Árboles de decisión para grandes conjuntos de datos

por

Anilu Franco Arcega

Tesis sometida como requisito parcial para obtener el grado de

DOCTOR EN CIENCIAS EN LA ESPECIALIDAD EN CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Julio2010

Tonantzintla, Puebla

Supervisada por:

Dr. J. Ariel Carrasco Ochoa, INAOE
Dr. Guillermo Sánchez Díaz, UdeG

Para mis papis y hermana.
Este es un logro también de ustedes.
Los AMO

Agradecimientos

A DIOS por cada día de vida que me da y por permitirme alcanzar un anhelo más. Siempre estas en mi corazón.

A mis papis porque todos los kilómetros que hemos recorrido juntos para llevar a cabo esta aventura han sido una muestra de su apoyo incondicional. Gracias por todo lo que me dan. Los amo mucho.

A Bere por apoyarme incansablemente en cada decisión que he tomado, por estar ahí cada vez que lo necesito, eres muy importante en mi vida. Te amo.

A Esteban por ser una luz en mi vida. Gracias por todo el apoyo, amor y comprensión que me has brindado. Te amo.

A Chema, Lili, Erasmo, Gerardo, Ali, Judith, Gerardo, Azucena, Ivan, Gris, Chagüita, Gilito, Fabiola, Paul, Carmen, Crescencio, Xochilt, Fernando, Lolita, Cuco y el resto de mi familia por darme su apoyo y amor. Los quiero mucho.

A todos mis amigos, en especial a Janet, Oscar, Edith y Juan, por apoyarme siempre, por todas esas divertidas pláticas y por compartir un pedacito de su vida conmigo.

De manera muy significativa quiero agradecer a Dr. J. Ariel Carrasco Ochoa, Dr. J. Francisco Martínez Trinidad y Dr. Guillermo Sánchez Díaz por su paciencia, guía y asesoría. Sus ideas y apoyo permitieron la culminación de este proyecto. Ha sido una gran ayuda contar con ustedes en todo momento.

A mis revisores Dra. Ma. del Pilar Gómez Gil, Dr. Jesús A. González Bernal, Dr. Manuel Montes y Gómez, Dr. Eduardo Morales Manzanarez y Dr. Jesús M. Pérez de la Fuente por dedicar tiempo a la revisión de este trabajo y por sus sugerencias y observaciones realizadas.

Finalmente, agradezco al Consejo Nacional de Ciencia y Tecnología (CONACyT) por el apoyo brindado para la realización de esta investigación y al Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE) por ser la institución que me permitió realizar uno de mis grandes sueños.

Resumen

Los Árboles de Decisión son de los algoritmos de clasificación supervisada más utilizados. Actualmente existen diversos algoritmos de generación de árboles de decisión, sin embargo, son pocos los que permiten procesar grandes conjuntos de datos. Además, aquellos que lo permiten tienen diversas restricciones, por ejemplo, en cuanto al manejo de memoria y al número de veces que recorren el conjunto de entrenamiento para generar el árbol de decisión, o bien algunos algoritmos no usan el conjunto de entrenamiento completo o tienen parámetros que pueden ser difíciles de determinar por el usuario.

Por esta razón, en esta tesis se proponen algoritmos para la generación de árboles de decisión para grandes conjuntos de datos, que superan las limitaciones de los algoritmos más recientes del estado del arte, asumiendo que el número de clases es mucho menor que el número de objetos en el conjunto de entrenamiento. Los algoritmos propuestos usan todo el conjunto de entrenamiento para generar el árbol de decisión, sin necesidad de almacenarlo completo en memoria. En particular, en esta tesis se proponen dos algoritmos que generan árboles de decisión multivaluados para objetos descritos por atributos numéricos. El primer algoritmo utiliza todo el conjunto de atributos, como atributos de prueba, en los nodos internos del árbol de decisión. Sin embargo, si los objetos están descritos en términos de una gran cantidad de atributos, el tiempo de procesamiento que emplea este algoritmo, cuando se recorre el árbol de decisión, puede ser muy grande. Por esta razón, se propone el segundo algoritmo, el cual utiliza subconjuntos de atributos en los nodos internos. No obstante, aunque se generen árboles de decisión multivaluados con subconjuntos de atributos, el recorrido del árbol de decisión puede ser costoso todavía. Por este motivo, en esta tesis también se proponen dos algoritmos para generar árboles de decisión univaluados. El primero de estos algoritmos es para objetos descritos por atributos numéricos y el segundo es para objetos con atributos mezclados.

Con base en los resultados experimentales, concluimos que los algoritmos

propuestos en esta tesis son más rápidos que los algoritmos más recientes para generación de árboles de decisión a partir de grandes conjuntos de datos, obteniendo una calidad de clasificación competitiva. Por lo cual, podemos afirmar que los algoritmos propuestos son una buena solución al problema de la generación de árboles de decisión para grandes conjuntos de datos.

Abstract

Decision Trees are are among the most used supervised classification algorithms. Currently, there are several algorithms for building decision trees, however, just a few of these algorithms allow processing large datasets. Besides, those algorithms designed for processing large datasets have some restrictions, for example: spatial restrictions; the number of times that they have to scan the whole training set for building the decision tree; some algorithms only use a small subsample of the training set, but for obtaining this subsample they spend a lot of time, specially for large datasets; other algorithms use several parameters, which can be very difficult to determine by the user.

For this reason, in this thesis we propose algorithms for building decision trees for large datasets, that solve the restrictions of the most recent algorithms in the state of the art, considering that the number of classes is lesser than the number of instances in the training set. The proposed algorithms use the whole training set for building the decision tree, without storing the whole training set in memory. In particular, in this thesis, we propose two algorithms for building multivariate decision trees for instances described by numeric attributes. The first algorithm uses all the attributes in the internal nodes of the decision tree. However, if the instances are described by a large number of attributes, the time needed for traversing the tree can be too long. For this reason, we propose a second algorithm, which uses splitting attribute subsets in the internal nodes. Although the previous algorithm generates multivariate decision trees using splitting attribute subsets, the time needed for traversing the decision tree can also be too long. Hence, in this thesis, we propose two algorithms for building univariate decision trees. The first one for instances described by numeric attributes, and the second for instances with mixed attributes.

Based on the experimental results, we can conclude that our algorithms are faster than the most recent algorithms for building decision trees for

large datasets, maintaining competitive accuracy. Therefore, the proposed algorithms are a good option for building decision trees for large datasets.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

Dentro del Reconocimiento de Patrones uno de los problemas más estudiados es la Clasificación Supervisada. Este problema tiene como objetivo determinar la pertenencia de un objeto (descrito por un conjunto de atributos) a una o varias clases, basándose en la información contenida en un conjunto de objetos previamente clasificados (conjunto de entrenamiento) [RSyJMT99].

Existen diversos algoritmos para resolver el problema de clasificación supervisada [RD01]. Sin embargo, entre los algoritmos más utilizados para resolver este tipo de problemas se encuentran los árboles de decisión. En este trabajo se abordará el problema de clasificación supervisada, proponiendo diversos algoritmos que permitan la generación de árboles de decisión.

Un árbol de decisión (AD) es una estructura que se compone de nodos internos, hojas y arcos. Los nodos internos están caracterizados por uno o varios atributos de prueba y de estos nodos se desprenden uno o más arcos. Cada uno de estos arcos tiene asociado un valor para cada atributo de prueba y estos valores determinan qué camino seguir en el recorrido del árbol. Cada hoja tiene asociada una etiqueta de clase, la cual será asociada a los objetos que lleguen a esa hoja.

Recientemente, se ha abordado el problema de generar ADs para grandes conjuntos de datos (conjuntos de datos que tienen millones de objetos, en específico en esta tesis se manejarán conjuntos de hasta 6,500,000 de objetos). Estos conjuntos cuentan con una gran cantidad de objetos y en muchos de los casos no caben en la memoria disponible, por lo cual la aplicación de algoritmos convencionales de generación de ADs (algoritmos en los cuales es necesario mantener en memoria a todo el conjunto de datos para construir un AD) es muy costosa, tanto en tiempo como en espacio, siendo en algunos

casos inaplicables.

El hecho de tener una gran cantidad de datos y no contar con herramientas que los puedan procesar ha producido un fenómeno descrito como: riqueza en datos pero pobreza en información [HK01]. El crecimiento constante de datos, los cuales son guardados en grandes bases de datos, ha excedido la capacidad del ser humano para poder analizarlos. Además, en ciertos problemas prácticos se puede tener un flujo constante de datos, con lo que resultaría más difícil poder hacer un análisis de la información.

Cuando se trata con el problema de generar árboles de decisión, contar con un conjunto de entrenamiento que tiene un gran número de objetos hace que el entrenamiento del clasificador sea muy costoso en tiempo y espacio. Este comportamiento se debe a que el tiempo requerido para procesar todos los objetos de entrenamiento por el clasificador es muy grande y, por otra parte, porque la memoria resulta ser insuficiente para almacenar todos los objetos del conjunto de entrenamiento.

