entonces D_{lX} es el subconjunto de transacciones de D cuyos timestamps $t_i \in l_X$. Con $|D_{lX}|$ se indica la cantidad de transacciones de D_{lX}.

El soporte de una colección de ítems X en una base de transacciones es la frecuencia de ocurrencias de X, como ya se describió. Para incluir los ejemplos enunciados hay que extender la definición. ***Soporte temporal*** se define como la amplitud del lifespan del

itemset. El límite del soporte temporal τ, si l_D es el lifespan de la base de datos y $|l_D|$ su duración, es una fracción de $|l_D|$.

Formalmente la nueva definición queda: El soporte de X en D sobre su lifespan l_X, denotado $s(X,l_X)$ (nuevamente omitimos D para claridad), es la fracción de transacciones en D que contienen a X en todo el intervalo de tiempo correspondiente a l_X:$|V(X,D)|/|D_{lX}|$. Para cada subintervalo [t,t'] calculamos el soporte como $|V(X,[t,t'])|/|D[t,t']|$. Dado un límite de soporte $\sigma \in [0,1]$ y un límite de soporte temporal τ, X es frecuente en su lifespan l_X si $s(X,l_X) \geq \sigma$ y $|l_X| \geq \tau$. En este caso, decimos que X tiene soporte mínimo en l_X.

La incorporación del tiempo permite determinar si un itemset es frecuente al calcular la proporción entre el número de transacciones que contienen al itemset y el número de transacciones en la base de datos tal que su vigencia esté incluido en el lifespan del itemset.

También permite filtrar los ítems, y los itemsets, con períodos de vida muy cortos, como por ejemplo aquellos que se han vendido solamente una vez.

Por otro lado, el usuario podría especificar un instante del tiempo t_0, tal que cualquier ítem cuyo lifespan es $[t_1,t_2]$ y $t_2 < t_0$ se considera obsoleto.

Una *Regla de Asociación Temporal* para D es una expresión de la forma:

$X \rightarrow Y\,[t_1,t_2]$, donde $X \subseteq I$, $Y \subseteq I \backslash X$, y $[t_1,t_2]$ es una porción de tiempo correspondiente al lifespan de $X \cup Y$ expresada en una granularidad determinada por el usuario.

Una regla de asociación temporal tiene asociados tres factores: soporte, soporte temporal que ya fue definido y confidencia que se define como:

La ***confidencia*** de una regla $X \rightarrow Y\,[t_1,t_2]$, denotado por $conf(X \rightarrow Y,[t_1,t_2],D)$ es la probabilidad condicional de que una transacción de D, seleccionada al azar en el período de tiempo $[t_1,t_2]$, que contiene a X también contenga a Y. Se expresa como:

$Conf(X \rightarrow Y, [t_1,t_2],D) = s(X \cup Y, l_{XUY},D) / s(X, l_{X\cup Y},D)$, donde $l_{X\cup Y} = \{[t_1,t_2]\}$.

La regla de asociación temporal $X \rightarrow Y\,[t_1,t_2]$ vale en D con soporte s, soporte temporal $|l_{XUY}|$ y confidencia c, si el s% de las transacciones de D contienen X U Y y además, el c% de las transacciones de D que contienen X también contienen Y en el período de tiempo $[t_1,t_2]$.

Dado un conjunto de transacciones D, niveles mínimos de soporte, soporte temporal y confidencia, el problema de descubrir reglas de asociación temporales es generar todas las reglas de asociación que tienen, al menos, el soporte, soporte temporal y confidencia dados.

3.3.2 Descubrir Reglas de Asociación Temporales

Como se mencionó en 2.2.2, descubrir las reglas de asociación de un conjunto de transacciones D, puede hacerse en dos pasos: Primero descubrir los itemsets frecuentes y segundo utilizar esos itemsets para encontrar las reglas.

Este procedimiento puede modificarse para encontrar reglas de asociación temporales:

- Fase 1: Encontrar todos los itemsets X tal que X sea frecuente en su lifespan l_X, o sea $s(X,l_X,D) \geq \sigma$ y $|l_X| \geq \tau$.
- Fase 2: Utilizar los itemsets frecuentes X para encontrar las reglas: verificar para cada $Y \subset X$, con $Y \neq \emptyset$, si la regla $X \setminus Y \rightarrow Y[t_1,t_2]$ se satisface con confidencia suficiente, en otras palabras, excede la confidencia mínima θ establecida en el intervalo $[t_1,t_2]$.

3.3.2.1 Generación de Itemsets Frecuentes

El modelo temporal (TDIC) que se mostrará es una extensión del modelo no-temporal (DIC) y se basa en la idea de limitar la búsqueda de conjuntos de ítems frecuentes en el tiempo de vida del itemset. Con este concepto se logran encontrar itemsets que no serían tomados en cuenta en el modelo no-temporal.

En el desarrollo del algoritmo el tiempo de vida de un itemset incluye un intervalo en el cual el soporte temporal es maximal y además frecuente. Esto permite detectar itemsets no frecuentes en el tiempo de vida entero (es decir el tiempo de toda la información) pero sí en ciertos subintervalos.

Se introduce el concepto de historia de un itemset que sirve para calcular el subintervalo contenido en su tiempo de vida y analizar su comportamiento para poder compararlo con el de otros itemsets.

El algoritmo comienza con una base de transacciones formada por un tupla <elem, t>, donde elem corresponde a un conjunto de ítems y t es el tiempo o momento de ocurrencia de la transacción.

Con la incorporación del tiempo se puede determinar si un itemset es frecuente, calculando una proporción entre el número de transacciones que contienen al itemset y el número de transacciones de la base de datos.

El soporte temporal es la amplitud del tiempo de vida del itemset.

Un itemset puede no ser frecuente en el intervalo correspondiente a su período de vida entero pero sí serlo en algún subintervalo de este período. Entonces, podría participar en

reglas interesantes en estas porciones de su tiempo de vida. Estos subintervalos no dependen estrictamente de los datos pero sí de los parámetros que pueda asignar el usuario.

TDIC comienza buscando los 1-itemsets y después de M transacciones analiza si hay frecuentes. Luego combina los 1-itemsets frecuentes para formar los 2-itemsets. Al final de la porción M leída, se analizan sus lifespan y se verifican quienes son frecuentes. Estos formarán los 3-itemsets y así siguiendo. Cuando el algoritmo completa una pasada sobre los datos, comienza desde el principio nuevamente mientras que se hayan generado nuevos itemsets. Los itemsets que han dado una vuelta a los datos se dejan de contar.

El lifespan de un *k*-itemset, con $k > 1$, se obtiene de la siguiente manera: Si el *k*-itemset u se obtiene del (*k*-1)-itemset v y el (*k*-1)-itemset w. El lifespan de u es $[u.t_1,u.t_2]$ donde $u.t_1$= max $[v.t_1,w.t_1]$ y $u.t_2$= min $[v.t_2,w.t_2]$.

3.3.2.1.1 Pseudocódigo

```
pass ← 1;
kf ← 0;
k ← 1;
while kf <> k do
 begin
   kf := k;
   while not end of d do begin
        read dM;  /* dM es una porción de d ordenada por tiempo*/
        foreach s ∈ dM do begin
             if pass = 1 then
                 mirar si cada 1-itemset c ∈ s esta en C1, si no esta insertar c en C1 y TRAIN;
             Cs ← subset(TRAIN,s); /*Cs contiene todos los itemset c de TRAIN en los cuales está s*/
             foreach c ∈ Cs do begin
                 c.Fr ← c.Fr + 1;
                 if s.t > c.t2 then begin
                       c.t2 ← s.t;
                       FTr ← FTr + FTr';
                       FTr' ← 0;
                 endif;
             end;
             foreach c ∈ TRAIN do
                 if s.t ∈ [c.t1,c.t2] then
                    c.FTr' ← c.FTr' + 1;
                 else
                    Si c es un 1-itemset agrego 1 a c.FTr';
              end;
        end; /* s ∈ dM */
        for j = pass to k do begin
            LL = {c/c ∈ Cj ∧ c.Fr >= σ x FTr ∧ (c.t2 – c.t1 + 1) >= τ}
                                                    /* obtiene los j-itemsets frecuentes*/
            if |Lk| > 1 then
                  Lj' ← LL – Lj;
            else
                  Lj' ← LL; /*Lj' contiene los nuevos j-itemsets */
            Lj ←LL;
```

```
        endfor;
        if |Lk| > 1 then
            k ← k + 1;
        for j = pass +1 to k do begin
           Cj' ← apriori-gen(Lj-1,Lj-1');
           if Lj-1 <> Lj-1' then
                 Cj' ← Cj' ∪ apriori-gen (Lj-1',Lj-1);
           Cj ←Cj ∪ Cj';
           TRAIN ← TRAIN ∪ Cj';
        endfor;
        foreach c ∈ TRAIN do          /* Este paso borra de TRAIN todos los itemset */
           if c.complete = YES then   /* que tienen completa una pasada sobre d
                 delete c from TRAIN;
   endwhile; /*not end of d */
   pass ← pass + 1;
   prune-a-posteriori(Lpass);
endwhile; /* kf <> k*/
for j = pass + 1 to k do
   a-posteriori(Lj);
endfor;
F = ∪ i=1,..k Li
```

a-posteriori(L_{pass}) encuentra los itemsets frecuentes tales que no son frecuentes en todo su lifespan, pero si en algún subintervalo maximal
apriori-gen(L,L') es una sutil modificación de apriori-gen definido en 2.2.4.2.2

3.3.2.2 Generación de Reglas de Asociación Temporales

Para generar reglas, es necesario encontrar todos los subconjuntos para todos los Itemsets Frecuentes. Entonces, dado un itemset frecuente Z se debe encontrar, para cada subconjunto X de Z, las reglas X ➔ (Z - X) $[t_1, t_2]$ tal que $s(Z,l_Z,D) / s(X,l_Z,D) \geq \theta$.

Uno de los problemas, al calcular la confidencia, $Conf(X \Rightarrow Y,[t_1,t_2],D) = s(X \cup Y,l_{XUY},D) / s(X,l_{X\cup Y},D)$, donde $l_{X\cup Y}= \{[t_1,t_2]\}$, es la determinación de $s(X, l_{X\cup Y}, D)$. Evidentemente, puede no ser igual a $s(X, l_X,D)$, ya que $l_{X\cup Y} \subseteq l_X$. Pero, en la primer fase se calcula $s(X, l_X, D)$, y no $s(X, l_{X\cup Y}, D)$. Si XY es un itemset frecuente de tamaño k, tendremos 2^k subconjuntos posibles; entonces se debería recalcular la frecuencia para 2^k-2 itemsets en $l_{X\cup Y}$, y repetirlo en cada *k*-ésima pasada, con $k > 1$. Una forma de evitar esto, es utilizar una estimación. El caso más simple, es considerar que los itemsets tienen distribución temporal uniforme, la posibilidad de que aparezcan en cualquier subconjunto de l_X, en particular en $l_{X\cup Y,}$ será la misma. Entonces, se puede estimar $s(X, l_{X\cup Y}, D)$ como $s(X, l_X, D)$. Al modificar el algoritmo propuesto en 2.2.5 se puede obtener las reglas temporales de forma inmediata.

Capítulo 4

Experimentación

4.1 Objetivo

Uno de los principales objetivos del presente trabajo es la comparación entre los algoritmos investigados y desarrollados. Para ello se realizaron una serie de experimentaciones con distintas muestras de datos generados en forma sintética con las cuales se trató de buscar performance en tiempos y resultados.

4.2 Generación de Datos Sintéticos

Para poder evaluar los algoritmos es necesario, de alguna manera, hacer una imitación del mundo real de las transacciones que se realizan en el ambiente de las ventas. Para ello, a continuación, se plantea un mecanismo de generación de datos imitación o sintéticos que simulan la realidad.

En la creación de datos sintéticos se distinguen dos grandes actividades: la generación del conjunto de transacciones que estarán formadas por los elementos que se compran, según el modelo presentado en [3], y la generación de los momentos en los que las transacciones se realizan.

4.2.1 Generación del Conjunto de Transacciones

Se puede afirmar que la gente tiende a comprar ítems juntos o agrupados. Como por ejemplo, las medialunas y café con leche mencionados en el Capítulo 1. Cada conjunto de estos elementos comprados a la vez es un potencial Itemset Frecuente.

Una transacción de compra puede tener más de un elemento considerado Itemset Frecuente. Casi siempre el tamaño de la transacción o compra es pequeño, es decir que muy pocas transacciones tienen muchos ítems. Por consiguiente, los Itemsets Frecuentes que se puedan deducir de una base de transacciones estarán formados por pocos ítems, y sólo algunos de muchos ítems.

Los parámetros que se deben considerar en la generación de datos sintéticos son: el número de transacciones que se desean generar, el tamaño promedio de estas transacciones, el tamaño promedio de los potenciales Itemsets Frecuentes, la cantidad de potenciales

Itemsets Frecuentes que se generarán y por supuesto, la cantidad de ítems de los que se dispone para armar las transacciones.

En el proceso de generación de datos se parte de una base *T* de Itemsets Frecuentes generados. La cantidad de itemsets en la base está dada por un parámetro definido por el usuario. Existe una relación inversa entre la cantidad de potenciales Itemsets Frecuentes y el soporte promedio para los mismos. El tamaño de un itemset en *T* estará dado por una distribución de Poisson con media μ igual al tamaño promedio de los potenciales Itemsets Frecuentes.

Los elementos del primer itemset de la base, son elegidos en forma aleatoria.

El paso siguiente es determinar el tamaño de la próxima transacción a ser generada. Este tamaño estará dado por una distribución de Poisson con media μ igual al tamaño promedio de las transacciones. Cada ítem será elegido con probabilidad *p*. Si se tienen N ítems para tomar, la cantidad esperada de estos en una transacción, estará dada por una distribución Binomial con parámetros N y *p* y se aproxima a una distribución de Poisson con media N*p*.

Para modelar la situación en que los Itemsets Frecuentes tienen elementos en común, se toma una fracción del itemset actual que pasará a formar parte del próximo itemset. El resto se completará con ítems aleatorios. Para decidir qué fracción del itemset tomar para llevar a otro se tomará un valor aleatorio exponencialmente distribuido con media igual a un nivel de correlación 0.25.

En cada transacción se debe asignar una serie de potenciales Itemsets Frecuentes que son tomados de la base *T*. Si cuando se intenta incorporar un Itemset Frecuente a una transacción supera su tamaño, a veces se coloca de todas maneras aunque la mayoría de las veces se trasladará a la próxima transacción que se genere.

Para modelar la situación en que todos los ítems de un Itemset Frecuente no siempre se compran juntos se le asigna a cada itemset en *T* un nivel de corrupción *c* fijo obtenido con una distribución normal con media 0.5 y varianza 0.1. Cuando se agrega un itemset a una transacción se elimina un ítem siempre que un número aleatorio uniformemente distribuido que esté entre 0 y 1 sea menor que *c*. Por lo tanto, para un itemset de tamaño *l*, se agregan *l* ítems a la transacción $1 - c$ de las veces, $l - 1$ items $c(1 - c)$ de las veces, $l - 2$ ítems $c^2(1 - c)$ de las veces, y así siguiendo.

Con este mecanismo se logra simular el comportamiento de las transacciones y obtener una base con una cantidad previamente definida de potenciales Itemsets Frecuentes. Se garantiza que la muestra se comporta símil a la realidad y tiene los componentes necesarios para realizar la experimentación.

A continuación, se detalla la otra parte necesaria para lograr imitar la realidad, los momentos en los cuales se realizan las transacciones.

4.2.2 Generación de los Momentos en los que Fueron Realizadas las Transacciones

"Apúrate y espera..." es un dicho común en la vida cotidiana que resume lo que se debe considerar cuando se necesita simular los momentos de las transacciones. La civilización pasa grandes partes de su tiempo esperando en una cola: esperando el desayuno, detenido en un semáforo con luz roja, en un peaje, esperando un ascensor, esperando ser atendido en un call center, y así siguiendo. Existen innumerables situaciones de la vida cotidiana en las cuales se hacen colas.

En los comportamientos nombrados existe un fenómeno que se ha logrado modelar con la teoría denominada "de Colas", creada por el matemático danés A. K. Erlang por el año 1909. Esta teoría permite simular las líneas de espera que se producen cuando llegan clientes demandando un servicio, *esperando* si no se les puede atender inmediatamente y *partiendo* cuando ya han sido servidos.