Por esta razón se han desarrollado algoritmos de generación de ADs capaces de trabajar con grandes conjuntos de datos, sin tener que mantener en memoria todo el conjunto de entrenamiento. Ejemplos de algoritmos de este tipo son: SLIQ [MAR96], SPRINT [SAM96], CLOUDS [ARS98], RainForest [GRG00], BOAT [GGRL99], ICE [YAR99], VFDT [DH00], RainForest mejorado [NTC07] y BOAI [YWYC08].

La problemática principal atacada por estos algoritmos es precisamente poder manipular todo el conjunto de entrenamiento, a fin de inducir un árbol de decisión. Sin embargo, algunos de los algoritmos mencionados continúan teniendo restricciones en cuanto al manejo de la memoria, por la representación que usan para el conjunto de entrenamiento (como SLIQ, SPRINT, CLOUDS, BOAI), o bien porque no procesan todos los objetos de entrenamiento al generar el árbol de decisión (como BOAT e ICE).

Otro problema de los algoritmos para generar ADs que procesan grandes conjuntos de datos, es el número de veces que recorren el conjunto de entrenamiento, algunos de ellos (como RainForest) recorren el conjunto hasta dos veces en cada nivel del árbol, lo que resulta en un tiempo de procesamiento elevado. Por otra parte, existen algoritmos que necesitan de varios parámetros para su funcionamiento (por ejemplo VFDT), los cuales pueden ser difíciles de definir por el usuario.

Observando las limitaciones anteriores se puede notar que los algoritmos existentes usan representaciones costosas para manejar el conjunto de entrenamiento, o bien emplean tiempo adicional para procesar el conjunto de entrenamiento, antes de iniciar la generación del AD. De las limitaciones

vistas en los algoritmos de generación de ADs para grandes conjuntos de datos, el objetivo general de esta tesis es:

Proponer algoritmos para generar árboles de decisión para grandes conjuntos de datos, que procesen todo el conjunto de entrenamiento en un tiempo menor que el de los algoritmos existentes y que obtengan una calidad de clasificación competitiva.

Esta tesis tiene los siguientes objetivos particulares:

1. Desarrollar un algoritmo que genere árboles de decisión multivaluados, con todo el conjunto de atributos, para grandes conjuntos de datos con atributos numéricos.

2. Desarrollar un algoritmo que genere árboles de decisión multivaluados, con subconjuntos de atributos, para grandes conjuntos de datos con atributos numéricos.

3. Desarrollar un algoritmo para generar árboles de decisión a partir de grandes conjuntos de datos mezclados (con atributos numéricos y no numéricos).

En específico, en esta tesis se desarrollan cuatro algoritmos para la generación de ADs a partir de grandes conjuntos de datos. En todos los algoritmos propuestos, con la finalidad de poder procesar todo el conjunto de entrenamiento para generar el AD sin almacenarlo completo en memoria, se procesa todo el conjunto de entramiento de manera incremental, y se utiliza solamente un pequeño conjunto de objetos para expandir un nodo en el AD.

Para abordar el problema de generación de ADs para grandes conjuntos de datos, debido a que un algoritmo de generación de ADs emplea el mayor tiempo de procesamiento en la expansión de un nodo, el primer algoritmo que se propone construye ADs multivaluados utilizando todo el conjunto de atributos en los nodos internos del AD (inicialmente utilizaremos conjuntos de datos con atributos numéricos con la finalidad de simplificar el problema). Siguiendo esta idea, utilizar todos los atributos en los nodos internos, hace que nuestro primer algoritmo evite tener que evaluar los atributos para alegir un atributo de prueba en un nodo a expandir. Sin embargo, este algoritmo utiliza la mayor parte del tiempo de procesamiento en recorrer el AD, ya que tiene que evaluar todos los atributos para descender de un nivel a otro. Por esta razón se desarrolla el segundo algoritmo, el cual utiliza subconjuntos

de atributos en los nodos internos, reduciendo el tiempo de procesamiento al recorrer el AD, especialmente cuando los subconjuntos elegidos son pequeños. Siguiendo la idea para generar un AD de los algoritmos propuestos, se propone un tercer algorimo que permite generar ADs univaluados para grandes conjuntos de datos numéricos. Este algoritmo es desarrollado con la finalidad de generar ADs más comprensibles por el usuario. Además, en este algoritmo se propone una nueva forma de encontrar el atributo de prueba en un nodo a expandir. Finalmente, debido a que es común encontrar conjuntos de datos mezclados, el último algoritmo es desarrollado para procesar este tipo de conjuntos de datos.

La principal contribución de este trabajo es el desarrollo de nuevos algoritmos para la generación de árboles de decisión a partir de grandes conjuntos de datos, competitivos en calidad y más rápidos que los actuales.

El resto del documento está organizado de la siguiente manera, el capítulo 2 describe algunos conceptos básicos relacionados con ADs. El capítulo 3 presenta el trabajo relacionado con algoritmos de generación de ADs para grandes conjuntos de datos. En el capítulo 4 se introduce un algoritmo que genera ADs multivaluados, los cuales utilizan todo el conjunto de atributos en sus nodos internos, para grandes conjuntos de datos numéricos. El capítulo 5 presenta un nuevo algoritmo para la generación de ADs multivaluados, que eligen subconjuntos de atributos en sus nodos internos, para grandes conjuntos de datos numéricos. En el capítulo 6 se introduce un algoritmo que genera ADs univaluados para grandes conjuntos de datos numéricos; en el capítulo 7 se propone un algoritmo de generación de ADs univaluados para grandes conjuntos de datos mezclados. Finalmente, el capítulo 8 expone las conclusiones y las contribuciones derivadas de esta investigación, así como el trabajo futuro.

Capítulo 2

Conceptos básicos

Un Árbol de Decisión (AD) está compuesto de un nodo llamado *raíz*, el cual no tiene arcos incidentes o de entrada, y de un conjunto de nodos que poseen un arco de entrada. Aquellos nodos con arcos de salida son llamados *nodos internos* o *nodos de prueba* y los que no tienen arcos de salida son conocidos como *hojas, nodos terminales o nodos de decisión* [RM05]. En la Figura 2.1 se pueden observar los elementos que componen a un AD.

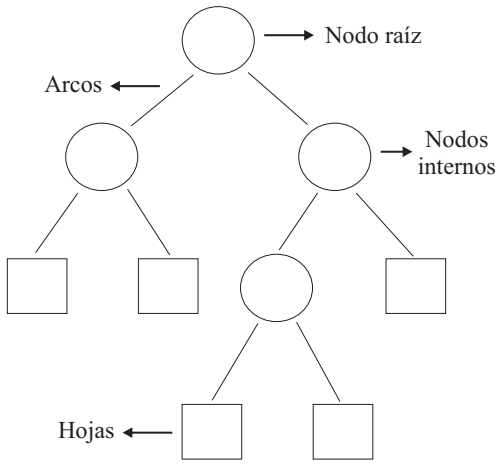


Figura 2.1: Elementos de un AD.

Los principales objetivos que se persiguen al crear un AD [SL91] son:

- Clasificar correctamente la mayor cantidad de objetos del conjunto de entrenamiento.
- Generalizar, durante la construcción del árbol, el conjunto de entrenamiento a fin de que nuevos objetos sean clasificados con el mayor porcentaje de aciertos posible.

Un algoritmo de generación de ADs consta de 2 etapas: la primera es la etapa de inducción del árbol y la segunda es la etapa de clasificación. En la primera etapa se construye el AD a partir del conjunto de entrenamiento. La construcción del árbol inicia eligiendo uno o más atributos de prueba, con los cuales se particiona el conjunto de objetos del conjunto de entrenamiento en un conjunto de nodos hijos. Cada arco entre el nodo padre y un nodo hijo tiene asociado un valor (o una combinación de valores) que caracteriza a los objetos del nodo hijo. Este procedimiento continúa con los nodos hijos hasta que el nuevo nodo creado tenga objetos de una sola clase convirtiendose en nodo hoja, al cual se le asigna la etiqueta de esa clase. La construcción del AD termina cuando ya no hay nodos que expandir.

En la segunda etapa del algoritmo, para clasificar un objeto nuevo usando el árbol construido, se recorre el árbol desde el nodo raíz hasta una hoja y se asigna la etiqueta de clase asociada a dicha hoja al objeto.

2.1. Árboles de decisión univaluados

Un árbol de decisión univaluado se caracteriza por tener en cada nodo interno sólo un atributo de prueba. Cuando un nodo se expande, el conjunto de entrenamiento es dividido en uno o varios arcos de acuerdo a los valores que pueda tomar ese atributo. La Figura 2.2 muestra un ejemplo de un AD univaluado. Como puede observarse, cada nodo tiene un atributo de prueba. Si el atributo de prueba es numérico, entonces el nodo tiene asociada una prueba de la forma $X \leq V$ y posee dos arcos de salida. El primer arco es para los objetos que tengan en el atributo X un valor menor o igual al valor V; el segundo arco para los objetos que tengan un valor mayor a V. Por otra parte, si un nodo tiene un atributo de prueba no numérico, entonces tiene un número de arcos igual al número de posibles valores del atributo. Finalmente, las hojas tienen asociada una etiqueta de clase.