La descripción del fenómeno mencionado es como sigue:

A lo largo del tiempo se producen llegadas de clientes que requieren un servicio a la cola de un sistema. Los clientes salen o son emitidos por una determinada fuente. Los servicios son satisfechos por entidades llamadas servidores del sistema.

Se denomina fuente al dispositivo del que emanan las unidades que solicitan un servicio. Si el número de unidades potenciales es finito, la fuente es finita; en caso contrario es infinita.

Los servidores del sistema seleccionan miembros de la cola según una regla predefinida denominada disciplina de la cola. Los criterios o disciplinas pueden ser: escoger al cliente que llegó antes (FIFO), o al cliente que llegó último (LIFO), o el que menos tiempo de servicio requiere, o el que más requiere o alguno creado específicamente para la situación.

Cuando un cliente seleccionado de la cola según un criterio, termina de recibir su servicio (tras un tiempo de servicio) abandona el sistema, pudiendo o no unirse de nuevo a la fuente de llegadas.

A veces, se sabe exactamente cuándo se van a producir las llegadas de los clientes al sistema, pero en general el tiempo que transcurre entre dos llegadas consecutivas se puede modelar mediante una variable aleatoria. En particular, cuando la fuente es infinita se supone que las unidades que van llegando al sistema dan lugar a un proceso estocástico llamado de conteo; si todos los tiempos entre llegadas son variables aleatorias

independientes idénticamente distribuidas, se dice que es un proceso de renovación. Usualmente el proceso que se utiliza es de Poisson con media λ.

Otro dato a tener en cuenta es la capacidad del servicio; es decir, el número de clientes que pueden ser atendidos simultáneamente. Si la capacidad es uno se dice que hay un solo servidor o que el sistema es monocanal, y si hay más de un servidor es multicanal.

El tiempo que el servidor necesita para atender la demanda de un cliente (tiempo de servicio) puede ser constante o aleatorio. Cuando los tiempos de servicio son aleatorios se pueden representar con variables aleatorias idénticamente distribuidas e independientes de los tiempos entre llegadas. La distribución exponencial es una buena aproximación a la distribución del servicio.

Un modelo simple a modo de ejemplo consta de un único servidor con una cola infinita de espera. Los eventos que ocurren son:

1. Llega un cliente a la cola.
2. Es atendido el primer cliente de la cola.

El modelo se centra, entonces, en saber la longitud de la cola en cualquier instante de tiempo. La llegada de un cliente incrementa la cola y la salida o atención la decrementa.

La información de tiempo a incorporar será un número aleatorio uniformemente distribuido sobre el intervalo [0,1].

Sea X la variable aleatoria y F(x) su función de distribución

$$F(x) = Pr\{X <= x\}$$

Sea F(X) = Y, Y definida en el intervalo [0,1]. Si Y es una variable aleatoria uniformemente distribuida entre 0 y 1, la variable X queda definida por

$$X = F^{-1}(Y)$$

Se tiene la distribución acumulativa F(X)

$$Pr\{X <= x\} = Pr\{Y <= F(x)\} = F(x)$$

Y es la variable aleatoria deseada construida con la función inversa F^{-1}.

El arribo de clientes a la cola de espera es una distribución de Poisson con media λ, los tiempos entre arribo de clientes es una distribución exponencial negativa con media 1/ λ.

$$F(x) = 1 - e^{-\lambda x}$$

Entonces, la variable aleatoria uniforme Y generada es igual a

$$Y = 1 - e^{-\lambda x}$$

Por lo tanto, la variable deseada es

$$X = F^{-1}(Y) = - (\ln (1 - Y)/ \lambda)$$

Con esta aproximación se logra modelar el sistema deseado de momentos de transacciones.

4.3 Diseño de los Experimentos

Los experimentos a realizar se idearon para lograr verificar los tiempos de ejecución, escalabilidad con respecto al número de transacciones y resultados encontrados de los algoritmos Apriori (implementado en WEKA), DIC y TDIC (desarrollados teniendo como base WEKA).

Los experimentos fueron ejecutados en dos equipos diferentes con las siguientes características:

Procesador	PC Clon procesador Intel Pentium III 850 Mhz	PC Clon procesador Intel Pentium III 350 Mhz
Memoria RAM	1.5 GB de memoria RAM	256 MB de memoria RAM
DISCO Rígido	Disco Rígido IDE de 7200 RPM de 30 GB	Disco Rígido IDE de 5400 RPM de 20 GB
Sistema Operativo	Windows NT 4.0	Windows 98

Tabla 4.1: Características de Equipos

Los archivos de datos sintéticos utilizados poseen las siguientes características:

Número de Potenciales Itemsets Frecuentes = 2000				
Cantidad de Ítems a utilizar = 1000				
Nombre del Archivo	Número de Transacciones	Tamaño Promedio de las Transacciones	Tamaño Promedio de los Potenciales Itemsets Frecuentes	Tamaño del Archivo Mbytes
T5I2D100K	100K	5	2	4.3
T10I2D100K	100K	10	2	7.5
T10I4D100K	100K	10	4	7.5
T20I2D100K	100K	20	2	11.7
T20I4D100K	100K	20	4	12.7

T20I6D100K	100K	20	6	13.2
T10I4D200K	200K	10	4	15
T10I4D300K	300K	10	4	22.6
T10I4D400K	400K	10	4	28.6
T10I4D500K	500K	10	4	38.6
T10I4D600K	600K	10	4	46.3
T10I4D700K	700K	10	4	55.2
Cantidad de Ítems a utilizar = 10				
T10I4D10KN10	10K	10	4	0.205

Tabla 4.1: Características de Archivos de Datos

Los momentos de las transacciones fueron generados por un servidor con un promedio de arribos de 35 personas por minuto y un promedio de salidas de 150 personas por minuto.

Los soportes fueron elegidos con el objetivo de ver el comportamiento de los algoritmos a medida de que éste cambie. Con una misma muestra de datos, si el soporte se decrementa la cantidad de información encontrada se incrementa. Ejemplo: con un muestra de 100000 transacciones, y un soporte de 2.0%, un elemento debería estar presente 2000 veces para ser considerado; con un soporte de 0.25% debería estar 250 veces. Los soportes seleccionados son: 2.00%, 1.50%, 1.00%, 0.75%, 0.50%, 0.33% y 0.25%. En la experimentación de cantidad de itemsets y reglas encontradas se utilizaron los soportes 0.30%, 0.25% y 0.15%.

El tamaño del intervalo, en el que se divide el archivo de transacciones (M), a tomar en los algoritmos DIC Y TDIC también fue elegido tratando de encontrar un valor óptimo. Se utilizaron 100, 1000 y 5000.

Los tamaños del intervalo de tiempo mínimo utilizado en el algoritmo TDIC considerados son 1.0 y 6021.5 segundos.

4.4 Resultados de la Experimentación

La experimentación, como se mencionó anteriormente, se enfocó básicamente en los tiempos de ejecución, escalabilidad y cantidad de itemsets y reglas encontrados en los algoritmos estudiados (Apriori, DIC y TDIC). El detalle es el que sigue:

4.4.1 Tiempos de Ejecución

Las muestras de las pruebas se dividieron en tres grandes grupos para su mejor visualización. El primer grupo (Figura 4.1) muestra los tiempos de ejecución que se obtuvieron al ejecutar el algoritmo Apriori implementado por WEKA. El segundo grupo (Figura 4.2) muestra los tiempos de ejecución que se obtuvieron al ejecutar el algoritmo DIC implementado teniendo como base la estructura de WEKA. Y el tercer grupo (Figura 4.3) muestra los tiempos de ejecución que se obtuvieron al ejecutar el algoritmo implementado

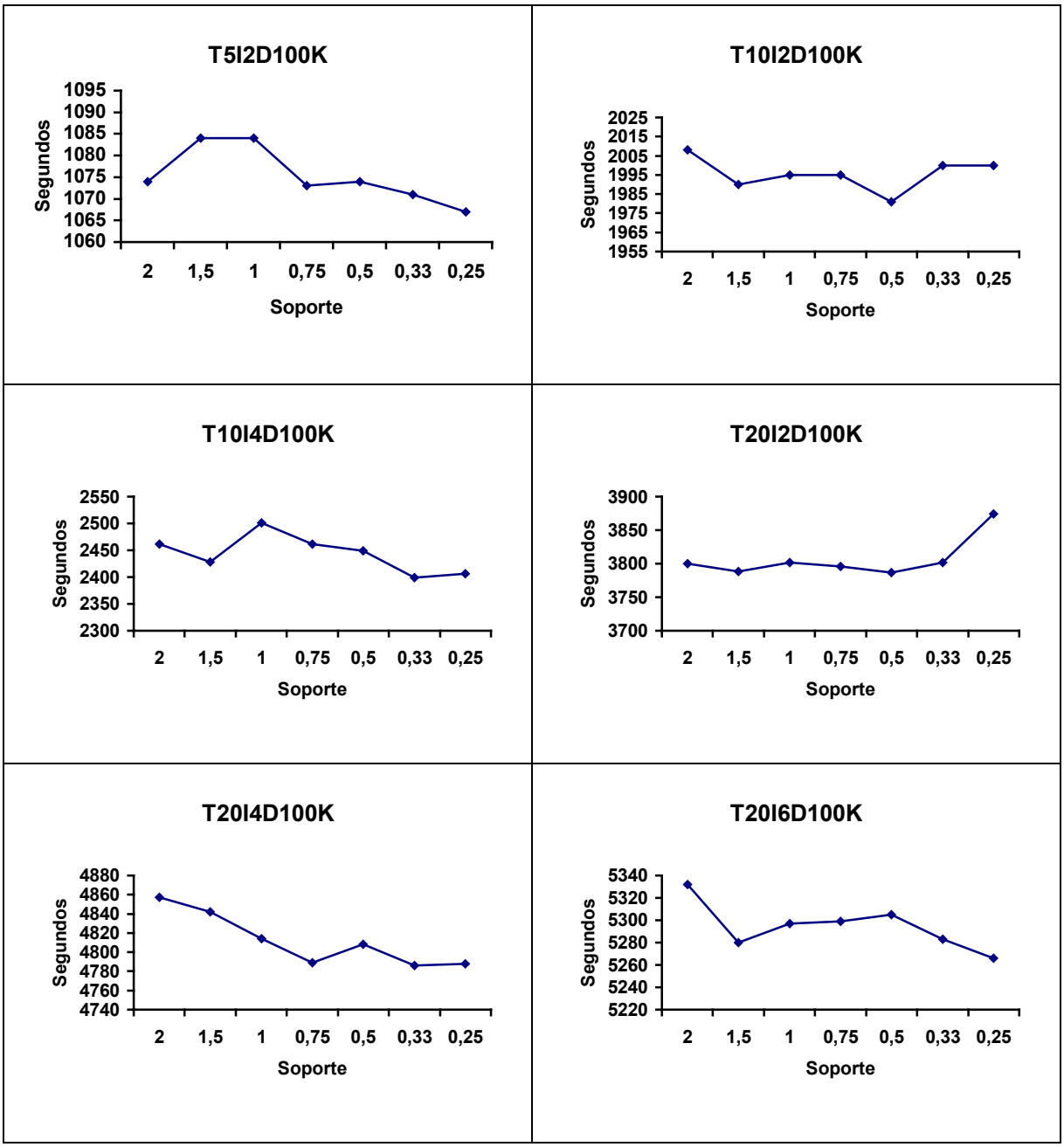


Figura 4.1: Tiempos de Ejecución Algoritmo Apriori

TDIC, teniendo como base la estructura de DIC. Las muestras de datos utilizadas son las mismas en los tres grupos y descriptas anteriormente.

Los algoritmos DIC y TDIC se ejecutaron con distintos valores de M, es decir con distintos tamaños de porciones para analizar el archivo de transacciones.