Existen diversos criterios que permiten seleccionar a un atributo de prueba para la expansión de un nodo en la construcción de un AD. Entre los criterios más usados se encuentran:

X1	X2	X3	Clase
52	65	Sedan	B
43	15	Deportivo	B
40	75	Sedan	B
55	40	Sedan	G
33	100	Sedan	G
25	60	Deportivo	G

Conjunto de Entrenamiento

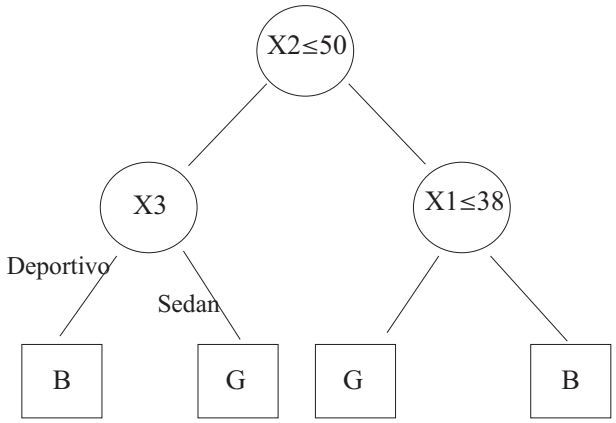


Figura 2.2: Ejemplo de un AD univaluado.

- Ganancia de Información [Sha48, Qui93]. La Ganancia de Información se define a partir de la Entropía. Dada una colección S de objetos, con c clases, la entropía de S se mide como:

$$Entropia(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{2.1}$$

  Donde p_i es la proporción de ejemplos en S que pertenencen a la clase i. Usando la entropía, se define la Ganancia de Información, de un atributo X en un conjunto de datos S, como:

$$Ganancia(S, X) = Entropia(S) - \sum_{v \in Valores(X)} \frac{|S_v|}{|S|} Entropia(S_v) \tag{2.2}$$

  Donde $Valores(X)$ es el conjunto de todos los posibles valores que puede tomar el atributo X y S_v es el subconjunto de S, donde los objetos toman el valor v en el atributo X. El atributo a elegir será aquel que proporcione mayor valor de Ganancia.

- Proporción de la Ganancia de Información [Qui93]. Utiliza la medida anterior (Ganancia(S,X)) pero se aplica una normalización, la cual ajusta la ganancia obtenida por cada atributo. La Proporción de la Ganancia de Información, utilizando 2.2, queda expresada como:

$$Gain(S, X) = Ganancia(S, X) \Big/ \left(- \sum_{v \in Valores(X)} \frac{|S_v|}{|S|} \log_2 \left(\frac{|S_v|}{|S|} \right) \right) \tag{2.3}$$

  El atributo a elegir será aquel que maximice 2.3.

- Índice de diversidad de Gini [KQ02]. Esta medida se calcula para cada dupla atributo-valor. Sea CE_X el conjunto de entrenamiento descrito solamente por el atributo que será evaluado, el cual contiene c clases, el índice de Gini se define como:

$$I_{gini}(CE_X) = 1 - \sum_{j=1}^{c} [p_j(X, v)]^2 \tag{2.4}$$

Donde $p_j(X, v)$ es el número de objetos de la clase j que tienen el valor v en el atributo X dividido entre el número de objetos de la clase j. La dupla atributo-valor elegida para caracterizar a los nodos del árbol, será aquella que obtenga el menor valor con el Índice de Gini.

- Berzal et al. [FBS04] proponen un criterio de selección de un atributo de prueba, en el cual se manejan los atributos numéricos de manera diferente. Los valores de cada atributo numérico son discretizados, de modo que el algoritmo de construcción de ADs los maneje como atributos no numéricos; de esta manera los ADs construidos son más pequeños ya que cada atributo numérico sólo podrá estar una vez en cada rama del árbol. La discretización de los valores se realiza aplicando un algoritmo de agrupamiento y para elegir al atributo de prueba se tienen que evaluar diversos conjuntos de intervalos, lo que hace muy costoso el proceso.

- Un nuevo criterio de selección de un atributo de prueba es propuesto por Chandra y Varghese en [CV09]. Para aplicar este criterio se ordenan los valores de cada atributo numérico. El criterio es evaluado en cada punto medio p entre dos valores distintos de cada atributo. El atributo junto con el punto medio que obtengan el menor valor al evaluarlos con este criterio, serán elegidos como atributo de prueba y punto de división para el nodo a expandir. Este criterio se define como:

$$Criterio(p) = \frac{|S_1|}{|R|} * \left(\frac{C_{S_1}}{C_R}\right) * \left(\sum_{i=1}^{C_{S_1}} \frac{n_i(S_1)}{n_i(R)}\right) + \frac{|S_2|}{|R|} * \left(\frac{C_{S_2}}{C_R}\right) * \left(\sum_{i=1}^{C_{S_2}} \frac{n_i(S_2)}{n_i(R)}\right) \tag{2.5}$$

Donde, R es el conjunto de objetos en el nodo a expandir, S_1 es el conjunto de objetos que tienen en el atributo considerado un valor menor al punto medio p, S_2 es el conjunto de objetos que tienen en el atributo considerado un valor mayor al punto medio p, C_{S_1} es el número de clases en S_1, C_{S_2} es el número de clases en S_2, C_R es el número de clases en R, $n_i(S_1)$ es el número de objetos en $S_1 \subset R$ que tienen clase c_i, $n_i(S_2)$ es el número de objetos en $S_2 \subset R$ que tienen clase c_i, y $n_i(R)$ es el número de objetos en R que tienen clase c_i.

La ecuación 2.5 está diseñada para procesar atributos numéricos, ya que a partir de un punto medio comprueba la homogeneidad de los objetos

presentes en el nodo (con los objetos que tienen un valor menor al punto medio y aquellos con un valor mayor al punto medio, de ahí que la ecuación tiene dos términos). Para extender este criterio y poder procesar atributos no numéricos, se tiene que poner un término en la ecuación por cada valor diferente que tenga el atributo, donde el conjunto S_i en cada término será el conjunto de objetos que tienen el valor i en el atributo en consideración.

2.2. Árboles de decisión multivaluados

Un AD multivaluado se construye de manera similar que un AD univaluado, pero en lugar de usar un atributo de prueba para expandir un nodo se utilizan dos o más atributos. La Figura 2.3 presenta un ejemplo de un AD multivaluado. Como puede observarse, en cada nodo interno están involucrados los dos atributos del conjunto de entrenamiento como atributos de prueba. De cada nodo interno se desprenden dos arcos de salida, cada uno asociado a una de las combinaciones atributo-valor utilizadas en el nodo para particionar a los objetos de entrenamiento. Finalmente, cada hoja del AD tiene asociada una etiqueta de clase, la cual permitirá clasificar nuevos objetos.

Algunos métodos que se usan para la selección de atributos de prueba en los AD multivaluados son:

- Utgoff y Brodley [UB90] proponen un algoritmo que genera AD multivaluados. Este algoritmo usa una unidad LTU (por sus siglas en inglés Linear Threshold Unit) para caracterizar a cada nodo interno. LTU es un conjunto de atributos que permite separar a los objetos del nodo en dos partes, una positiva y una negativa (un arco por cada parte). Cada parte está asociada a una clase del conjunto de entrenamiento, por lo tanto este algoritmo sólo puede trabajar con problemas de dos clases. Además, para encontrar el mejor conjunto de atributos para la LTU, el algoritmo tiene que evaluar cada subconjunto de atributos posible.

- LMDT [BU91] es otro algoritmo que genera ADs multivaluados propuesto por Brodley y Utgoff. Este algoritmo utiliza una función discriminante lineal para representar a cada clase de objetos presente en el nodo a expandir, por lo tanto, para expandir un nodo se crearán tantos arcos de salida como clases en el conjunto de entrenamiento. Cada función discriminante lineal es entrenada con los objetos de la

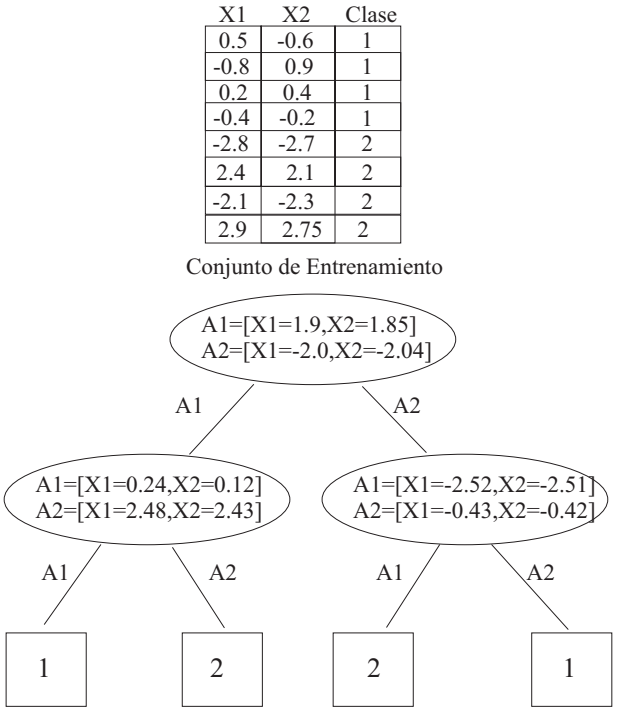


Figura 2.3: Ejemplo de un AD multivaluado.

clase a la que correspondan, hasta encontrar los coeficientes para la función que mejor representen a esa clase. Posteriormente, el algoritmo evalúa si todos los atributos en el nodo contribuyen a la clasificación de los objetos de la clase, si no es así, se eliminan aquellos que no sean necesarios.