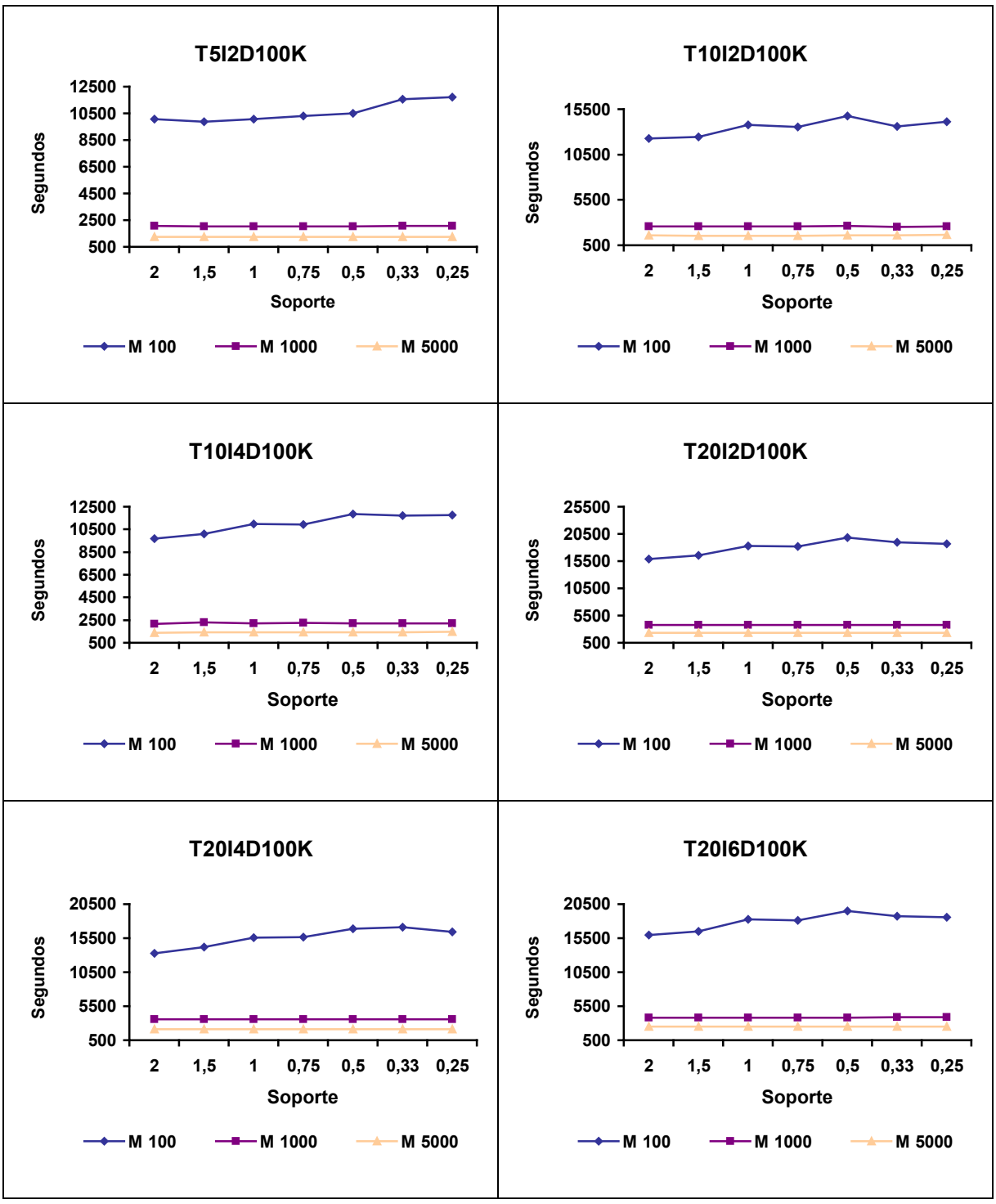


Figura 4.2: Tiempos de Ejecución Algoritmo DIC

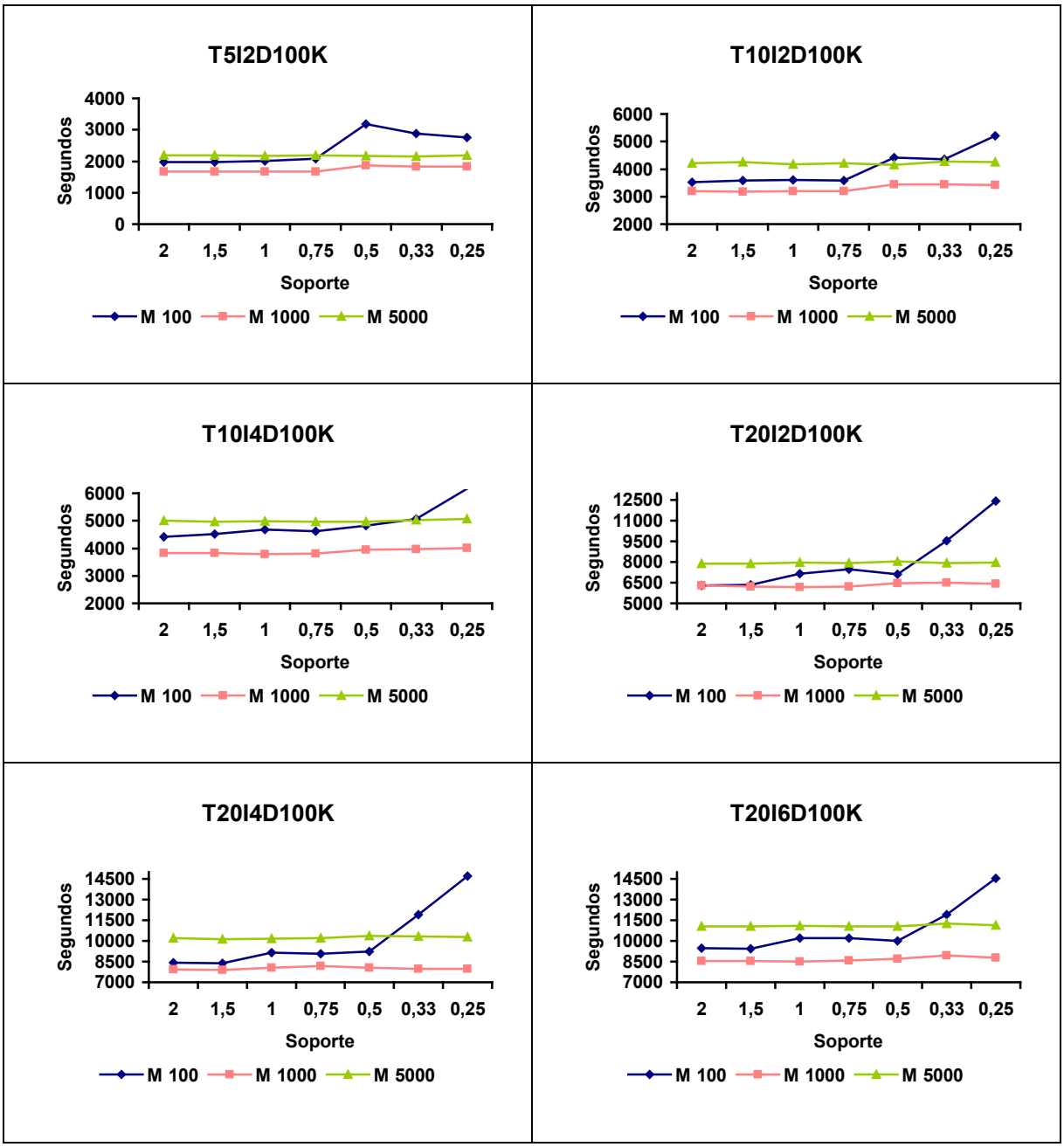
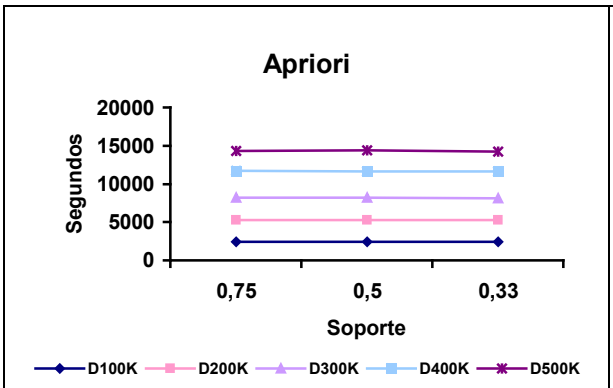
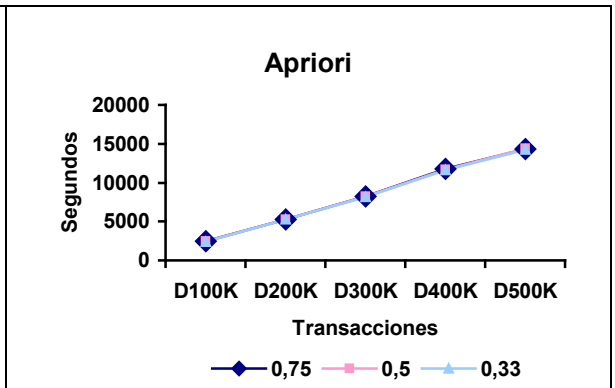


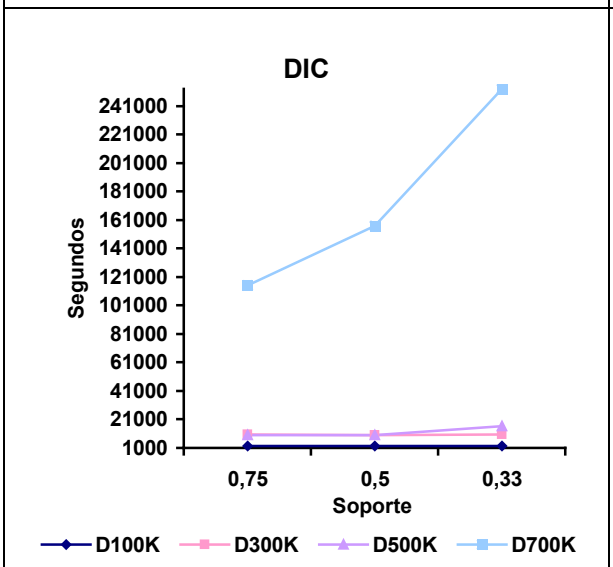
Figura 4.3: Tiempos de Ejecución Algoritmo TDIC

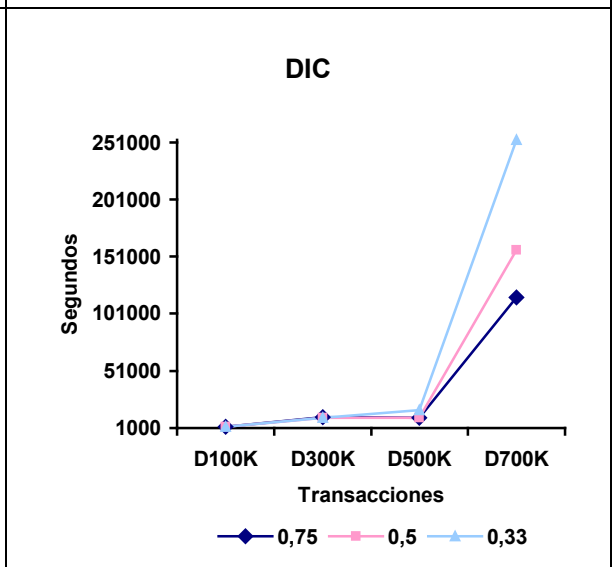
4.4.2 Escalabilidad

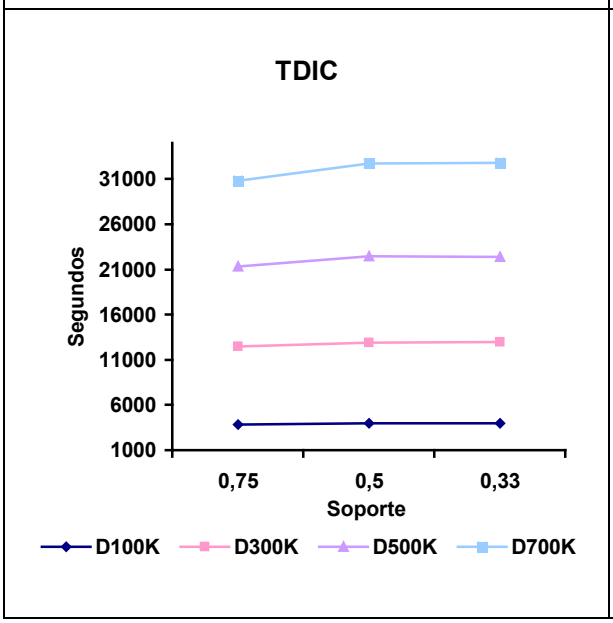
Las figuras 4.4 y 4.5, muestran distintas pruebas que se realizaron para ver el funcionamiento de los algoritmos con muestras de datos de distintos tamaños con respecto a cantidad de transacciones. La misma experimentación está representada de dos formas: por soporte y por cantidad de transacciones.











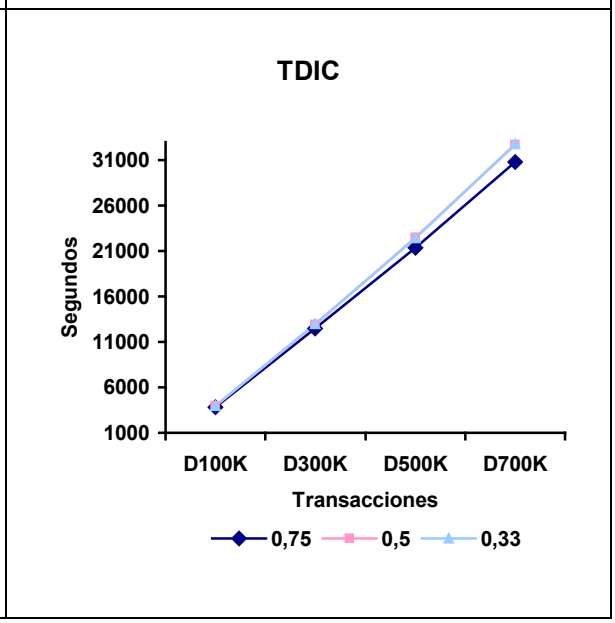


Figura 4.4: Escalabilidad de Algoritmos por Soporte

Figura 4.5: Escalabilidad de Alg. por Cant. de Transac.

Se utilizaron archivos del tipo T10I4 (descriptos en la tabla 4.2, tamaño promedio de las transacciones = 10 y tamaño promedio de los potenciales itemsets frecuentes = 4) con cantidad de transacciones entre 100K y 700K. Los soportes utilizados para las muestras fueron 0.75%, 0.50% y 0.33%. Para los algoritmos DIC y TDIC el parámetro M fue 1000.

Con el algoritmo Apriori no se pudo hacer la prueba de escalabilidad con una muestra de datos de 700K transacciones por una limitación física de almacenamiento de la estructura de datos que maneja WEKA.

4.4.3 Resultados Encontrados

Los experimentos de la Figura 4.6, muestra la cantidad de itemsets y reglas que cada algoritmo encuentra para una determinada muestra de datos.

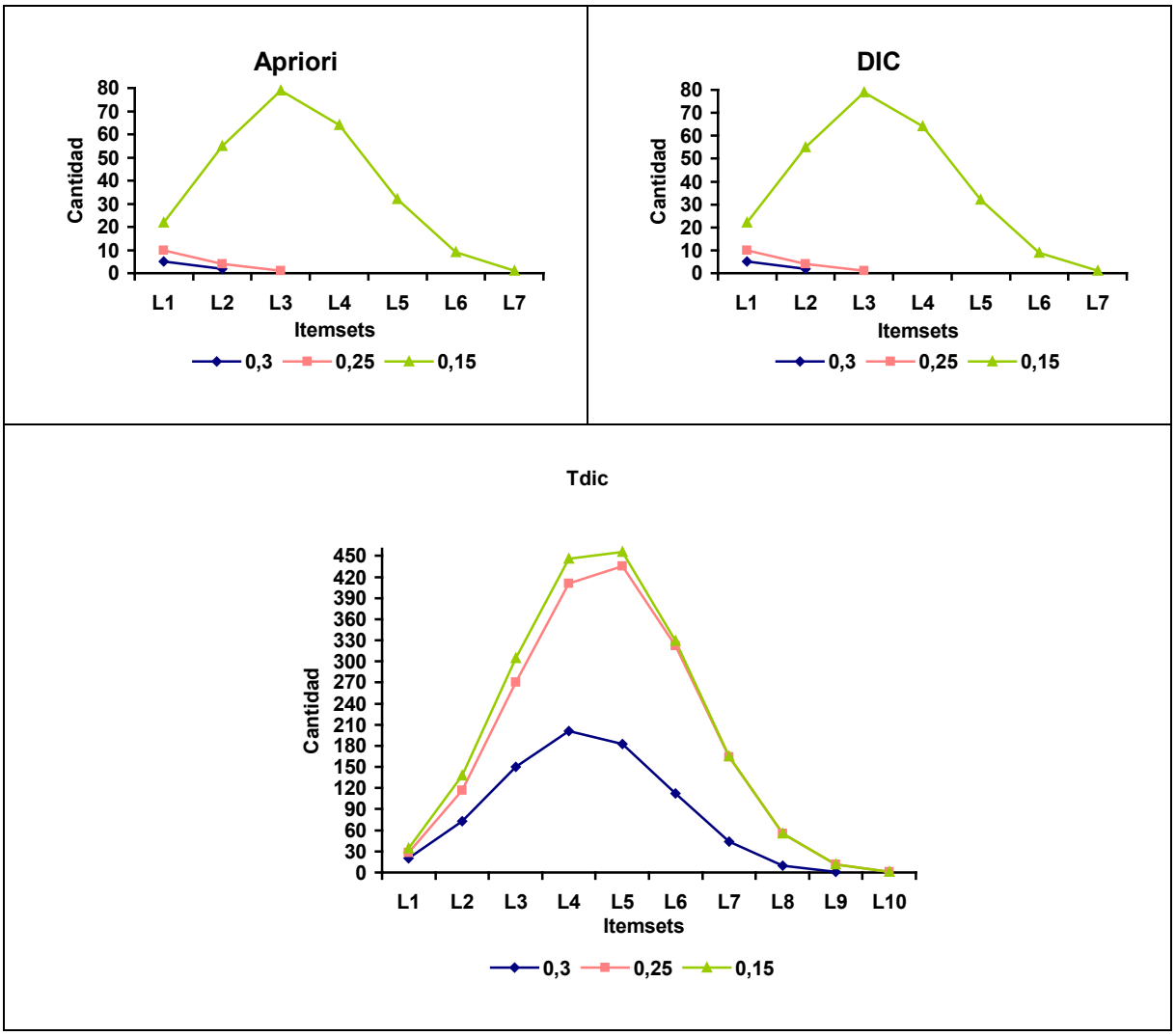


Figura 4.6: Cantidad de Itemset Encontrados

En este caso se utilizó el archivo T10I4D10KN10 (descripto en la tabla 4.2) con distintos soportes: 0.30%, 0.25% y 0.15%.

Para los algoritmos DIC y TDIC se probaron distintos M.

Los algoritmos Apriori y DIC obtuvieron los mismos itemsets y TDIC obtuvo mayor cantidad de información.

En la figura 4.7 se presenta el resultado de ejecutar TDIC con distintos parámetros de intervalo mínimo. Se puede notar que dependiendo del mínimo intervalo definido varía la cantidad de itemset encontrados.

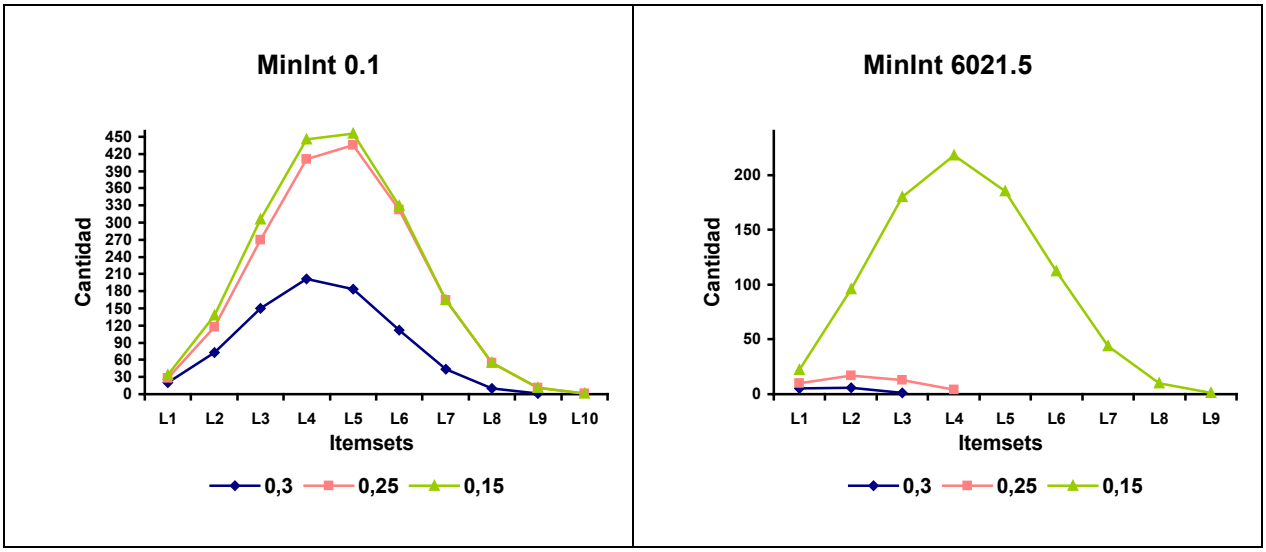


Figura 4.7: Cantidad de Itemset Encontrados en TDIC con Distintos Intervalos Mínimos

A continuación se presentan resultados con respecto a la cantidad de reglas encontradas. WEKA en su generador de reglas posee un parámetro que le indica la cantidad de reglas a buscar. En este caso, como se desea encontrar la cantidad máxima de reglas se informó este parámetro en un valor significativamente grande para así lograr encontrar todas las reglas que se pudieran formar.

Hay que notar que a mayor cantidad de itemset en los conjuntos L_i mayor cantidad de reglas se pueden lograr.

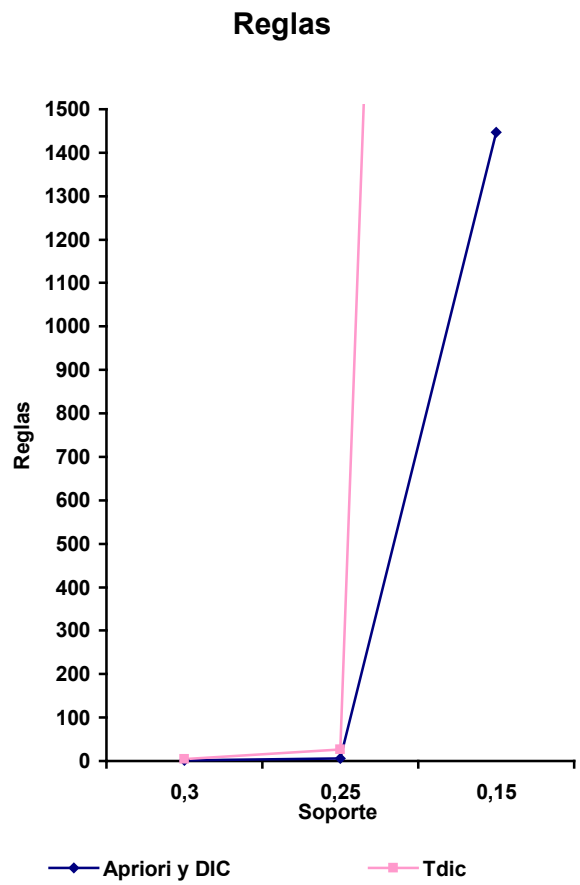


Tabla de Datos

- Apriori y DIC

Soporte	0.30	0.25	0.15
Cant. Reglas	2	6	1447

- TDIC

Soporte	0.30	0.25	0.15
Cant. Reglas	5	27	9907

Figura 4.8: Cantidad de Reglas

Capítulo 5

Análisis y Conclusiones de los Resultados Obtenidos

Analizando las ejecuciones y gráficos presentados en el capítulo anterior, podemos observar que los tres algoritmos se comportan de manera similar a medida que se decrementa el soporte, los tiempos de ejecución se mantienen, aunque en TDIC se observa un pequeño incremento, dada la mayor cantidad de itemsets generados.

Todos los algoritmos incrementan linealmente los tiempos de ejecución al aumentar la complejidad de los datos ingresados. O sea, al incrementar los parámetros T (tamaño promedio de transacciones) e I (tamaño promedio de los itemset potencialmente frecuentes), se incrementan en igual proporción los tiempos de ejecución.

Como mencionamos anteriormente, al haber implementado DIC desde una versión existente de Apriori, no es performante nuestra versión. Esto sucede porque el algoritmo Apriori provisto por WEKA, lee físicamente sólo una vez el archivo de entrada, y lo recorre N veces en memoria. La diferencia con los algoritmos DIC y TDIC es que deben leer físicamente el archivo por partes y mantener los itemsets candidatos en memoria para cada lectura, lo que implica mayor tiempo de procesamiento.

Esta baja de performance puede observarse en DIC y TDIC al incrementar el parámetro M en ejecución. A medida que se incrementa, los tiempos de ejecución decrecen. Recordemos que, desde el último itemset frecuente generado, se debe realizar una lectura completa más del archivo. Y en cada lectura, procesar todos los L anteriores.

Para aclarar lo expuesto, tomemos en cuenta que el archivo se divide en

C = Cantidad de transacciones del archivo / M.

Entonces, DIC y TDIC deben realizar al menos C accesos a disco para leer la información. Además, cada L(*k*) tiene que estar en memoria y ser procesado al menos C veces. Esto es porque cada elemento de cada L(*k*) debe ser evaluado y no puede ser descartado hasta que hayamos dado una vuelta completa al archivo desde que se empezó a contar.

La decisión de realizar la lectura física por partes fue porque la versión de Apriori no es escalable a grandes volúmenes de datos. Inicialmente, se habían implementado DIC y TDIC cargando en memoria el archivo y llamando a la generación de itemsets con partes de esta estructura en memoria, pero al igual que sucede con Apriori no podíamos ejecutarlo con archivos de más de 500.000 transacciones.

Por lo expuesto podemos decir que el algoritmo Apriori con la implementación provista no es escalable, mientras que DIC y TDIC sí lo son. Respecto de los tiempos de ejecución, en TDIC se incrementan linealmente a medida que se incremente la cantidad de transacciones a procesar. En DIC, se incrementan exponencialmente.

Respecto de cantidad de itemsets frecuentes y reglas de asociación encontradas, Apriori y DIC generan la misma cantidad en cada uno de los soportes testeados. TDIC, dependiendo del mínimo intervalo que se defina, encuentra mayor cantidad de asociaciones. Esto es lo esperado, dado que el soporte de un itemset se calcula entre la primer y última transacción en las que aparece, y no en toda la extensión del archivo.

A medida que se expande el mínimo intervalo, decrece la cantidad de itemsets y reglas encontrados. Si se eligiera el intervalo de tiempo definido entre la primera y la última transacción los resultados serían idénticos a los retornados por el algoritmo DIC.

Aunque en este trabajo no se presentan, algunas de las ejecuciones se realizaron en las dos máquinas diferentes mencionadas en 4.3, y los resultados muestran que los tiempos de ejecución dependen del procesador, y no de la cantidad de memoria, ni capacidad del disco rígido. En promedio, las ejecuciones tardaron un 30% en la máquina de procesador menos potente.

Capítulo 6

Conclusión

En este trabajo, hemos presentado información sobre los principales aspectos de Data Mining, hemos descripto los principales algoritmos conocidos para la búsqueda de información en un mundo lleno de datos, pero en el cual es muy difícil encontrar información.

También presentamos diversos enfoques actuales respecto de la búsqueda anteriormente mencionada. Nos centramos en uno de estos enfoques, la temporalidad de la información, y estudiamos su aplicación.

A partir del algoritmo Apriori implementado por la Universidad de Waikato de Nueva Zelanda, implementamos el algoritmo Dynamic Itemset Counting, probando que reduce la cantidad de pasadas sobre el archivo, con la consecuente reducción de tiempos de ejecución, y mejora de performance.

Posteriormente, incorporamos información temporal a este algoritmo, implementando el algoritmo Temporal Dynamic Itemset Counting.

Generamos datos sintéticos para probar las tres aplicaciones, ejecutamos las mismas y estudiamos los resultados obtenidos, desde el punto de performance y desde la cantidad de información obtenida.

Pudimos demostrar que al incorporar temporalidad, se incrementa la cantidad de información encontrada (itemsets y reglas de asociación), nos permite obtener resultados acotados a determinados intervalos de tiempo, que con Apriori y DIC no se encuentran.

Como extensión a este trabajo, podemos sugerir realizar nuevas implementaciones de los algoritmos para que sean escalables a grandes volúmenes de datos, manteniendo la performance de Apriori. Las mejoras deben enfocarse a resolver las restricciones de memoria generadas por Java.

También podrían extenderse las aplicaciones mencionadas en [2], e incorporar el factor de obsolescencia de los itemsets, para descartar aquellos que sean anteriores a un determinado parámetro definido por el usuario.

Además, consideramos que sería interesante realizar pruebas sobre los algoritmos utilizando datos reales en lugar de sintéticos. En este trabajo se simularon con algoritmos estudiados para reflejar la realidad.

Apéndice A

WEKA, DIC y TDIC

A.1 Introducción a WEKA

La sigla "WEKA" significa Waikato Environment for Knowledge Analysis y es un sistema desarrollado en la Universidad de Waikato en Nueva Zelanda. Principalmente, provee implementaciones de algoritmos de aprendizaje que pueden ejecutarse desde la línea de comandos. También incluye varias herramientas para transformar datos, como los algoritmos de discretización. Se puede pre-procesar un conjunto de datos, procesarlo con algún plan de aprendizaje, analizar el clasificador resultante y su performance sin escribir una sola línea de código.

WEKA puede utilizarse en diferentes niveles. Si bien los algoritmos de aprendizaje son el objetivo principal y las herramientas para pre-procesar los datos, llamados *filters*, son el segundo; incluye importantes implementaciones de algoritmos para encontrar reglas de asociación y clustering.

Una forma de utilizar WEKA es aplicar un método de aprendizaje a un conjunto de datos y analizar su salida para obtener información sobre los datos. Otra, es aplicar diferentes métodos de aprendizaje y comparar la performance con el fin de elegir uno para realizar predicciones. Los métodos de aprendizaje se encuentran en *classifiers*.

WEKA Provee una interfaz uniforme a distintas plataformas para diferentes algoritmos de aprendizaje, junto con los métodos para pre- y post- procesamiento y para la evaluación de los resultados de los esquemas de aprendizaje sobre cualquier base de datos. Esto se logra pues el sistema está escrito en Java, un lenguaje de programación orientado a objetos que está disponible para la mayoría de las plataformas. WEKA fue testeado bajo los sistemas operativos Linux, Windows, y Macintosh.

Todo el sistema está implementado y disponible en la Web (www.cs.waikato.ac.nz/ml/weka) para ser utilizado. El recurso más importante para navegar a través del sistema es la documentación OnLine, que ha sido generada automáticamente desde el código fuente y refleja la estructura de forma clara y concisa.

A.2 Organización de WEKA

WEKA está organizado en packages que corresponden a una jerarquía de directorios. Un package es simplemente un directorio que contiene una colección de clases relacionadas.

Cada programa Java se implementa como una clase (*class*). En programación Orientada a Objetos, una clase es una colección de variables con *métodos* que operan sobre esas variables. Juntos, definen el comportamiento de un *objeto* que pertenece a la clase. Un objeto es simplemente una instanciación de la clase que tiene valores asignados para todas las variables de la clase.

WEKA también provee varias interfaces. Una interface es muy similar a una clase, la única diferencia es que la interface no hace nada por sí misma, es simplemente una lista de métodos sin su implementación. Otras clases pueden declarar que "implementan" una interface particular, y proveer el código para los métodos.

La implementación de cada algoritmo de aprendizaje se representa por una clase. Los programas más largos se dividen en más de una. Cuando hay muchas clases (como en WEKA) puede ser difícil comprender y navegar la estructura.

A.2.1 Packages

El Package weka.core

El package *core* es el principal del sistema WEKA. Contiene clases que son accedidas desde la mayoría de las otras clases.

Las clases principales en este package son Attribute, Instance e Instances. Un objeto de la clase Attribute representa un atributo. Contiene el nombre del atributo, su tipo y en el caso de los atributos nominales sus valores posibles. Un objeto de la clase Instance contiene los valores de los atributos de una instancia particular. Un objeto de la clase Instances contiene un conjunto ordenado de instancias.

- **Weka.core.Instance**

Es la clase que implementa una instancia. Todos los valores se almacenan internamente como números floating-points. En el caso de los atributos nominales el valor almacenado es el índice del valor nominal correspondiente en la definición del atributo.

Todos los métodos que cambian una instancia son seguros, o sea que un cambio en una instancia no afecta ninguna otra instancia. Todos los métodos que cambian el valor de un atributo de la instancia clonan el vector de valores del atributo antes de cambiarlo.

- **Weka.core.Instances**

Es la clase para manejar un conjunto ordenado de instancias pesadas.

Todos los métodos que cambian un conjunto de instancias son seguros, o sea que un cambio en un conjunto de instancias no afecta ningún otro conjunto de instancias. Todos los

métodos que cambian la información del atributo del conjunto de datos, clonan el mismo antes de cambiarlo.

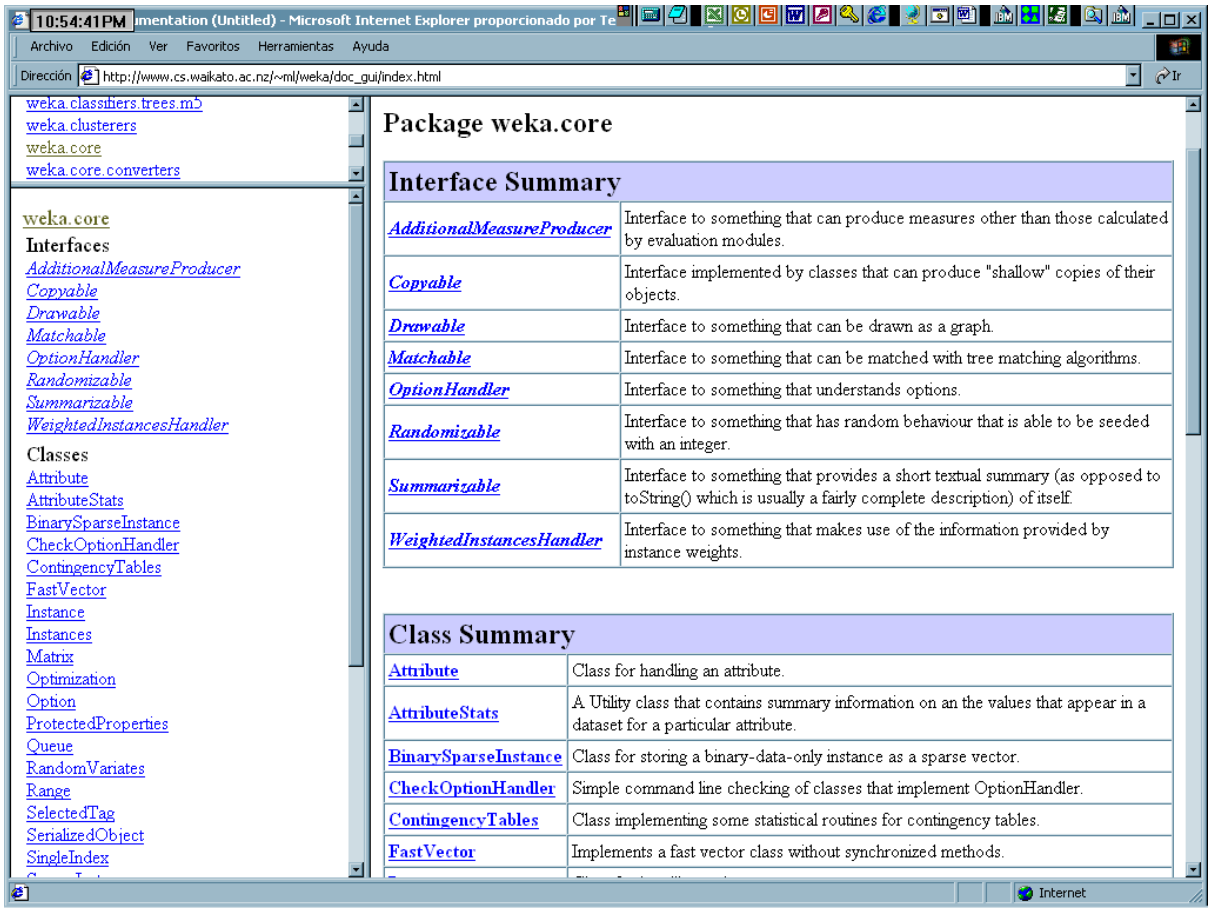


Figura A.1: Package weka.core

El Package weka.classifiers

El package *classifiers* contiene la implementación de la mayoría de los algoritmos para clasificación y predicción numérica. La predicción numérica se incluye en *classifiers* porque se interpreta como predicción de una clase continua. La clase más importante es *Classifier*, que define la estructura general de cualquier escenario de clasificación y predicción numérica.