- Utgoff y Brodley [UB95] proponen otra manera de construir ADs multivaluados. Cuando un nodo tiene que ser expandido la selección de atributos se hace con un algoritmo de selección de variables (como por ejemplo, la selección secuencial hacia adelante o hacia atrás). Una vez elegidos los atributos de prueba, se aplica un método para la obtención de los valores para cada atributo elegido, los cuales permitirán la partición del conjunto de objetos del nodo (por ejemplo, el método recursivo de mínimos cuadrados o el algoritmo Pocket [Gal86]). Por cada clase se obtendrá un valor para cada atributo, y estas combinaciones de valores representarán a los arcos de salida del nodo, por lo tanto el nodo a expandir tendrá tantos arcos como clases de objetos tenga el nodo.

- Pedrycz y Sosnowski [PS05a] utilizan todo el conjunto de atributos para caracterizar a los nodos internos del AD. Cuando un nodo es expandido, se crea un arco de salida por cada clase de objetos presentes en el nodo y posteriormente se aplica un algoritmo de agrupamiento a los objetos del nodo para dividirlos entre los arcos creados. El número de grupos a crear es igual al número de clases y el conjunto de valores que caracterizan a cada arco está representado por la semilla de cada grupo.

- Uno de los métodos más recientes para la creación de nodos internos multivaluados, es el propuesto por Ouyang et al. [JOS09]. Este método se basa en funciones discriminantes lineales de Fisher para expandir un nodo, creando nodos binarios (con dos arcos) que permiten separar a los objetos en dos clases, sin embargo, este algoritmo permite trabajar con problemas multiclase, creando superclases (una clase contra el resto de las clases unidas).

Estos son algunos ejemplos de métodos utilizados para la selección de subconjuntos de atributos de prueba. Sin embargo, estos métodos aplican técnicas que son muy costosas en tiempo, por la cantidad de procesos que tienen que hacer para seleccionar el subconjunto de atributos de prueba en un nodo a expandir, especialmente si se trabaja con grandes conjuntos de datos.

Capítulo 3

Trabajo relacionado

Actualmente existen diversos algoritmos que permiten generar ADs, por ejemplo: ID3 [Mit97], C4.5 [Qui93, KQ02], CART [BFO84], FACT [VL88], QUEST [SL97], Model Trees [CKJ05], CTC [PMA+07], ID5R [Utg89], ITI [Utg94, UBC97], UFFT [GM05, GMR04], StreamTree [JA03], FDT [Jan98], G-DT [PS05b], SPIDA [WNS07], PT2 [UB90], LMDT [UB95, BU91], GALE [LW04] y C-DT [WSJ06, PS05a]. Sin embargo, estos algoritmos no permiten procesar grandes conjuntos de datos.

Entre los algoritmos que se han desarrollado para trabajar con grandes conjuntos de datos existen algunos que usan estructuras de datos para representar a los atributos del conjunto de entrenamiento (SLIQ [MAR96], SPRINT [SAM96], CLOUDS [ARS98], RainForest [GRG00], RainForest mejorado [NTC07] y BOAI [YWYC08]). Por su parte, otros algoritmos trabajan de manera incremental el conjunto de entrenamiento, con lo cual no es necesario almacenar el conjunto de entrenamiento completo en memoria (BOAT [GGRL99], ICE [YAR99] y VFDT [DH00]).

El resto del capítulo se divide en dos partes, primero se describen a detalle los trabajos relacionados con la línea de investigación de algoritmos que generan árboles de decisión para grandes conjuntos de datos y posteriormente se presenta una discusión de las limitaciones de estos algoritmos.

3.1. Algoritmos que generan ADs para grandes conjuntos de datos

En este apartado se presenta una descripción de los algoritmos de generación de ADs para grandes conjuntos de datos, mostrando las principales

características de cada uno de ellos.

SLIQ [MAR96]

SLIQ (por sus siglas en inglés, Supervised Learning In Quest) es un algoritmo de generación de ADs para grandes conjuntos de datos, propuesto por Mehta et al., que representa el conjunto de entrenamiento por medio de listas, evitando almacenar en memoria todo el conjunto de entrenamiento para construir el AD.

Las listas que utiliza SLIQ almacenan todos los valores de cada atributo que aparecen en el conjunto de entrenamiento. Cada uno de los atributos tendrá una lista que lo describa, y cada lista será de tamaño igual al número de objetos en el conjunto de entrenamiento. Además, se tiene otra lista que almacena la clase a la que pertenece cada objeto, así como el número de nodo en el cual se encuentra el objeto. Se tendrá un total de $A + 1$ listas, donde A es el número de atributos que describen a los objetos.

El criterio de selección del atributo de prueba que SLIQ utiliza es el Índice de Gini y la forma de construcción del AD que sigue SLIQ es similar a la de C4.5 [Qui93]. La figura 3.1 muestra un ejemplo de las listas que genera SLIQ.

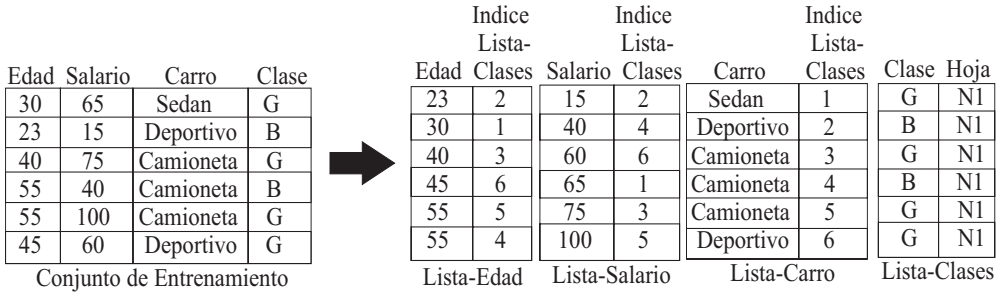


Figura 3.1: Ejemplo de las listas usadas por SLIQ.

Las listas que contienen las descripciones de los atributos pueden ser almacenadas en disco; la única que se mantiene en memoria es la lista que contiene la pertenencia de los objetos a las clases y el nodo en el cual se encuentra cada objeto. Cada vez que un nodo es expandido, esta lista actualiza el número de nodo en que está cada objeto. Sin embargo, si la cantidad de objetos del conjunto de entrenamiento es muy grande, la lista que almacena

la pertenencia a las clases puede ser de gran tamaño, tanto como para no poder ser almacenada en la memoria.

SPRINT [SAM96]

SPRINT (Scalable PaRallelizable INduction of decision Trees) es una mejora del algoritmo SLIQ propuesto por Shafer et al. La diferencia entre estos dos algoritmos radica en la forma en cómo se representan las listas para cada atributo. SPRINT no necesita almacenar en memoria ninguna lista completa, sin embargo, debido a que para expandir cada nodo se deben recorrer las listas almacenadas en disco, el tiempo de procesamiento puede ser demasiado grande, si el conjunto de entrenamiento tiene un gran número de objetos.

SPRINT continúa con la idea de tener una lista por cada atributo, pero ahora en lugar de tener una lista exclusiva para representar las pertenencias de los objetos a las clases, se agrega un valor por cada registro (objeto) a las listas de cada atributo, este valor almacena la clase a la que pertenece cada objeto. Con esta modificación, el espacio para almacenar las listas aumenta, ya que además de tener en su descripción al valor del atributo, contiene la clase del objeto en cada lista. Un ejemplo de estas listas es mostrado en la figura 3.2.

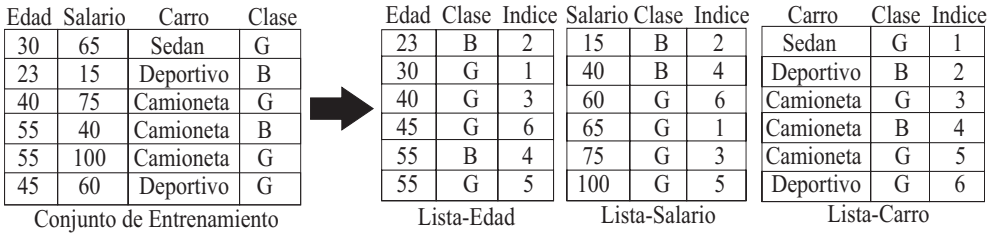


Figura 3.2: Ejemplo de las listas usadas por SPRINT.