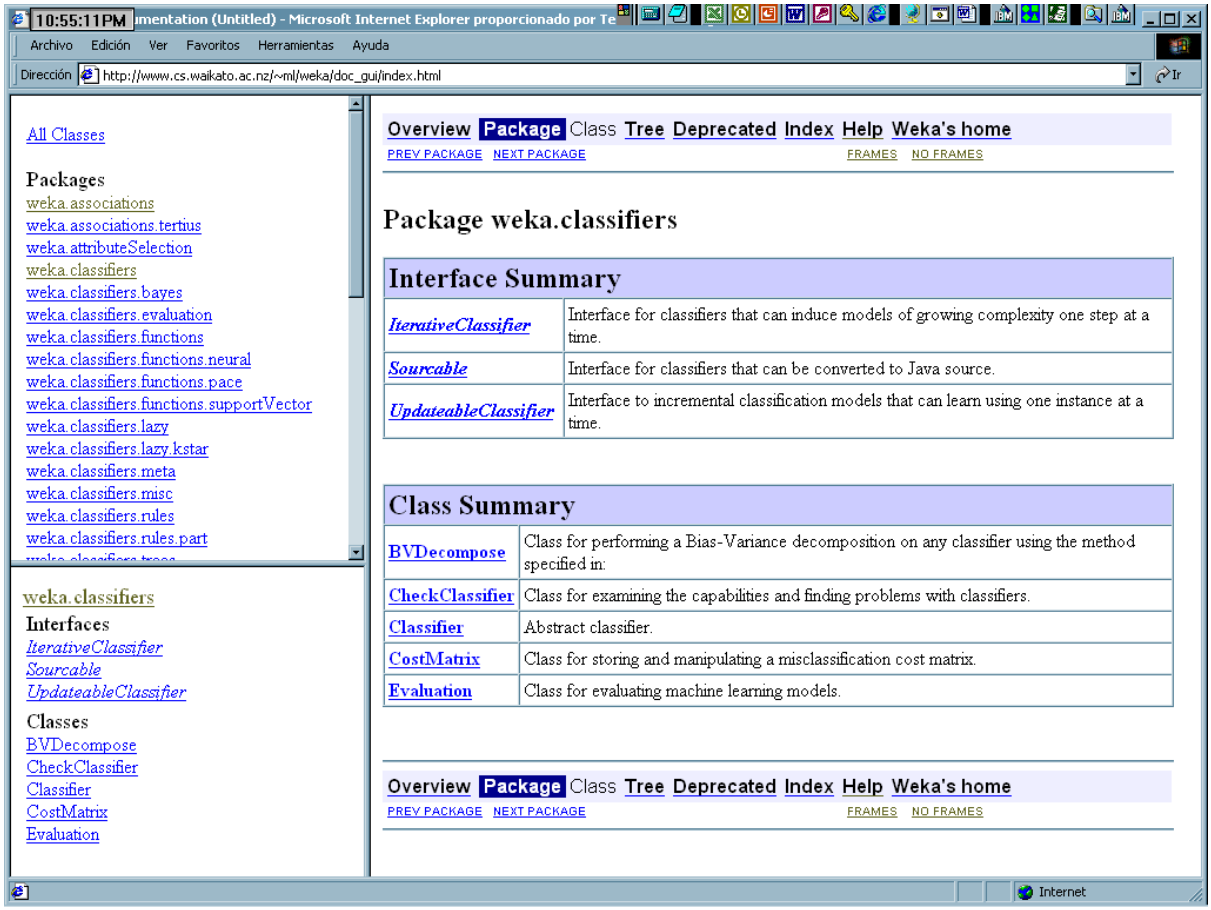


Figura A.2: Package weka.classifiers

El Package weka.associations

El package *associations* contiene dos clases importantes Apriori e ItemSet, que implementan el algoritmo Apriori para el descubrimiento de reglas de asociación.

- **Weka.associations.Apriori**

Es la clase que implementa un algoritmo del tipo Apriori. Iterativamente reduce el mínimo soporte hasta que encuentra el número pedido de reglas con la mínima confidencia solicitada.

- **Weka.associations.ItemSet**

Es la clase para almacenar un conjunto de ítems en orden lexicográfico, determinado por la información cabecera del conjunto de instancias utilizado para generar el conjunto de ítems. Todos los métodos de la clase asumen que los ítems están en el mencionado orden.

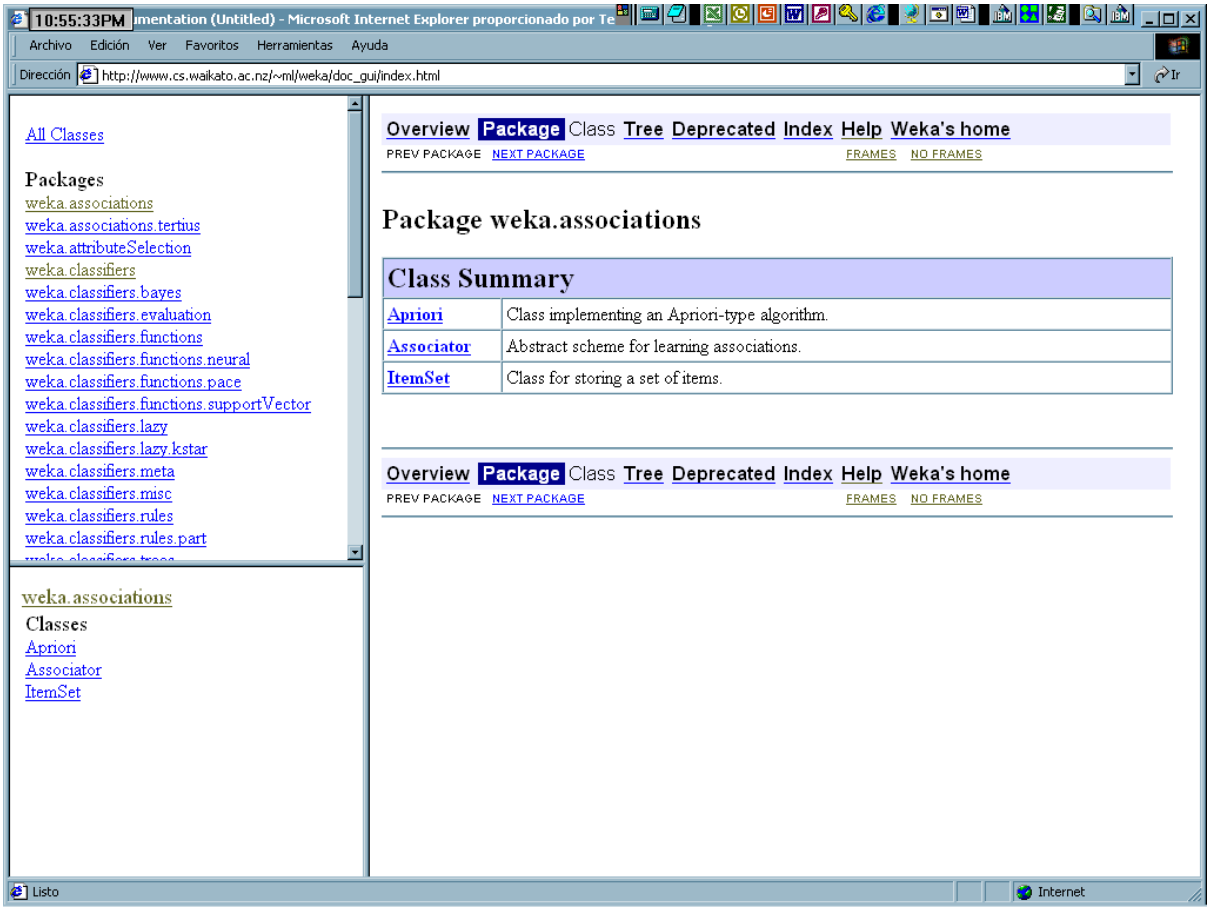


Figura A.3: Package weka.associations

Otros Packages

Se pueden nombrar otros packages que se encuentran en WEKA: *weka.classifiers.j48, weka.classifiers.m5, weka.clusterers, weka.estimators, weka.filters,* y *weka.attributeSelection*. Estos poseen distintos tipos de algoritmos de aprendizaje y pre-procesamiento de datos.

A.3 Entrada de Datos

Los datos de entrada a WEKA deben tener un formato establecido por la aplicación y es formato ARFF. Las aplicaciones necesitan tener información del tipo de cada atributo que no puede ser deducido desde los valores del atributo. Antes de aplicar cualquier algoritmo, los datos deben ser convertidos a formato ARFF que es una lista de todas las instancias con los valores de los atributos separados por comas.

A.3.1 Descripción de Archivos ARFF (Attribute-Relation File Format)

Los archivos del tipo ARFF fueron desarrollados por el Proyecto de Machine Learning de la Universidad de Waikato para ser usados en WEKA. Es un archivo de texto ASCII que describe una lista de instancias que comparten un conjunto de atributos. Están compuestos por dos secciones diferentes, la primera parte es la información de *Header* (cabecera) y la segunda es la sección *Data* (datos).

A.3.1.1 Sección Header

El Header de un archivo ARFF contiene el nombre de la relación, una lista de atributos (las columnas de los datos), y sus tipos. Un header ejemplo de un conjunto de datos se puede observar en la tabla A.1

1. Titulo: archivo de prueba 2. Corresponde a las transacciones: * 1 3 4 5 * 1 2 3 4 5 * 1 2 3 5 * 2 5 * 1 3 5	@relation transaction @attribute A {1,2} @attribute B {2,3,5} @attribute C {3,4,5} @attribute D {4,5} @attribute E {5}

Tabla A.1: Ejemplo de Header

Declaración @relation

El nombre de la relación se define en la primera línea de un archivo ARFF. El formato es:

@relation <nombre_de_la_relación>

donde <nombre_de_la_relación> es un string que debe estar entre comillas si el nombre contiene espacios.

Declaraciones @attribute

Las declaraciones de atributos tienen la forma de una secuencia ordenada de sentencias @attribute. Cada atributo debe tener su propia sentencia @attribute que define unívocamente el nombre del atributo y su tipo de datos. El orden en que los atributos se definen indica la posición de la columna en la sección de datos del archivo. Por ejemplo, si un atributo es el tercero declarado, WEKA espera que todos los valores de ese atributo se encuentren en la tercera columna delimitada por comas.

El formato de @attribute es el siguiente:

@attribute <nombre_de_atributo> <tipo_de_dato>

Donde <nombre_de_atributo> debe comenzar con un carácter alfabético. El <tipo_de_dato> puede ser cualquiera de los cuatro tipos de datos soportados por WEKA. Los tipos de datos son:

- Numérico
- <especificación-nominal>: es el único tipo de dato que puede utilizarse con el algoritmo Apriori.

  Los valores nominales se definen como: <especificación-nominal> seguida de la lista de los posibles valores:

  {<nombre-nominal1>, <nombre-nominal2>, <nombre-nominal3>, ...}
- string
- fecha [<formato-fecha>]

Las declaraciones @RELATION, @ATTRIBUTE respetan mayúsculas y minúsculas.

A.3.1.2 Sección Data

La sección Data de un archivo ARFF contiene la línea de declaración y las líneas de instancias.

Declaración @data

La declaración @data es una sola línea que marca el comienzo del segmento de datos del archivo. El formato es:

@data

La declaración @data respeta las mayúsculas y minúsculas.

Instancias de los datos

Cada instancia se representa en una línea, con carriage return al final de cada línea. Los valores de los atributos para cada instancia están delimitados por comas. Deben aparecer en el orden que fueron declarados en la sección Header (o sea, el dato correspondiente a la declaración del n-ésimo @attribute siempre es el n-ésimo campo del atributo). Los valores faltantes se representan por un signo de interrogación.

La sección Data se ve de la siguiente manera:

```
@data
1,3,4,5,?
1,2,3,4,5
1,2,3,5,?
2,5,?,?,?
1,3,5,?,?
```

En resumen, el archivo de datos de las transacciones detalladas en la Tabla A.1 queda como sigue en formato ARFF.

1. Titulo: archivo de prueba 2. Corresponde a las transacciones: 1 3 4 5 1 2 3 4 5 1 2 3 5 2 5 1 3 5	@relation prueba @attribute A {1,2} @attribute B {2,3,5} @attribute C {3,4,5} @attribute D {4,5} @attribute E {5} @data 1,3,4,5,? 1,2,3,4,5 1,2,3,5,? 2,5,?,?,? 1,3,5,?,?

Tabla A.2: Ejemplo de archivo ARFF

A.4 Aplicación Apriori

Para realizar una ejecución del algoritmo Apriori a través de la línea de comandos, se debe ingresar:

java weka.associations.Apriori -I -t C:\weka\prueba.arff

La salida del archivo presentado en Tabla A.2 es la siguiente:

```
Apriori
=======
Minimum support: 0.5
Minimum metric <confidence>: 0.9
Number of cycles performed: 10

Generated sets of large itemsets:

Size of set of large itemsets L(1): 5

Large Itemsets L(1):
A=1 4
B=2 2
B=3 2
C=3 2
D=5 2

Size of set of large itemsets L(2): 5

Large Itemsets L(2):
A=1 B=2 2
A=1 B=3 2
A=1 C=3 2
A=1 D=5 2
B=2 C=3 2

Size of set of large itemsets L(3): 1
```

```
Large Itemsets L(3):
A=1 B=2 C=3 2

Best rules found:

 1. B=2 2 ==> A=1 C=3 2    conf:(1)
 2. C=3 2 ==> A=1 B=2 2    conf:(1)
 3. A=1 B=2 2 ==> C=3 2    conf:(1)
 4. A=1 C=3 2 ==> B=2 2    conf:(1)
 5. B=2 C=3 2 ==> A=1 2    conf:(1)
 6. B=2 2 ==> C=3 2    conf:(1)
 7. C=3 2 ==> B=2 2    conf:(1)
 8. D=5 2 ==> A=1 2    conf:(1)
 9. C=3 2 ==> A=1 2    conf:(1)
10. B=3 2 ==> A=1 2    conf:(1)
```

Tabla A.3: Salida de Apriori con archivo de datos prueba.arff

La última parte (Best rules found:) muestra las reglas de asociación encontradas. El número que precede al símbolo ==> indica el soporte de la regla. Después de la regla está el número de ítems para los cuales vale el consecuente de la premisa. Entre paréntesis está la confidencia de la regla. En este ejemplo, la confidencia es 1 para todas las reglas. Apriori ordena las reglas de acuerdo a su confidencia. Antes de las reglas se muestran los itemsets frecuentes encontrados para cada soporte considerado. En este caso, el itemset frecuente encontrado con mínimo soporte contiene tres ítems.

Apriori utiliza parámetros en la búsqueda de itemsets frecuentes y reglas de asociación que tienen valores por defecto pero pueden ser cambiados por el usuario. Los principales son:

-t <training file>	Nombre del archivo de datos.
-N <required number of rules output>	Número de reglas buscadas (por defecto = 10)
-T <0=confidence | 1=lift | 2=leverage | 3=Conviction>	Métrica por la cual presentar las reglas (por defecto = 0)
-C <minimum metric score of a rule>	Confidencia mínima de una regla (por defecto = 0.9)
-D <delta for minimum support>	Delta por el cual se decrementa el soporte mínimo en cada iteración (por defecto = 0.05)
-U <upper bound for minimum support>	Límite superior para el soporte mínimo (por defecto = 1.0)
-M <lower bound for minimum support>	Límite inferior para el soporte mínimo (por defecto = 0.1)
-S <significance level>	Si se especifica, se verifica el significado de las reglas en un nivel dado (por defecto = no). Es más lento.
-I	Si se muestran los ItemSets encontrados (por defecto = no).
-R	Eliminar las columnas con valores faltantes (por defecto = no).
-V	Reporte de ejecución iterativamente (por defecto = no).

Tabla A.4: Parámetros requeridos por Apriori

La ejecución de Apriori trata de generar una determinada cantidad (N) de reglas. Comienza con un soporte mínimo (U) y lo decrementa de acuerdo a lo indicado en D hasta que haya al

menos N reglas con la confidencia mínima buscada, o hasta que el soporte haya alcanzado el límite inferior de M%, lo que ocurra primero. La confidencia mínima se parametriza en 0.9 por defecto. Se puede observar este comportamiento en la salida de la Tabla A.3, el soporte mínimo comienza en 1.0 o 10% y fue decrementado a 0.5 o 50%, antes que el total de reglas pudiera ser generado. Este proceso implicó un total de 10 iteraciones sobre los datos.