Debido a que SPRINT no almacena en memoria ninguna lista completa, porque ninguna es actualizada cuando se expande un nodo, SPRINT necesita procesar todo el conjunto de datos cada vez que un nivel del árbol es expandido.

CLOUDS [ARS98]

CLOUDS es un algoritmo de generación de ADs para grandes conjuntos de datos propuesto por Alsabti et al. Al igual que SLIQ y SPRINT, CLOUDS (Classification for Large or OUt-of-core DataSets) sigue la filosofía de representar el conjunto de entrenamiento en listas para evitar almacenar todo el conjunto en la memoria principal. Las listas son manejadas como lo hace SPRINT. Sin embargo, la ventaja sobre ese algoritmo, es que los valores de los atributos numéricos son divididos en intervalos, y por lo tanto no se hace una revisión sobre todos los posibles valores de esos atributos.

CLOUDS divide el conjunto de valores que puede tomar cada atributo numérico en intervalos, de tal forma que cada intervalo contenga aproximadamente el mismo número de valores. Cuando un nodo va a ser expandido, CLOUDS aplica el índice de Gini a los límites de cada intervalo, aquél con menor valor en el índice de Gini será el atributo elegido para ese nodo. Para encontrar el valor de prueba que tendrá el atributo elegido en el nodo a expandir, CLOUDS estima un valor mínimo global con el índice de Gini para cada intervalo. Este valor mínimo determina si el intervalo es candidato a tener el mejor valor de prueba. Si un intervalo es candidato, entonces se evalúa el índice de Gini para cada uno de los valores que lo forman, lo que significa que, si todos los intervalos llegan a ser candidatos, se requerirá procesar una vez más todo el conjunto de entrenamiento.

El hecho de que CLOUDS trabaje con intervalos en los valores de los atributos numéricos, y no con todos sus valores como lo hace SPRINT, hace al algoritmo más rápido. Sin embargo, tiene el mismo problema que SPRINT y SLIQ, el tamaño de las listas a utilizar en la construcción del árbol puede ser casi el doble de lo que se requiere para almacenar el conjunto de entrenamiento.

RAINFOREST [GRG00]

Debido a que en los algoritmos anteriores las estructuras utilizadas para representar al conjunto de entrenamiento pueden llegar a ocupar hasta el doble de espacio de lo requerido por el conjunto completo, RainForest trata de reducir estas estructuras de manera que puedan ser almacenadas en memoria. Este algoritmo fue propuesto por Gehrke et al.

La idea principal de RainForest es formar conjuntos de listas (denominados *AVC* por Atributo-Valor, Clase) para describir a los objetos del conjunto de entrenamiento. Estas listas contendrán todos los posibles valores

que un atributo pueda tomar, además de su frecuencia en el conjunto de entrenamiento. De esta manera, las listas no serán de gran longitud y con ello podrán ser almacenadas en memoria. Sin embargo, como en cada nodo se tienen que calcular estas listas *AVC*, en algún momento la memoria puede ser insuficiente para almacenarlas. Además, si el conjunto de datos está representado en su mayoría por atributos numéricos descritos por una gran cantidad de valores diferentes, el problema de memoria se incrementa, ya que las listas serán tan grandes como las utilizadas por el algoritmo SPRINT. La figura 3.3 muestra un ejemplo de cómo RainForest representa los atributos mediante listas.

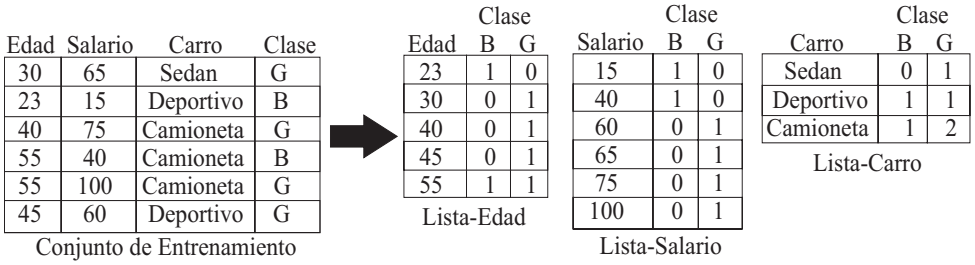


Figura 3.3: Ejemplo de las listas usadas por RainForest.

Para expandir un nodo, RainForest leé una vez el conjunto de entrenamiento para construir la lista *AVC* a utilizar en ese nodo, luego aplica un criterio de selección de atributos (este algoritmo puede utilizar cualquier criterio de selección) y crea un número determinado de hijos para el nodo (de acuerdo al atributo elegido). Por último, RainForest lee otra vez el conjunto de datos, a fin de distribuir los objetos en alguno de los hijos creados. Como se puede observar, para la construcción de cada nivel del AD, este algoritmo tiene que leer dos veces el conjunto de entrenamiento completo, y escribir una vez el conjunto para generar las nuevas listas *AVC*, lo cual es muy costoso, especialmente para grandes conjuntos de datos.

BOAT [GGRL99]

Para evitar almacenar todo el conjunto de entrenamiento en memoria, BOAT (algoritmo propuesto por Gehrke et al.) sólo utiliza una parte de este conjunto para construir el AD, con lo cual no presenta restricciones espaciales. BOAT es un algoritmo incremental capaz de trabajar con grandes conjuntos

de datos y con cualquier criterio de selección de atributo de prueba para construir los nodos del árbol. Lo innovador de este algoritmo es la forma de construir el AD.

Como primer paso obtiene una submuestra D del conjunto de entrenamiento. A esa submuestra se le aplica una técnica conocida como bootstrapping, la cual trabaja de la siguiente manera: a partir de la muestra D, se obtienen submuestras de ella y con cada una se construye un árbol, utilizando un algoritmo tradicional de generación de AD (como C.45 o CART). Cada árbol es recorrido simultáneamente de manera *top-down* (recorrido que empieza en la raíz y termina en una hoja) y para cada nodo recorrido se verifica si el atributo es el mismo en todos los árboles; si es así entonces el árbol final contendrá a ese atributo en ese nodo. En caso de que no sea el mismo atributo, se ignora ese nodo en todos los árboles construidos (borrando los nodos y los subárboles que se desprendan de ellos) y se sigue con el recorrido.

Con los atributos no numéricos se tendrán tantos arcos como valores en su dominio. Con los atributos numéricos se formará un intervalo considerando los valores que tengan en esos nodos los árboles que se construyeron de las submuestras. Una vez que se terminó de construir el árbol a partir de todos los árboles generados por las submuestras, se recorre una vez más todo el conjunto de entrenamiento para terminar de construir el árbol final, ya que en los nodos con atributos de prueba numéricos, se tendrá que calcular el valor del atributo que caracterizará al nodo.

Para poder trabajar de manera incremental, BOAT guarda todos los valores necesarios para poder realizar el cálculo del criterio de selección del atributo, de esta manera no es necesario calcular todo otra vez, sino sólo considerar al nuevo objeto.

El autor asegura que BOAT construye el mismo AD que cualquier método no incremental, tomando en cuenta el criterio de selección del atributo de este último, sin embargo, no demuestra que esto sea posible. Además, obtener la submuestra D requiere de un tiempo adicional, el cual puede ser grande ya que se está trabajando con grandes conjuntos de datos, por otro lado el autor no muestra el procedimiento que se utiliza para formar la submuestra D del conjunto de entrenamiento, con lo cual no se sabe si es, o no, representativa de todo el conjunto, ni se sabe el costo computacional de este procedimiendo.

ICE [YAR99]

ICE (Incrementally Classifying Ever-growing large datasets), propuesto por Yoon et al., procesa el conjunto de entrenamiento por partes, de tal manera que no lo tenga completamente almacenado en memoria. Así, cada parte es procesada individualmente y de cada una de estas partes se obtiene una porción de objetos, que se van acumulando conforme se procesan las partes.

ICE es un modelo que funciona para grandes conjuntos de datos y basa su funcionamiento en un proceso incremental. El procedimiento general de este modelo inicia particionando al conjunto de entrenamiento en subconjuntos de un mismo tamaño. Cada subconjunto será tratado como una época.

En cada época, se construye un AD con la partición dada, aplicando cualquier algoritmo de generación de árboles de decisión (C4.5, CART, etc.), y a partir de ese árbol se genera una submuestra representativa de la partición (ICE genera esta submuestra aplicando alguna técnica de muestreo, como: muestreo aleatorio, agrupamiento local, etc. aunque el autor no especifica cómo son aplicadas estas técnicas).