A.4.1 Implementación Apriori

Como dijimos anteriormente, las clases principales de WEKA para encontrar reglas de asociación son weka.associations.Apriori y weka.associations.ItemSet. Esta última clase implementa el método para encontrar los itemsets frecuentes.

Para la generación de ItemSets, el algoritmo Apriori carga en memoria (weka.core.Instances) todas las instancias del archivo de entrada (archivo en formato ARFF con datos nominales).

Con esta estructura de información en memoria, ejecuta el método *findLargeItemSets(Instances instances)* descripto a continuación:

Private void findLargeItemSets(Instances instances) throws Exception {	
FastVector kMinusOneSets, kSets;	
Hashtable hashtable;	
Int necSupport, i = 0;	
M_instances = instances;	
// Find large itemsets	
NecSupport = (int)(m_minSupport * (double)instances.numInstances()+0.5);	Calcula la cantidad de itemsets necesarios para el soporte
KSets = ItemSet.singletons(instances);	Genera todos los posibles 1-itemsets
ItemSet.upDateCounters(kSets, instances);	Actualiza contadores
KSets = ItemSet.deleteItemSets(kSets, necSupport);	Borra los 1-itemsets que no superan el soporte mínimo
If (kSets.size() == 0)	
return;	
Do {	
m_Ls.addElement(kSets);	Guarda los (k-1)-itemsets, dado que son frecuentes.
kMinusOneSets = kSets;	
kSets = ItemSet.mergeAllItemSets(kMinusOneSets, i);	Genera todos k-itemsets, con la información de los (k-1)-itemsets
hashtable = ItemSet.getHashtable(kMinusOneSets, kMinusOneSets.size());	Crea una tabla de hash, con los (k-1)-itemsets
m_hashtables.addElement(hashtable);	
KSets = ItemSet.pruneItemSets(kSets, hashtable);	Elimina los k-itemsets que no tengan subconjuntos en (k-1)-itemsets

`ItemSet.upDateCounters(kSets, instances);`	Actualiza los contadores de los k-itemsets
`KSets = ItemSet.deleteItemSets(kSets, necSupport);`	Borra los k-itemsets que no superan el soporte mínimo
`i++;`	
`} while (kSets.size() > 0);`	Mientras genere algún k-itemset
`}`	

Tabla A.5: Método findLargeItemSets

Con los itemsets obtenidos pasa a verificar las reglas de asociación que pueden generarse, y su confidencia. Si no encuentra el número solicitado de reglas con la confidencia pedida, repite el proceso de generación de itemsets, decrementando el soporte.

A.5 Modificaciones Introducidas a WEKA

Para obtener los algoritmos TDIC y DIC fue necesario modificar las clases weka.associations.Apriori y weka.associations.ItemSet. Por lo tanto, se tomó la decisión de replicar las clases necesarias e incorporar las modificaciones. Con esto se logró independencia y claridad.

La clase weka.associations.Apriori se replicó como weka.associations.Dic, y se modificaron los métodos necesarios para el nuevo mecanismo del algoritmo. En la clase weka.associations.ItemSet se incorporaron métodos similares a los existentes en Apriori, con las incorporaciones de los parámetros necesarios.

Una vez implementado el algoritmo DIC, se inició la implementación de TDIC dado que comparten la forma de procesamiento de los datos. Se replicó la clase weka.associations.Dic como weka.associations.TDic y weka.associations.ItemSet como weka.associations.ItemSetTDic. A estas últimas se les realizaron la incorporación de las funcionalidades e información temporal necesaria para el nuevo algoritmo.

Modificaciones introducidas para DIC.

Las modificaciones puntuales que se debieron incorporar son:

- Modificación en la forma de leer el archivo de entrada. En lugar de leer y procesar todas las instancias del archivo, se leen de a M transacciones (nuevo parámetro introducido por el usuario).
- Cada M transacciones leídas se llama a generar los itemsets frecuentes.

- Incorporación de nuevos candidatos encontrados en la porción de transacciones que se está evaluando en ese instante a cada conjunto de Itemsets frecuentes L_i. Generación de nuevo conjunto de Itemsets frecuentes L_{i+1} con el nuevo conjunto generado L_i.
- Incorporación de información que indique en qué parte de la muestra de datos se generó cada itemset. Las modificaciones son en los métodos de generación, actualización y borrado de itemsets para respetar y procesar las variables incorporadas.

Modificaciones introducidas para TDIC.

Las modificaciones puntuales que se debieron incorporar son:

- Incorporación de nueva variable para indicar el mínimo intervalo de tiempo a tener en cuenta para considerar frecuente un itemset en un lapso de tiempo.
- Incorporación de variables para contener el momento en que un itemset aparece por primera vez (m_t1), el momento en que aparece por última vez (m_t2), y los contadores temporales (m_Ftr y m_Ftr_prima).
- Incorporación de información del momento de realización de la transacción.
- Adaptaciones en los métodos de creación, modificación y borrado de itemsets, considerando las variables incorporadas en los puntos anteriores.

A.5.1 Detalle de las Modificaciones Introducidas para DIC

A continuación se detallan las modificaciones nombradas anteriormente para la implementación del algoritmo DIC.

- Lecturas del archivo de información de a M transacciones. Se incorpora en Instances la variable mi_tokenizer para guardar el puntero en el archivo y leer de a M.

  Para poder realizar esta modificación, en lugar de recibir las Instancias, el método de DIC *buildAssociations(String, int, int)* recibe el nombre del archivo a leer, la cantidad M de transacciones a leer y el tamaño del archivo (cantidad total de transacciones).

  La declaración:

  ```
  Public void buildAssociations(Instances instances) throws Exception {
  ```

  se reemplaza por:

  ```
  Public void buildAssociations(String trainFileString,int M,int
  cantInstances) throws Exception
  ```

  La llamada de generación de itemsets:

```
findLargeItemSets(instances);
```

se reemplaza por el siguiente código:

```
// Find large itemsets and rules

division =  cantInstances / M;
if ((cantInstances % M) != 0) division++;
int cantLeidos = M;
desde = 0;
reader = new BufferedReader(new FileReader(trainFileString));

instances = new Instances(reader,cantLeidos);

do {
       if (desde >= cantInstances){
              desde = 0;
              reader = new BufferedReader(new FileReader(trainFileString));
              instances = new Instances(reader,cantLeidos);
       }
       if ((desde+M) < (cantInstances))
          cantLeidos = M;
       else
          cantLeidos = cantInstances - desde;
       instances.proximaInstances(cantLeidos);              // lee m trx.
       desde = desde+cantLeidos;
       m = findLargeItemSetsDICV3(instances,vez,m,division,cantInstances);
       x = vez+1;
       vez++;
} while (vez <= m+division);
```

En Instances de WEKA, el Constructor que se utiliza para cargar el archivo ".ARFF" en memoria es:

```
Public Instances(Reader reader) throws IOException {

       StreamTokenizer tokenizer;

       tokenizer = new StreamTokenizer(reader);
       initTokenizer(tokenizer);
       readHeader(tokenizer);
       m_ClassIndex = -1;
       m_Instances = new FastVector(1000);
       while (getInstance(tokenizer, true)) {};
       compactify();
  }
```

Se reemplaza por el constructor y el método siguiente:

```
Instances(Reader reader, int capacity)
proximaInstances(int aLeer).
```

La variable tokenizer (local hasta el momento) pasa a ser una variable del Objeto Instances para poder guardar la posición del archivo que estoy leyendo.

El constructor, solamente inicializa el archivo y la lectura se realiza en el método *proximaInstances(int)*.

```
Public Instances(Reader reader, int capacity) throws IOException {
```

```
      if (capacity < 0) {
        throw new IllegalArgumentException("Capacity has to be positive!");
      }
      mi_tokenizer = new StreamTokenizer(reader);
      initTokenizer(mi_tokenizer);
      readHeader(mi_tokenizer);
      m_ClassIndex = -1;
      m_Instances = new FastVector(capacity);
  }
```

```
Public Instances proximaInstances(int aLeer) throws IOException {

      boolean continua = true;
      int cuenta = 0;

      m_Instances = new FastVector(1000);
      do {
            continua = getInstance(mi_tokenizer, true, aLeer,cuenta);
            cuenta++;
      }     while ((continua)&&(cuenta<aLeer));
      compactify();
      return this;
      }
```

En el método *getInstance()* de la clase Instances se incorporan parámetros: cantidad de registros del archivo que han sido leídos y cantidad que tengo que leer, de manera de sólo incorporar las instancias necesarias.

```
// Add instance to dataset
add(new Instance(1, instance));
      return true;
  }
```

```
// Add instance to dataset
if (cuenta <= aLeer )
      add(new Instance(1, instance));
return true;
  }
```

- Incorporación de la variable m_posLeida en ItemSet para guardar la partición del archivo en el que se generó el itemset.
- En la clase ItemSet se incorporan modificaciones en los métodos *deleteItemSet(FastVector, int, int, int, int)*, *mergeAllItemSets(FastVector, int, int)* y *updateCounters(FastVector, Instances, int, int)* para la verificación de la m_posLeida.

  En la declaración del método

  ```
  public static FastVector deleteItemSets(FastVector itemSets, int
  minSupport, int maxSupport)
  ```

  se agrega la "porción del archivo que estoy leyendo" y la cantidad de divisiones que tiene el archivo.

  ```
  public static FastVector deleteItemSets(FastVector itemSets, int
  minSupport, int maxSupport, int vez, int divisiones)
  ```

  y en el código se reemplaza

  ```
  if ((current.m_counter >= minSupport) )
  ```

  con

```
if ( (current.m_counter >= minSupport) ||((current.m_counter >
0)&&((vez - current.m_posLeida)<divisiones)))
```

El método original de WEKA borra todos los itemsets que no superan el soporte. Se debió incorporar una modificación que evite que se borren los itemsets hasta no haber terminado de contarlos. Es decir, hasta que no se haya dado una vuelta completa al archivo de transacciones desde la creación del itemset, no se elimina o deja de contar.

En el método

```
public static FastVector mergeAllItemSets(FastVector itemSets, int
size, int totalTrans)
```

se incorpora la porción del archivo que se está leyendo y la cantidad de divisiones del archivo que son utilizados para:

posLeida: para cargarlo en el itemset que se genera, y así saber hasta cuándo contarlo.

divisiones: para saber si los dos itemsets a los que se está haciendo merge se terminaron de contar. Si los dos itemsets se terminaron de contar, no se genera nada.

```
public static FastVector mergeAllItemSets(FastVector itemSets, int
size, int totalTrans, int posLeida, int divisiones)
```

Modificación de método *updateCounters()* para que solamente cuente el itemset si no se ha dado una vuelta completa al archivo desde la generación del mismo.

```
Public static void upDateCounters(FastVector itemSets, Instances instances) {

       for (int i = 0; i < instances.numInstances(); i++) {
         Enumeration enum = itemSets.elements();
         while (enum.hasMoreElements())
       ((ItemSet)enum.nextElement()).upDateCounter(instances.instance(i));
       }
}
```

```
Public static void upDateCounters(FastVector itemSets, Instances instances,
       int posLeidaMasTope, int divisiones) {
       ItemSet current;
       int j=0;
       for (int i = 0; i < instances.numInstances(); i++) {
         Enumeration enum = itemSets.elements();
         while (enum.hasMoreElements()) {
              current = (ItemSet)enum.nextElement();
              if (posLeidaMasTope < (current.getM_posLeida()+divisiones))
                     current.upDateCounter(instances.instance(i));

         }
       }
  }
```

- El método *findLargeItemSetsDICV3(Instances,int, int, int, int)* incorpora la cantidad de instancias del archivo, para calcular el soporte. Se controlan las iteraciones para verificar

la porción del archivo que se está procesando. Se incorpora, también, la administración que si el conjunto de Itemset Frencuentes L_i que se está evaluando existe lo actualiza con los nuevos datos y si no existe se genera y retorna.

```
Private int findLargeItemSetsDICV3(Instances instances,int vez, int mayor,
       int divisiones,int cantInstances) throws Exception {

FastVector kMinusOneSets, kSets, k1,k2;
Hashtable hashtable;
int necSupport, necMaxSupport,i = 0;

m_instances = instances;

// Find large itemsets
// minimum support
necSupport = (int)(m_minSupport * (double)cantInstances+0.5);
necMaxSupport = (int)(m_upperBoundMinSupport * cantInstances+0.5);

if (vez < divisiones) {
       kSets = ItemSet.singletons(instances,vez);
       ItemSet.upDateCounters(kSets, instances);
       kSets = ItemSet.deleteItemSets(kSets,necSupport,necMaxSupport,
              vez,divisiones);
} else
       kSets = new FastVector();

do {
       if  (m_Ls.size() <= i)  {
          if (i != 0) {
              ItemSet.upDateCounters(kSets, instances,vez,divisiones);
              kSets = ItemSet.deleteItemSets(kSets,necSupport,necMaxSupport,
                     vez,divisiones);
          }
          m_Ls.addElement(kSets);
          if (kSets.size() != 0)
              mayor = vez;
          kMinusOneSets = kSets;
          kSets = ItemSet.mergeAllItemSets(kMinusOneSets,i, cantInstances,
              vez,divisiones);
          hashtable =
                     ItemSet.getHashtable(kMinusOneSets,kMinusOneSets.size());
          if ((vez == 0) || (m_hashtables.size() <= i))
              m_hashtables.addElement(hashtable);
          return mayor;
       }
       else {
          k2=k1=((FastVector)(m_Ls.elementAt(i)));
          if (i == 0)
              ItemSet.upDateCounters(k1,instances,vez,divisiones);
          if((!(k1.equals(kSets)))&&(k1.size() != 0)&&(kSets.size() != 0))
              k1 = insertarItemSet(k1,kSets);
          if (!(i == 0)) {
              hashtable = (Hashtable)m_hashtables.elementAt(i-1);
              k1 = ItemSet.pruneItemSets(k1, hashtable);
              ItemSet.upDateCounters(k1,instances,vez,divisiones);
          }
          k1 =
              ItemSet.deleteItemSets(k1,necSupport,
                                     necMaxSupport,vez,divisiones);
          kMinusOneSets = k1;
          if (mayor(k1,k2) && (k1.size() != 0) && (k2.size() != 0))
              mayor = vez;
          m_Ls.setElementAt(k1,i);
```

```
        }
        kSets = itemSet.mergeAllItemSets(kMinusOneSets,i,cantInstances,
              vez,divisiones);
        hashtable = ItemSet.getHashtable(kMinusOneSets,kMinusOneSets.size());

        if ((vez == 0) || (m_hashtables.size() <= i))
          m_hashtables.addElement(hashtable);
        else
          m_hashtables.setElementAt(hashtable,i);

        kSets = ItemSet.pruneItemSets(kSets, hashtable);
        i++;
} while (true);

}
```

A.5.2 Detalle de las Modificaciones Introducidas para TDIC

- Se genera la clase ItemSetTDic, dado que las modificaciones a realizar no eran compatibles con la versión de DIC.
- Se incorpora la lectura del archivo que contiene la información de los momentos de las transacciones. Para esto, se genera un nuevo constructor en Instances, que incorpora el nuevo archivo.

```
// Find large itemsets and rules


division =  cantInstances / M;
if ((cantInstances % M) != 0) division++;

reader = new BufferedReader(new FileReader(trainFileString));
reader_t = leer_times(trainFileString_t);

instances = new Instances(reader,reader_t,cantLeidos);
reader_t.mark(cantLeidos);

do {
       if (desde >= cantInstances){
              desde = 0;
              reader = new BufferedReader(new FileReader(trainFileString));
              instances = new Instances(reader,reader_t,cantLeidos);
              reader_t = leer_times(trainFileString_t);
              reader_t.mark(cantLeidos);
       }
       if ((desde+M) < (cantInstances))
              cantLeidos = M;
       else
              cantLeidos = cantInstances - desde;

       instances.proximaInstances(reader_t,cantLeidos);

       desde = desde+cantLeidos;
       m = findLargeItemSetsDICV3(instances,vez,m,division,cantInstances);
       vez++;
} while (vez <= m+division);
```

- Se incorporan los métodos *proximaInstances (BufferedReader, int)* y *getInstance(StreamTokenizer, boolean, BufferedReader, int, int)*

```
Public Instances proximaInstances(BufferedReader reader_t, int aLeer) throws
IOException {

      boolean continua = true;
      int cuenta = 0;

      m_Instances = new FastVector(1000);
      do {
            continua = getInstance(mi_tokenizer, true,
                                    reader_t,aLeer,cuenta);
            cuenta++;
      }     while ((continua)&&(cuenta<aLeer));
      compactify();
      return this;
      }
```

En *getInstance(StreamTokenizer, boolean, BufferedReader, int, int)*, se incorpora el tiempo efectivamente de la instancia.

```
      try{
        linea= reader_t.readLine();
        Date myDate = DateFormat.getDateTimeInstance().parse(linea);

      // Add instance to dataset
      if (cuenta <= aLeer )
            add(new Instance(1, instance,myDate));
```

- En la clase ItemSetTDic, se incorporaron las variables m_t1, m_t2, m_Ftr y m_Ftr_prima, necesarias para guardar la información correspondiente a los tiempos: momento de la primer y última transacción en la que aparece el itemset, y contadores temporales.
- En *deleteItemSet()*, se agrega el intervalo mínimo solicitado por el usuario, para controlar que el soporte mínimo temporal se cumpla en rango.

```
public static FastVector deleteItemSets(FastVector itemSets,
                                    int minSupport,
                                    int maxSupport,
                                    int vez, int divisiones) {

FastVector newVector = new FastVector(itemSets.size());

for (int i = 0; i < itemSets.size(); i++) {
  ItemSet current = (ItemSet)itemSets.elementAt(i);
  if ( (current.m_counter >= minSupport) ||
     ( (current.m_counter > 0) && ((vez - current.m_posLeida) < divisiones) ))
      newVector.addElement(current);
}
return newVector;
  }
```

```
Public static FastVector deleteItemSets(FastVector itemSets,
                                    double minSupport,
                                    double minInt,
                                    int maxSupport,
```

```
                                    int vez,
                                    int divisiones) {

long temporal;
FastVector newVector = new FastVector(itemSets.size());

for (int i = 0; i < itemSets.size(); i++) {
  ItemSetTdic current = (ItemSetTdic)itemSets.elementAt(i);
  if (!(current.m_t2 == null || current.m_t1 == null)) {
    boolean superaSoporte = current.m_counter >= minSupport * current.m_FTr;
    boolean estoyContando = (vez - current.m_posLeida) < divisiones;
    if (superaSoporte && (current.m_FTr > 0) )
       {
       temporal = current.m_t2.getTime() - current.m_t1.getTime();
       temporal = (temporal / 1000)+1; // segundos
       if ((temporal >= minInt) || (estoyContando &&
                                    (current.m_Ftr_prima != 0)))
             newVector.addElement(current);
       }
    else
       if ((current.m_FTr==0) && estoyContando && (current.m_Ftr_prima != 0))
             newVector.addElement(current);
  }
}
return newVector;
}
```

- En el método *mergeAllItemSets(FastVector,int, int, int, int)* que genera a partir de (k-1)-itemsets los k-itemsets, se inicializan las nuevas variables.

```
/**
   * Merges all item sets in the set of (k-1)-item sets
   * to create the (k)-item sets and updates the counters.
   *
   * @param itemSets the set of (k-1)-item sets
   * @param size the value of (k-1)
   * @return the generated (k)-item sets
   */
  public static FastVector mergeAllItemSets(FastVector itemSets, int size,
                                            int totalTrans, int posLeida,
                                            int divisiones) {
... // Hay código anterior
       result.m_counter = 0;
       result.m_posLeida = posLeida;
       result.m_Ftr_prima = result.m_FTr = 0;
       if ((first.m_t1).after(second.m_t1))
             result.m_t1 = first.m_t1;
       else
             result.m_t1 = second.m_t1;

       if ((first.m_t2).before(second.m_t2))
             result.m_t2 = first.m_t2;
       else
             result.m_t2 = second.m_t2;
       if ((result.m_t1).before(result.m_t2) ||
                    (result.m_t1).equals(result.m_t2))
             newVector.addElement(result);

       return newVector;
  }
```

- Los métodos que actualizan los contadores, se modifican para soportar las actualizaciones en las variables que representan los soportes temporales, y actualización de límites de tiempo de los itemsets que se están contando.

```
/**
   * Updates counters for a set of item sets and a set of instances.
   *
   * @param itemSets the set of item sets which are to be updated
   * @param instances the instances to be used for updating the counters
   */
public static void upDateCounters(FastVector itemSets, Instances instances,
int posLeidaMasTope, int divisiones) {
ItemSetTdic current;

For (int i = 0; i < instances.numInstances(); i++) {
  Enumeration enum = itemSets.elements();
  While (enum.hasMoreElements()) {
       current = (ItemSetTdic)enum.nextElement();

       if (posLeidaMasTope < (current.getM_posLeida()+divisiones))
             current.upDateCounter(instances.instance(i),posLeidaMasTope);
       else if ((current.m_FTr == 0) && (current.m_Ftr_prima != 0))
         {
             current.m_FTr = current.m_FTr + current.m_Ftr_prima;
             current.m_Ftr_prima = 0;
         }
  }
}
   }
```

```
/**
   * Updates counter of item set with respect to given transaction.
   *
   * @param instance the instance to be used for ubdating the counter
   */
public final void upDateCounter(Instance instance, int vez) {

if (containedBy(instance)){
       m_counter++;
       if (m_t1 == null)
          m_t1 = m_t2 = instance.getM_time();
       else {
         if (m_t2.before(instance.getM_time())){
             m_t2 = instance.getM_time();
             m_FTr = m_FTr + m_Ftr_prima;
             m_Ftr_prima = 0;
         }
         if (m_t1.after(instance.getM_time())){ // si cambio el t1, es como si
                                                // lo empezara a contar de nuevo.
             m_t1 = instance.getM_time();
             m_Ftr_prima = m_FTr = 0;
             m_posLeida = vez;
             m_counter = 1;
         }
       }
}
if ((!(m_t1 == null))&&
   ((m_t1.before(instance.getM_time())) || (m_t1.equals(instance.getM_time()))))
       if ( (!((m_t1 == null)||(m_t2 == null))) &&
       (((m_t1.before(instance.getM_time())) ||
       (m_t1.equals(instance.getM_time()))) &&
       ((m_t2.after(instance.getM_time()))
       || (m_t2.equals(instance.getM_time()))))
       || (esUnItemset()))
```

```
                m_Ftr_prima++;
}
```

- En el método central que genera los itemsets, las únicas modificaciones introducidas consisten en la modificación de llamadas, incorporando las nuevas variables a considerar.

Apéndice B

Generador de Datos Sintéticos

El generador de datos sintéticos está basado en el concepto expuesto en [3] y comentado en el Capítulo 4. Se requiere el ingreso de ciertos parámetros que darán forma al tipo de datos sintéticos que se requiere y luego de generar las transacciones arma un archivo de texto con el formato y extensión definido para la entrada de datos de las aplicaciones WEKA (Ver Apéndice A).

Fue escrito en lenguaje Java, lo que lo hace altamente portable y compatible con distintos sistemas.

Algunos de las clases más importantes de los cuales se compone son las siguientes:

- **SyntheticDataGenerator.java**

Posee un constructor de clase que requiere que se le ingresen parámetros que definirán el tipo de archivo que se desea realizar.

```
   * @param num_transactions    numero de transacciones a generar
   * @param avg_transaction_size    tamaño promedio de transacción
   * @param num_large_itemsets   numero de Large Itemset a ser usados como
base en la generación de transacciones
   * @param avg_large_itemset_size    tamaño promedio de transacción
   * @param num_items   numero de ítems que aparecen en las transacciones
   * @param correlation_mean   correlacion entre Itemset Frecuente
   * @param corruption_mean   coeficiente de corrupción
```

Tabla B.1: Parámetros del Generador de Datos

Genera con el método *initLargeItemsets()* un conjunto de Itemset frecuentes que luego serán usados en la generación de las transacciones.

Una vez generado el conjunto de Itemset frecuentes comienza a generar las transacciones con el método *getNextTransaction()*. Este método arma la transacción teniendo en cuenta el tamaño de transacción obtenido por Distribución de Poisson con media avg_transaction_size, cuántos ítems va a cambiar para simular ítems que se compran juntos, evalúa si entra o no en el tamaño de la transacción definido.

```
  private void initLargeItemsets()
  {
       // para tomar un item de forma random
       rand_item = new Random();

       // asigno la probabilidad al item
       initItemProbabilities();
```

```
large_itemsets = new Vector(num_large_itemsets);

RandomPoissonDistribution poisson_large_size
  = new RandomPoissonDistribution(avg_large_itemset_size - 1);
RandomExponentialDistribution exp_correlation
  = new RandomExponentialDistribution(correlation_mean);
RandomExponentialDistribution exp_weight
  = new RandomExponentialDistribution();
Random normal_corruption = new Random();

for (int i = 0; i < num_large_itemsets; i++)  {
// El tamaño del LI se saca de la D. de Poisson con media
// avg_large_itemset_size
int large_itemset_size = (int)poisson_large_size.nextLong() + 1;

if (large_itemset_size > num_items)
  large_itemset_size = num_items;

LargeItemset large = new LargeItemset();
large.is = new Itemset(large_itemset_size);

if (i > 0)  {   // un nuevo LI
  LargeItemset prev_large = (LargeItemset)large_itemsets.get(i - 1);

  // Con D. Exponencial se saca que parte del Item se usara con media
  //correlation_mean
    int fraction = (int)(((double)large_itemset_size) *
                   exp_correlation.nextDouble() + 0.5);

    if (fraction > large_itemset_size)
      fraction = large_itemset_size;
    if (fraction > prev_large.is.size())
      fraction = prev_large.is.size();

    // que parte del item se sacara
    if (fraction > 0)
      {
       RandomSample rand_sample
         = new RandomSample(prev_large.is.size(),
                            fraction, rand_sampling);
       long[] sample = rand_sample.nextSample();
       for (int j = 0; j < sample.length; j++)
         large.is.addItem(prev_large.is.getItem((int)sample[j] - 1));
      }
  }

// agrega items aleatorios hasta llenar el itemset
while (large.is.size() < large_itemset_size)
  large.is.addItem(nextRandomItem());
large.weight = exp_weight.nextDouble();

// se asigan un nivel de corruption obtenido con una D. Normal con media
//corruption_mean y varianza 0.1
large.corruption = normal_corruption.nextGaussian() * 0.1 +
                   corruption_mean;

large_itemsets.add(large);
  }

double sum = 0.0;
for (int i = 0; i < num_large_itemsets; i++)
  sum += ((LargeItemset)large_itemsets.get(i)).weight;
for (int i = 0; i < num_large_itemsets; i++)
  ((LargeItemset)large_itemsets.get(i)).weight /= sum;
```

```
      for (int i = 1; i < num_large_itemsets - 1; i++)
        {
      LargeItemset prev_large = (LargeItemset)large_itemsets.get(i - 1);
      LargeItemset large = (LargeItemset)large_itemsets.get(i);
      large.weight += prev_large.weight;
        }
      ((LargeItemset)large_itemsets.get(num_large_itemsets - 1)).weight = 1.0;

  }
```

Tabla B.2: Generador de Itemsets Frecuentes

```
  public Itemset getNextTransaction()
  {
       if (current_transaction >= num_transactions)
         throw new NoSuchElementException("No hay mas elementos para
generar!");

       // tamaño de transaccion obtenido por Distribucion de Poisson con
       // media avg_transaction_size
       int transaction_size = (int)poisson_transaction_size.nextLong() + 1;

       if (transaction_size > num_items)
         transaction_size = num_items;

       Itemset transaction = new Itemset(transaction_size);

       while (transaction.size() < transaction_size)
         {
       LargeItemset pattern = nextRandomLargeItemset();

       // cuantos itesm voy a cambiar de una transaccion
       int pattern_length = pattern.is.size();
       while (pattern_length > 0 && rand_transaction.nextDouble() <
              pattern.corruption)
         pattern_length--;

       // en caso que el Large Itemset no entre en la transaccion lo ponemos
       // en el 50% de los casos
       // y sino se pasan para la otra trasanccion
       if (pattern_length + transaction.size() > transaction_size)
         if (transaction.size() > 0 && rand_transaction.nextDouble() < 0.5)
           {
             ungetRandomLargeItemset();
             break;
           }

       if (pattern_length > 0)
         {
           RandomSample rand_sample = new RandomSample(pattern.is.size(),
                                       pattern_length, rand_sampling);
           long[] sample = rand_sample.nextSample();
           for (int j = 0; j < sample.length; j++)
             transaction.addItem(pattern.is.getItem((int)sample[j] - 1));
         }
         }
       current_transaction++;
       return transaction;
  }
```

Tabla B.3: Generador de Transacciones

Una vez completa la cantidad de transacciones solicitadas comienza el proceso final que es grabar en un archivo de texto con el formato requerido por WEKA. Este proceso es realizado por el método *generoArchivoFinal()*

- **RandomExponentialDistribution.java**

Esta clase es un generador de números aleatorios con distribución exponencial.

Posee 3 constructores, uno para generar números aleatorios con distribución exponencial con media 1, otro con media dada por el usuario y el último con media dada por el usuario y números tomados desde un objeto dado por el usuario también.

```
  public RandomExponentialDistribution()
  {
       this(1.0, new Random());
  }

  public RandomExponentialDistribution(double mean)
  {
       this(mean, new Random());
  }

  public RandomExponentialDistribution(double mean, Random randgen)
  {
       this.mean = mean;
       rand = randgen;
  }
```

Tabla B.4: Constructor de Clase

Una vez inicializado el objeto, el valor se obtiene con la función que a continuación se detalla.

```
public double nextDouble()
  {
       double val;

       do
         val = rand.nextDouble();
       while (val == 0.0);

       return mean * (-Math.log(val));

  }
```

Tabla B.5: Nro. Aleatorio con Distrib. Exponencial

- **RandomPoissonDistribution.java**

Esta clase es un generador de números aleatorios con distribución de Poisson.

Posee 2 constructores, uno para generar números aleatorios con distribución de Poisson con media dada por el usuario y el otro con media dada por el usuario y números tomados desde un objeto dado por el usuario también.

```
public RandomPoissonDistribution(long mean)
{
     this(mean, new Random());
}

public RandomPoissonDistribution(long mean, Random randgen)
{
     this.mean = mean;
     rand = randgen;
}
```

Tabla B.6: Constructor de Clase

Una vez inicializado el objeto el valor se obtiene con la función que a continuación se detalla.

```
public long nextLong()
{
     if (mean < 100)
       {
     double p = Math.exp(-(double)mean);
     long N = 0;
     double q = 1.0;

     while (true)
       {
         // Q2. [obtiene una variable uniforme]
         double U = rand.nextDouble();
         // Q3. [multiplica]
         q = q * U;
         // Q4. [testea]
         if (q >= p)
           N = N + 1;
         else
           return N;
       }
       }
     // cuando mean es grande, el valor se aproxima usando una D. Normal
     else
       {
     double z = rand.nextGaussian();
     long value = (long)(mean + z * Math.sqrt(mean) + 0.5);
     if (value >= 0)
       return value;
     else return 0;
       }

}
```

Tabla B.7: Nro. Aleatorio con Distrib. de Poisson

Apéndice C

Generador de Cola de Llegadas o Clientes

El generador de tiempos de transacciones está basado en la teoría de colas detallada en el Capítulo 4. Se implementó un simulador de arribos de personas a servidores que brindan un servicio por un determinado servicio. Si el servidor está ocupado se incorporan en una cola de espera hasta ser atendidos. Se puede setear o configurar el tipo de servidor que se desea simular y también el tipo cola que se desea; pueden ser uno o más servidores, cola finita o infinita. A medida que van pasando los clientes por los servidores se registra el instante en un archivo de texto que será tomado como el tiempo de las transacciones que fueron generadas con el generador de datos sintéticos.

Fue escrito en lenguaje Java, lo que lo hace altamente portable y compatible con distintos sistemas.