El procedimiento de ICE se muestra en la figura 3.4, en donde para cada época i, se construye un árbol T_i a partir de la partición D_i, y de cada árbol T_i se extrae una submuestra S_i usando una técnica de muestreo. La unión de S_i con los subconjuntos de muestras de las épocas anteriores se almacenan en U_i. El nuevo conjunto de muestras $U_i = U_{i-1} \cup S_i$ se guarda para construir el árbol de la siguiente época. Para un conjunto de entrenamiento dividido en k épocas, ICE une $S_1, S_2, \ldots, S_k$, los subconjuntos de muestras extraídos de $T_1, T_2, \ldots, T_k$, y construye el AD final C_k con la última muestra U_k, el algoritmo guarda el subconjunto de muestras U_k y el AD C_k. Si una nueva época D_{k+1} debe ser procesada, ICE construye T_{k+1} de D_{k+1}, extrae S_{k+1} de T_{k+1} y usa $U_{k+1} = U_k \cup S_{k+1}$ para construir el nuevo AD, C_{k+1}.

El procedimiento incremental de este algoritmo, no permite que el AD se vaya actualizando si los objetos van llegando de uno en uno. ICE espera a que un cierto número de objetos nuevos se hayan acumulado, así este nuevo conjunto de objetos será tratado como una época más del conjunto de datos. Una desventaja de ICE es que emplea demasiado tiempo en obtener los subconjuntos de muestras de cada época, sobretodo si se trata de grandes conjuntos de datos.

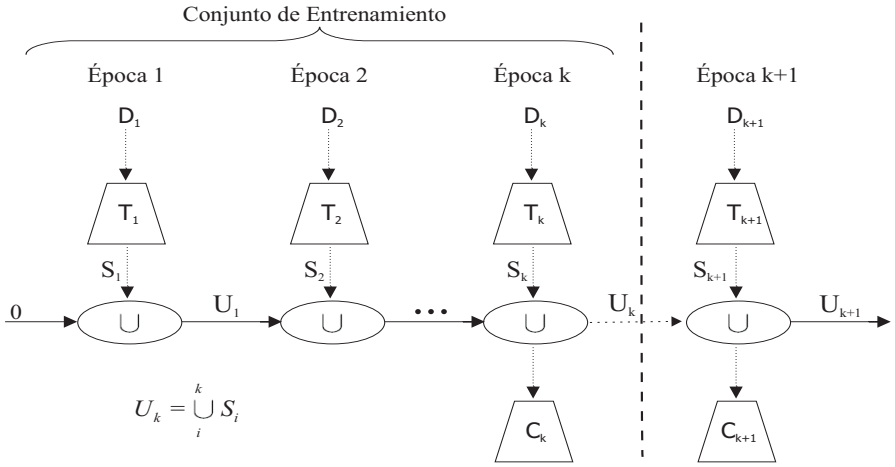


Figura 3.4: Procedimiento general de ICE [YAR99].

VFDT [DH00]

VFDT (Very Fast Decision Trees) es un algoritmo propuesto por Domingos y Hulten que trabaja de incremental, por lo cual no requiere almacenar el conjunto de entrenamiento completo en memoria para construir el AD. Este algoritmo procesa uno a uno los objetos del conjunto de entrenamiento y genera un AD muy similar al que construiría un algoritmo convencional de generación de ADs (como C4.5 o CART).

Con este algoritmo se proponen los árboles Hoeffding, los cuales están diseñados para poder procesar grandes conjuntos de datos. Para construir un AD, VFDT necesita que los objetos del conjunto de entrenamiento estén desordenados, si no lo están, aplica un preprocesamiento sobre los objetos para desordenarlos en forma aletoria. VFDT inicia con un AD construido con un algoritmo convencional de generación de ADs, a partir de un subconjunto de objetos del conjunto de entrenamiento. Posteriomente, VFDT procesa cada objeto del conjunto de entrenamiento recorriendo el AD hasta que llegue a una hoja; cuando el objeto llega a una hoja, el algoritmo mide la ganancia de información de cada atributo. VFDT usa un parámetro n_{min} para indicar el número mínimo de objetos que debe tener una hoja, antes de verificar que la hoja tenga la suficiente información para ser expandida. Cuando n_{min} objetos llegan a una hoja, VFDT obtiene la ganancia de información de cada atributo usando los valores almacenados en la hoja, elige los dos mejores

atributos (aquellos con mayor ganancia) $G(X_a)$ y $G(X_b)$, y obtiene el límite Hoeffding ε usando la Ecuación 3.1:

$$\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n_{min}}} \tag{3.1}$$

En la Ecuación 3.1 R es $log(c)$ (c es el número de clases) y δ es un parámetro del algoritmo, donde $1-\delta$ indica la probabilidad de elegir el mejor atributo de prueba. Si la diferencia entre la ganancia de información de los dos mejores atributos ($\Delta G = |G(X_a) - G(X_b)|$) es mayor que el límite Hoeffding ($\Delta G > \varepsilon$) entonces la hoja es expandida, pero si $\Delta G \leq \varepsilon$, entonces la hoja debe de recibir más objetos antes de ser expandida. VFDT usa n_{min} para evitar checar si $\Delta G > \varepsilon$ cada vez que un objeto llegue a una hoja, lo cual sería un proceso muy costoso para construir el AD, sin embargo si n_{min} objetos no son suficientes, entonces VFDT tiene que revisar con cada objeto extra que llegue a la hoja si ya se puede expandir, lo que resulta en un mayor tiempo de procesamiento . Por otro lado, si $\Delta G = 0$, VFDT usa otro parámetro τ para decidir si el mejor atributo (de acuerdo a la ganancia de información) puede ser elegido como atributo de prueba (si $\varepsilon < \tau$ entonces VFDT elige el mejor atributo como atributo de prueba). VFDT termina cuando todos los objetos del conjunto de entrenamiento han sido procesados.

Este algoritmo utiliza tres parámetros, n_{min}, δ, τ, los cuales pueden ser muy difíciles de determinar por el usuario. Además, para expandir un nodo VFDT tiene que evaluar para cada atributo numérico todos los posibles valores de prueba, lo cual puede ser muy costoso si los atributos poseen una gran cantidad de valores diferentes.

RainForest Mejorado [NTC07]

RainForest Mejorado fue propuesto por Nguyen y Tae-Choong. La principal diferencia entre este algoritmo y RainForest está en el conjunto de listas que representan a los atributos. RainForest genera una lista por cada atributo, la cual se forma con los diferentes valores de ese atributo en el conjunto de entrenamiento, además de la frecuencia que tiene cada valor en cada clase. Este nuevo algoritmo añade el índice de los objetos en los cuales este valor aparece en cada clase, para identificar a qué objetos pertenece cada valor. Esta nueva representación se muestra en la Figura 3.5.

Con estas listas, RainForest Mejorado evita tener que leer dos veces y escribir una vez el conjunto de entrenamiento completo cada vez que un

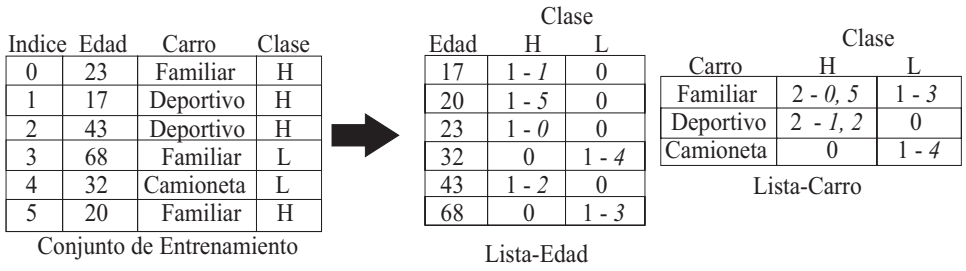


Figura 3.5: Ejemplo de las listas usadas por la mejora de RainForest.

nivel del AD es expandido, ya que las listas nuevas de cada nodo a expandir no son generadas a partir de las descripciones de los objetos, sino de las listas que contiene ese nodo; esto se puede realizar, gracias al índice de los objetos que fue agregado a las listas. El problema con estas listas es su tamaño, ya que está determinado por el número de valores diferentes de cada atributo y el número de objetos en los que aparece cada valor, con lo cual el tamaño de cada lista será proporcional al tamaño del conjunto de entrenamiento.

BOAI [YWYC08]

BOAI (BOttom-up evaluation for ADTree Induction), propuesto por Yang et al., se basa en el algoritmo de generación de ADs llamado ADT [FM99], el cual es un algoritmo para problemas de dos clases. Un AD construido por ADT (ADTree) contiene dos tipos de nodos, nodos de decisión y nodos de predicción. Un ejemplo de este tipo de ADs se muestra en la Figura 3.6. Los árboles ADTrees alternan niveles de nodos de predicción y de decisión. Un nodo de predicción está asociado con un peso, el cual representa su contribución para el conteo final de predicción, y un nodo de decisión tiene un atributo de prueba. ADT asocia un peso a cada objeto del conjunto de entrenamiento [FM99], el cual es usado para elegir el atributo de prueba cuando un nodo de predicción es expandido, así como para calcular los valores de predicción asociados a los nuevos nodos de predicción.

ADT emplea un proceso iterativo para construir el AD, empieza con un nodo raíz de predicción y elige el mejor atributo de prueba para expandir este nodo, con lo cual son creados un nodo de decisión y dos de predicción asociados al nodo raíz. En cada iteración, ADT expande un nodo de predicción,

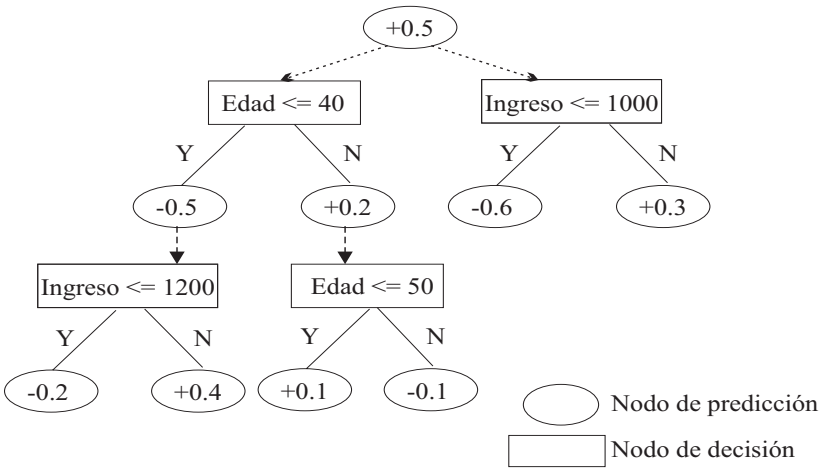


Figura 3.6: Ejemplo de un ADTree generado por el algoritmo BOAI [YWYC08].

recorriendo el AD de manera *top-down* y evaluando todos los posibles valores de prueba de todos los atributos en cada nodo de predicción, para encontrar el nodo que será expandido. Un nuevo nodo de decisión puede ser ligado a cualquier nodo de predicción existente, no sólo en las hojas. ADT termina cuando el número de iteraciones (definidas por el usuario) es alcanzado. La clasificación de un nuevo objeto en un ADTree depende del signo de la suma de los valores de predicción que existen en todos los caminos que pueden ser recorridos con el objeto a clasificar.

BOAI es un algoritmo que propone una manera alternativa de construir árboles ADTree para grandes conjuntos de datos. Para cada atributo, BOAI crea una lista para almacenar los pesos asociados a los objetos del conjunto de entrenamiento; estas listas son almacenadas en los nodos de predicción y son usadas para elegir al atributo de prueba cuando un nodo de predicción es expandido. A diferencia de ADT, BOAI recorre el AD de manera *bottom-up* para elegir el nodo que será expandido en cada iteración. Este tipo de recorrido, permite a BOAI mantener almacenadas en memoria sólo las listas de los últimos nodos de predicción (los que son hojas), las cuales sirven para generar las listas de los nodos de predicción que están en niveles superiores. Además, BOAI ordena los valores de cada atributo numérico, con lo cual evita tener que ordenar estos valores cada vez que un nodo sea expandido. Una desventaja de BOAI es el tamaño de las listas, ya que éstas pueder

ser demasiado grandes si los atributos poseen una gran cantidad de valores (lo cual suele ocurrir en los atributos numéricos cuando se trata de grandes conjuntos de datos).

3.2. Discusión

A partir de la descripción de algoritmos que generan árboles de decisión para grandes conjuntos de datos se puede notar que estos algoritmos tienen ciertas restricciones.

SLIQ, SPRINT, CLOUDS, RainForest y BOAI son algoritmos que procesan todo el conjunto de entrenamiento, pero recorren el conjunto de entrenamiento más de una vez para generar el AD, por ejemplo, RainForest recorre tres veces al conjunto para expandir cada nivel del AD. Además, estos algoritmos, incluyendo también a RainForest Mejorado utilizan estructuras de datos para cada atributo del conjunto de entrenamiento, las cuales pueden llegar a ser muy grandes si los atributos tienen una gran cantidad de valores diferentes; por ejemplo, si el conjunto de entrenamiento tiene millones de objetos, en algunos casos, cada lista tendrá millones de registros, por lo que para algunos problemas prácticos la memoria puede ser insuficiente.

Por su parte, BOAT e ICE no procesan el conjunto de entrenamiento completo para construir el AD; estos algoritmos obtienen un subconjunto de objetos para generar el AD, lo cual puede ser muy costoso en tiempo, principalmente si se trabaja con grandes conjuntos de datos.

Por otro lado, VFDT utiliza tres párametros que tienen que ser definidos por el usuario, los cuales pueden ser difíciles de determinar, además en cada expansión de un nodo se tienen que evaluar todos los posibles valores de prueba de todos los atributos para encontrar el mejor, lo cual puede ser un proceso costoso en tiempo si son muchos atributos y si cada uno tiene una gran cantidad de valores.

Existen otros algoritmos que aseguran poder trabajar con grandes conjuntos de datos, como por ejemplo NT [JZ06] y PUDT [TD07], sin embargo, estos algoritmos no demuestran que esto sea posible. NT presenta una nueva heurística que puede ser aplicada para elegir el atributo de prueba en un nodo a expandir, pero para poder generar el AD tiene que mantener en memoria a todo el conjunto de entrenamiento. Por su parte, PUDT sólo presenta la paralelización de dos algoritmos convencionales de generación de ADs, C4.5 y ADs basados en discriminantes lineales, la cual es una línea de investigación no abordada en este trabajo.

Como se puede observar la mayoría de los algoritmos del estado del arte tienen problemas de espacio, ya que algunos tienen que mantener en memoria el conjunto de entrenamiento completo o usan una representación para los atributos que requiere aún más espacio de almacenamiento que el conjunto de entrenamiento completo. Por otro lado, los algoritmos que no tienen problemas de espacio: (a) basan la construcción del AD sólo en un subconjunto de objetos de entrenamiento, pero para obtener ese subconjunto requieren tiempo de procesamiento adicional, lo cual puede ser muy costoso para grandes conjuntos de datos; o bien (b) usan diversos parámetros, que pueden ser muy difíciles de especificar por el usuario. Por lo anterior, el objetivo de esta tesis es proponer nuevos algoritmos, que generen ADs tanto univaluados como multivaluados, a partir de grandes conjuntos de datos, los cuales tomen en cuenta todo el conjunto de entrenamiento para construir el AD y procesen el conjunto de entrenamiento sólo una vez.

Capítulo 4

ADs multivaluados utilizando todo el conjunto de atributos

En este capítulo se presenta el algoritmo IIMDT (por sus siglas en inglés, *Incremental Induction of Multivariate Decision Trees*) el cual construye ADs multivaluados para grandes conjuntos de datos. En el Capítulo 3 se analizaron las limitaciones de los algoritmos existentes de generación de AD para grandes conjuntos de datos. IIMDT resuelve algunas de estas limitaciones, ya que procesa todo el conjunto de entrenamiento sin almacenarlo completo en memoria, reduciendo así los requerimientos de memoria, con lo cual es capaz de procesar conjuntos de datos más grandes. Para poder procesar todo el conjunto de entrenamiento y evitar almacenarlo todo en memoria, se propone que IIMDT procese uno a uno los objetos de entrenamiento (actualizando el AD con cada uno de ellos) y utilice un pequeño número de objetos para expandir un nodo, de tal manera que la expansión tome poco tiempo. Además, los objetos utilizados para expandir un nodo serán eliminados una vez que el nodo sea expandido, por lo que no es necesario mantener en memoria a todo el conjunto de entrenamiento, durante el proceso de construcción del AD. Para generar el AD multivaluado, se propone usar todo el conjunto de atributos en los nodos internos del árbol, con lo cual no se gasta tiempo en elegir atributos de prueba al expandir nodos.

La estructura del AD construido será similar a la de los construidos por algoritmos de generación de AD tradicionales. El AD construido tendrá un nodo raíz, nodos internos y nodos terminales (hojas). En nuestro caso, cada nodo interno estará caracterizado por todo el conjunto de atributos y tendrá un conjunto de arcos de salida (cada arco con una combinación de valores para los atributos de prueba). Por su parte, a los nodos terminales se

les asociará una etiqueta de clase.

La etapa de construcción del AD consiste en recorrer el AD con cada objeto de entrenamiento. El objeto empieza en la raíz y desciende por los nodos internos hasta que llegue a una hoja, en donde será almacenado (para más detalles del recorrido del árbol, ver la Sección 4.2). Cuando un nodo (hoja) tenga almacenados s objetos (s es un parámetro de nuestro algoritmo) será expandido. En la etapa de clasificación se recorre el AD con cada objeto de prueba hasta que llegue a una hoja, en donde se le asignará la etiqueta de clase asociada a la hoja.

4.1. Construcción del AD

Para construir el AD, se propone procesar de manera incremental los objetos del conjunto de entrenamiento. La construcción se inicia con un nodo raíz vacío (que al inicio del algoritmo es una hoja), y procesa uno a uno los objetos de entrenamiento. Cada objeto recorre el AD hasta que llegue a una hoja en donde será almacenado. Cuando una hoja tenga s objetos almacenados, si la hoja tiene objetos de 2 o más clases, la hoja será expandida, pero si tiene objetos de una sola clase, sólo se actualizará. Se propone usar solamente s objetos para expandir o actualizar una hoja, ya que, almacenar pocos objetos y eliminarlos después de usarlos para expandir o actualizar, permitirá procesar grandes conjuntos de datos. Sin embargo, asignar el valor al parámetro s puede no ser tarea fácil para el usuario, es por esto que en la Sección 4.5 se hace un estudio experimental del comportamiento de nuestro algoritmo con respecto a este parámetro.