Algunos de las clases más importantes de los cuales se compone son las siguientes:

- **QueueSim.java**

Es un applet que simula manejo de cola, de servidores con ciertas características que serán dadas por los parámetros que se eligieron al inicio de la simulación. Da comienzo al emisor de donde salen las personas o arrivos, al moderador de la cola y los servidores encargados de realizar el servicio.

```
//------------------ inicializacion del applet -------------------------
 public void init() {

       setFont(new Font("Arial",Font.PLAIN,12));
       // crear un buffer off-screen
       bg_image = createImage(WIDTH, HEIGHT);
       bg_graphics = bg_image.getGraphics();

       // crear botones
       setLayout(new FlowLayout(FlowLayout.LEFT,10,0));
       start = new Button("start simulacion");
       add(start);
       stop = new Button("stop simulacion");
       add(stop);
       redefine = new Button("redefinir");
       add(redefine);
       c_limit_simulation = new Checkbox("simulacion 5 minutos",
                                         null,true);
       c_limit_simulation.setForeground(Color.white);
       c_limit_simulation.setBackground(Color.black);
       add(c_limit_simulation);

       // crea una cola global que tendra los parametros
       // access_rate,service_rate y el numero de servers.
       global_queue = new QueueDefs();
       sim_params = new Label(fillSpaces(200));
```

```
        sim_params_unique = new Label(fillSpaces(200));
        add(sim_params);
        add(sim_params_unique);

        // correr la demo
        demo();
   }
   //----------------- reinicio de simulacion--------------------------
   public void restart() {

        // crea una cola global con los parametros
        // access_rate,service_rate y numero de serves.
        global_queue = new QueueDefs();

        // crea una ventana de mensajes - para errores / informacion
        createMsg();
        message.post("Elija una cola de simulacion.");
        // llama a la ventana que tiene los parametros de la cola
        if (w_define == null) {
             w_define = new DefineQueue("Tipos de colas de simulacion",
                                        global_queue,message);
             w_define.setSize(500,200);
             w_define.setLocation(1,140);
        }
        w_define.show();
   }
//----------- crea  un objeto de simulacion y lo corre ---------------
   public void startSim() {
        int server_y_pos;
        int i;

        // indica si la simulacion esta corriendo
        sim_started = true;

        // todas las ventanas de parametros estaran cerradas
        if (w_define != null) w_param = w_define.getParamWin();
        if (w_param != null)  { w_param.setVisible(false); w_param = null;
}
        if (w_define != null) { w_define.setVisible(false);
                                w_define = null; }

        createMsg();

        if (global_queue.access_rate == 0) {
             // valores defecto a la simulacion
             defaultPrm();
        }

        // muestra parametros
        sim_params.setText(simPrm());
        sim_params_unique.setText(simPrmU());

        // crea todos los objetos

        queue     = new Queue(message,global_queue);

        // ventana de estadisticas
        current_date = new Date();
        if (stat != null) { stat.setVisible(false); stat = null; }
        stat = new Statistics("Sistema de estadísticas",
                              queue,global_queue,current_date);
        stat.setSize(370,250);
        stat.setLocation(380,350);
        stat.show();
```

```
        // conductor
        animator  = new Animator(bg_graphics, this,stat,queue);

        // entrada
        p_access  = new  Poisson(global_queue.access_rate);
        door      = new  Door(p_access,animator,queue,message);

        // servidores
        servers_lock = new WaitObject();
        servers = new Vector(5);
        server_y_pos = 190 - (global_queue.servers_no*30 - 10);
        for (i = 1 ; i <= global_queue.servers_no ; i++) {
              servers.addElement(new Server(server_y_pos,
                    global_queue.service, animator,message, queue,
                    servers_lock));
              server_y_pos += 60;
        }

        // cola con server lockeado y conductor trabajando
        queue.getObjects(servers_lock, animator);

        // comienzo
        for (i = 0 ; i < global_queue.servers_no ; i++) {
              ((Server) (servers.elementAt(i))).start();
        }
        door.start();
        animator.start();

        // miro si son 5 minutos
//      if (c_limit_simulation.getState()) {
              // crea un timeout para parar en 5 minutos
//            timer  = new Timer(this,queue);
//            timer.start();
//      }
   }
```

Tabla C.1: Simulador de Cola

- **Animator.java**

Tiene el objetivo de hacer de moderador de las entradas de personas o conductor de las mismas y creación de los servers.

Las entradas serán objetos animados que aparecerán en el escenario y se encolarán en caso de que el servidor esté ocupado o pasarán a ser atendidas en caso contrario.

```
  // ----------  Metodo publico lllamado para inicializar -------
   // Retorna: un objeto recibidor de señales
   //  Esta funcion retorna sin esperar de que la animacion termine
   public WaitObject doAnimation(int x, int y, int destx, int desty, int
                                 img, int speed, int trigger) {
        AnimatedObject new_obj;
        WaitObject result = new WaitObject();
        Image obj_image;

        if (img == 1)
              obj_image = smiley;
        else if (img == 2)
              obj_image = smiley2;
        else
```

```
            obj_image = smiley3;

      new_obj = new
          AnimatedObject(x,y,destx,desty,obj_image,result,speed,trigger);
      g.drawImage(obj_image,x,y,app);// dibuja la imagen en la
                                     //posicion inicial
      all_objects.addElement(new_obj); // la agrega a un vector
      return result;  // retorna un objeto que sera llamado cuando la
                        //animacion aparece
}
public void drawServer(int x, int y, Color c) {
      synchronized (color_mutex) {
            g.setFont(new Font("TimesRoman",Font.PLAIN,14));
            g.setColor(c);
            g.fillRect(x,y-2,40,40);
            y+=10;
            g.setColor(Color.white);
            g.drawString("S",x,y);
            g.drawString("e",x+7,y+5);
            g.drawString("r",x+14,y+10);
            g.drawString("v",x+21,y+15);
            g.drawString("e",x+28,y+20);
            g.drawString("r",x+35,y+25);
            g.setFont(new Font("TimesRoman",Font.BOLD,24));
      }
}
// ------  maquina conductora, mueve las imagenes en la ventana ----
public void run() {
      int i;
      int x,y,oldx,oldy,speed;

      while (true) {
         // por cada objeto animado activo, lo mueve un paso a su destino
            for (i = 0 ; i < all_objects.size() ; i++) {
                  curr_object = (AnimatedObject) all_objects.elementAt(i);

            // Si alcanza el destino, para el objeto
                  if ((curr_object.destinationY == curr_object.curY) &&
                        (curr_object.destinationX == curr_object.curX)) {
                     (curr_object.signal).setEvent();
                     all_objects.removeElementAt(i);
                     i--;
                     if (curr_object.trigger == 1)
                           queue.insertPerson();
                  }
                  else {
                     oldx = curr_object.curX;
                     oldy = curr_object.curY;
                     speed = curr_object.speed;

                     if (curr_object.destinationX > curr_object.curX)
                           curr_object.curX += speed;
                     else if (curr_object.destinationX < curr_object.curX)
                           curr_object.curX -= speed;

                     if (curr_object.destinationY > curr_object.curY)
                           curr_object.curY += speed;
                     else if (curr_object.destinationY < curr_object.curY)
                           curr_object.curY -= speed;

                     moveTo(curr_object.image, oldx, oldy,
                            curr_object.curX, curr_object.curY);
                  }
            }
```

```
   // cada 10 segundos updetea las estadisticas
        if (stat_counter++ % 400 == 0) {
           stat.recalc();
        }

   // actualiza pantalla
        app.repaint();

        try {
           sleep(25);     // Sleep por el while
        } catch(InterruptedException e) {};

         } // end while
   }
```

Tabla C.2: Animador o Moderador

- **DefineParams.java**

Esta clase permite definir los parámetros generales para la simulación.

Esta clase es extendida por DefineParamsMf1, DefineParamsMm1, DefineParamsMms, DefineParamsMn1 y DefineParamsMu1 que permiten definir los parámetros especiales de cada tipo de simulación (configuración de colas, servidores y servicios).

```
M/M/1  - Un server, promedio de entradas poisson, promedio de servicio
exponencial.
M/M/S - s servers , promedio de entradas poisson, promedio de servicio
exponencial.
M/F/1  - Un server, promedio de entradas poisson, promedio de servicio fijo.
M/U/1 - Un server, promedio de entradas poisson, promedio de servicio
uniforme.
M/N/1 - Un server, promedio de entradas poisson, promedio de servicio
normal.
```

Tabla C.3: Parámetros

- **Door.java**

La clase Door es el generador de entradas para la cola y se queda esperando un lapso de tiempo para generar una nueva entrada.

```
public void run() {
         while (true) {
         int access_time;

         message.post("Nuevo cliente entra en la cola");
         // una carita animada a la cola
         // el doAnimation corre asincronicamente, el manejador de entradas
         //no se queda esperando que llegue
         animator.doAnimation

       (animator.DOOR_POSITION,170,animator.QUEUE_POSITION,170,1,10,1);
         // cuando la animacion finaliza, el Animator o conductor notifica a
         // la cola
         access_time = prob.rand();// Calcula el tiempo de espera entre
```

```
                                    //arrivos
        try {
       sleep(access_time); // dormido hasta que una nueva persona llega
        }
        catch(InterruptedException e) {};

        } //while
```

Tabla C.4: Generador de Entradas

- **Normal.java**

La clase Normal extiende la clase Probability (clase abstracta que genera un número aleatorio dada una distribución) y genera números entre 0 y 1 con distribución normal.

```
  int rand() {
        float v1=0,v2=0,w=2,y;
        int tmp;
        while (w > 1) {

              v1 = (float) Math.random()*2 - 1;
              v2 = (float) Math.random()*2 - 1;
              w = (float) (Math.pow(v1,2) + Math.pow(v2,2));
        }

        y = (float) Math.sqrt( ((-2)*Math.log(w)) / w );
        if (Math.random() < 0.5) {
              tmp = (int) (1000 * ((y*v1)*variance  + average));
              if (tmp < 0) tmp = 0;
        }
        else  {
              tmp = (int) (1000 * ((y*v2)*variance  + average));
              if (tmp < 0) tmp = 0;
        }
        return tmp ;
   }
```

Tabla C.5: Nro. con Distrib. Normal

- **Poisson.java**

La clase Poisson extiende la clase Probability (clase abstracta que genera un número aleatorio dada una distribución) y genera números entre 0 y 1 con probabilidad de Poisson.

```
Poisson(int param) {
        lamda = param;
   }

int rand() {
   return (int) ((((Math.log(1 - Math.random()) )*(-1) ) /lamda) * MINUTE);
   }
```

Tabla C.6: Nro. con Prob. de Poisson

- **Uniform.java**

La clase Uniform extiende la clase Probability (clase abstracta que genera un número aleatorio dada una distribución) y genera números entre 0 y 1 con probabilidad uniforme.

```
  Uniform (int param1,int param2) {
        low = param1;
        high = param2;
        average = (float) (high + low) /2;
        variance = (float) Math.pow((high - low),2) /12 ;
   }

   int rand() {
        return (int) (MINUTE / ((high-low)*Math.random()+low));
   }
```

Tabla C.7: Nro. con Prob. Uniforme

- **Queue.java**

La clase Queue es la encargada de mantener en la cola las personas que han salido del emisor y no pueden ser atendidas aún, y de enviar la persona que está lista para salir cuando el servidor puede atender. Si la cola se definió finita y el cliente que está llegando no puede ser encolado, es descartado.

- **Server.java**

La clase Server representa el objeto servidor que atiende clientes. Cada server es independiente de otro server y de los clientes. El server pide un cliente a la cola, hace el servicio y lo despacha. Si la cola esta vacía no hace nada, se queda esperando algún arribo.

```
   public void run () {
        WaitObject obj;
        int service_time;
        long sleep_time;

        while (true) {
                    // toma una persona de la cola
             // si la cola esta vacia, el server duerme
        person = null;
        while (person == null) {
              person = queue.getPerson();
              if (person == null)
                    wait_object.waitForEvent();
        }
        message.post("Cliente entrando en la estacion de servicio");
        obj = animator.doAnimation  (animator.QUEUE_POSITION+40, 170,
                         animator.SERVICE_POSITION,server_y_pos,2,10,0);
        obj.waitForEvent(); // espera hasta que la carita llegue al lugar
del
                              //server
        animator.drawServer   // Cambio el server al color ocupado
              (animator.SERVICE_POSITION+33, server_y_pos,
```

```
animator.SERVER_BUSY);
         service_time = prob.rand();  // Calcula el tiempo del servicio
         date = new Date();
         person.start_of_service = date.getTime();

         sleep_time = service_time - ((long)person.start_of_service -
                      (long)person.end_of_waiting);
         if (sleep_time < 0)
               sleep_time = 0;

         try {
               sleep(sleep_time);
         } catch(InterruptedException e) {};

         person.end_of_service = (long)person.start_of_service +
                                  service_time;
         message.post("Cliente Servido");
         queue.people_served++;
         person.end_flag = true;
         animator.eraseSmiley(animator.SERVICE_POSITION,server_y_pos);
                // Cambio el color del server (libre)
         animator.drawServer
               (animator.SERVICE_POSITION+33, server_y_pos,
                animator.SERVER_FREE);
                // Saco el cliente que finalizo fuera del server
         animator.doAnimation(animator.EXIT_POSITION,server_y_pos,
                              650,server_y_pos,2,10,0);
         } // while
   }
```

Tabla C.8: Servidor

Bibliografía

[1] Ale, Juan M. - Rossi, Gustavo H., *An Approach to Discovering Temporal Association Rules.* ACM - 2000, march 2000.

[2] Ale, Juan M. - Rossi, Gustavo H., *The Itemset´s Lifespan Approach to Discovering Temporal Association Rules.* ACM – Workshop on TDM – KDD-2002, Julio 2002.

[3] Rakesh Agrawal y Ramakrishnam Srikant, *Fast Algorithms for Minig Association Rules* - IBM Almaden Research Center - Proc. of the 20th Int'l Conference on Very Large Databases. Santiago, Chile, Sept. 1994. Expanded version available as IBM Research Report RJ9839.

[4] Agrawal, R. - Skrikant, R., *Mining Sequential Patterns*. Proc. IEEE Int'l Conference on Database Engineering: 3 - 14. 1995.

[5] Ming-Syan Chen, Jong Soo Park y Philip S. Yu, *Efficient Data Mining for Traversal Patterns*. IEEE Transactions on Knowledge and Data Engineering, Vol. 10, Nro. 2. Marzo/Abril de 1998.

[6] Bettini, C. - Wang, X. - Jajodia, S. - Lin, J., *Discovering Frequent Event Patterns With Multiple Granularities In Time Sequences*. IEEE TOKDE Vol. 10 Nº2: 222-237. April 1998.

[7] Servey Brin, Rajeev Motwani, Jeffrey D. Ullman, Shalom Tsur, *Dynamic Itemset Counting and Implication Rules for Market Basket Data*. Junio de 1997.

[8] Chakrabarti, S. - Sarawagi, S. - Dom, B., *Mining Surprising Patterns Using Temporal Description Length*. Proc. 24th VLDB Conf. 1998.

[9] Chan, K. - Fu, A., *Efficient Time Series Matching by Wavelets*. Proc. IEEE 15th Intl. Conf. On Data Engineering, 1999.

[10] Frans Coenen, *Dynamic Itemset Counting*. Department of Computer Science, The University of Liverpool. Enero de 2000. http://www.csc.liv.ac.uk.

[11] Chen, X. - Petrounias, I., Hethfield, H., *Discovering Temporal Associations Rules in Temporal Databases*. Proc. Int'l Workshop IADT'98. July 1998.

[12] Ian H. Witten, Eibe Frank, *WEKA - Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers – San Francisco 2000.

[13] Mannila, H. - Taivonen, H. - Verkamo, I., *Discovery of Frequent Episodes in Sequences*. KDD'95. AAAI:210-215. August 1995.

[14] Özden, B. - Ramaswamy, S. - Silberschatz, A, *Cyclic Association Rules*. ICDE 1998.

[15] Robert Cooley, Bamshad Mobaster and Jaideep Srivastava, *Grouping Web Page References Into Transactions for Mining World Wide Web Browsing Patterns*. Technical Report (TR 97-021) Department of Computer Science University of Minnesota. 26 de Junio de 1997.

[17] Ramaswamy, S. - Mahajan, S. - Silberschatz, A., *On the Discovery of Interesting Patterns in Associations Rules*. Proc. 24th VLDB Conf. 1998.

[18] Robert Cooley, Bamshad Mobaster and Jaideep Srivastava, *Web Mining: Information and Pattern Discovery on the World Wide Web*. Technical Report (TR 97-027) Department of Computer Science University of Minnesota.
9 de Julio de 1997.

[19] Osmar R. Zaïane y Jiawei Han, *WebML: Querying the World-Wide Web for Resources and Knowledge*.
http://www.cs.sfu.ca/research/groups/DB/sections/publication/kdd/kdd.html
Proc. (CIKM'98) Int'l Workshop on Web Information and Data Management (WIDM'98)
Bethesda, Maryland, Nov. 1998, pp. 9-12.

[20] Heikki Mannila, Hannu Toivonen, A. Inkeri Verkamo; *Efficient algorithms for discovering association rules*. KDDD-94: AAAI Workshop on Knowledge Discovery in Databases, Julio 1994.