Debido a que se procesarán los objetos del conjunto de entrenamiento de manera incremental (uno a uno), para evitar que nuestro algoritmo sólo procese objetos de una sola clase y por lo tanto sólo actualice los nodos y no los expanda, lo cual puede llevar a construir ADs sesgados, se propone un preprocesamiento de los objetos del conjunto de entrenamiento antes de iniciar la construcción del AD. Este preprocesamiento consiste en alternar los objetos de cada clase, el primer objeto será de la clase 1, el segundo de la clase 2 y así sucesivamente, si exiten c clases, entonces el objeto $c + 1$ será nuevamente de la clase 1, el $c + 2$ de la clase 2 y así sucesivamente. Esta reorganización de los objetos de entrenamiento se hace en el *Paso 1* del algoritmo IIMDT (ver Figura 4.3).

Para expandir un nodo, cuando éste tiene almacenados objetos de 2 o más clases, se propone crear un arco de salida por cada clase presente en

el nodo (esta idea es utilizada en el algoritmo C-DT [PS05a]). Cada arco estará ligado a un nuevo nodo vacío y sin descendientes (una hoja). El nodo a expandir se convierte en un nodo interno, el cual estará caracterizado por todo el conjunto de atributos y tendrá una combinación de valores asociada a cada arco de salida. Los valores asociados a cada arco se obtendrán de los objetos que pertenecen a la clase que está asociada al arco de salida. Para cada arco, se calcula para cada atributo, la media entre los valores de los objetos que pertenecen a la clase que está asociada al arco. Una vez que estos valores son obtenidos, se eliminan los objetos que están almacenados en el nodo que se está expandiendo. Los objetos pueden ser eliminados ya que los valores obtenidos son representativos de estos objetos, además de esta manera estos objetos sólo se procesarán una vez durante la construcción del AD y se liberará espacio en la memoria para poder procesar más objetos. La Figura 4.1 muestra el proceso de expansión de un nodo.

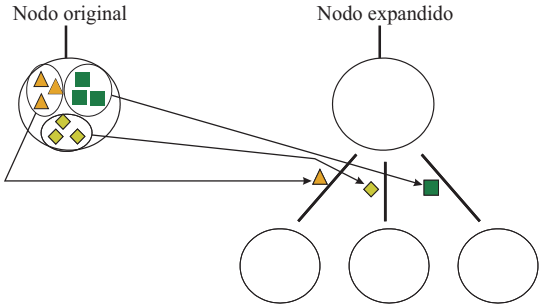


Figura 4.1: Ejemplo de expansión de un nodo para el algoritmo IIMDT.

De la Figura 4.1 se puede observar que para cada clase de objetos en el nodo se crea un arco (en el ejemplo son 3 arcos porque son 3 clases). Para cada arco, se toman en cuenta los objetos de la clase asociada al arco para obtener la combinación de valores que representará a estos objetos. Nótese que, después de la expansión, los s objetos son eliminados, quedando vacío tanto el nodo que se expandió como cada nuevo nodo.

Por otro lado, cuando un nodo tiene s objetos, todos de la misma clase, no se expande el nodo sólo se actualiza, debido a que el nodo ya es homogéneo. Para actualizar un nodo se calcula para cada atributo la media entre los valores de los objetos almacenados en el nodo. Estos valores son promediados con los valores asociados con el arco de entrada del nodo, para obtener nuevos valores para los atributos de prueba, los cuales serán asociados al arco de

entrada. Esto se hace para tener una nueva combinación de valores asociada al arco de entrada del nodo, la cual caracterize tanto a los objetos con lo cuales se había obtenido la combinación de valores anterior, como a los objetos que están almacenados ahora en el nodo. Una vez que se ha obtenido la nueva combinación de valores para el arco de entrada, se eliminan los objetos almacenados en el nodo y el nodo queda vacío. Con este proceso se conserva el nodo como una hoja. La Figura 4.2 muestra el proceso de actualización de un nodo, cuando todos los objetos son de la misma clase y se han almacenado s objetos en el nodo.

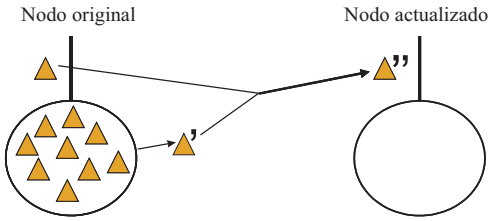


Figura 4.2: Ejemplo de actualización de un nodo para el algoritmo IIMDT.

En la Figura 4.2 se puede notar que de los s objetos almacenados en el nodo, se obtiene una combinación de valores, la cual es promediada con la combinación de valores que tiene el arco de entrada de la hoja. La nueva combinación de valores se asocia al arco de entrada. Finalmente, la hoja queda vacía.

Por último, una vez que todos los objetos del conjunto de entrenamiento han sido procesados, se asigna a cada hoja del AD construido, la clase mayoritaria de los objetos que quedaron almacenados en la hoja. Si la hoja está vacía, entonces se le asigna la clase de la cual se obtuvo la combinación de valores que está asociada al arco de entrada de la hoja.

La Figura 4.3 muestra el algoritmo IIMDT y la Figura 4.4 muestra la función utilizada para procesar un nodo cuando tiene almacenados s objetos. Como se puede observar en la Figura 4.3 el proceso de construcción de un AD con el algoritmo IIMDT es muy similar al de un algoritmo tradicional de inducción de ADs. Para cada objeto en el conjunto de entrenamiento se ejecuta un proceso recursivo para recorrer el AD. El algoritmo expande una hoja cuando tiene s objetos de 2 o más clases almacenados en ella (los cuales serán eliminados después de la expansión), lo que permite que el proceso

de expansión en IIMDT sea rápido y que se almacenen pocos objetos en la memoria.

```
IIMDT
   Entrada:
      CE: Conjunto de entrenamiento
      s: Número máximo de objetos almacenados en un nodo
   Salida:
      AD: Árbol de decisión
   Paso 1: Reorganizar el CE
   Paso 2: Crear nodo raíz R
   Paso 3: Para cada objeto O en CE, hacer
                ActualizaAD(O,R)
           Fin Para
   Paso 4: Asignar a cada hoja la clase mayoritaria de los objetos en ella
Fin IIMDT


ActualizaAD(O, NODO)
   Si número de objetos almacenados en NODO < s, entonces
      Almacenar objeto O en NODO
      Incrementar el número de objetos en NODO
      Si número de objetos almacenados en NODO = s, entonces
         ProcesaNodo(NODO)
         Incrementar el número de objetos en NODO
      Fin Si
   Sino /* número de objetos almacenados en NODO > s */
      Para cada arco Rj en NODO, hacer
         Obtener la diferencia Dj entre los valores de O y los del Rj
      Fin Para
      Hijo = Nodo ligado al arco con la menor diferencia D
      ActualizaAD(O, Hijo)
   Fin Si
Fin ActualizaAD
```

Figura 4.3: Algoritmo IIMDT.

En la Figura 4.4 se puede observar el proceso que sigue IIMDT cuando un nodo tiene almacenados s objetos. Nuestro algoritmo sigue una de las dos opciones, expande el nodo o sólo lo actualiza, dependiendo del número de clases que tengan los objetos almacenados en el nodo.

```
ProcesaNodo(NODO)
    Si NODO tiene objetos de más de una clase, entonces
        Para cada clase Ci en NODO, hacer
            Crear un arco Ri
            Para cada atributo Aj del CE, hacer
                Obtener la media Mij entre los valores de los objetos en NODO de clase Ci
                Asignar la media Mij al arco Ri
            Fin Para
            Crear nodo asociado a Ri
        Fin Para
        Eliminar los objetos almacenados en el nodo expandido
    Sino
        Para cada atributo Ai del CE, hacer
            Obtener la media Mi entre los valores de los objetos en NODO
            Promediar Mi con la media del atributo Ai asociada al arco de entrada de NODO
        Fin Para
        Eliminar los objetos almacenados en el nodo actualizado
    Fin Si
Fin ProcesaNodo
```

Figura 4.4: Procesamiento de un nodo con s objetos con el algoritmo IIMDT.

4.2. Recorrido del AD

El recorrido de un objeto O en un AD construido por IIMDT será de la siguiente manera. O inicia en el nodo raíz y desciende a un nodo hijo pasando por alguno de los arcos de salida asociados al nodo raíz. Para decidir por cuál arco descender, se busca la combinación de valores de los arcos que mejor represente a los valores de O. Para elegir esa combinación se calcula la suma de los valores absolutos de las diferencias entre O y la combinación de valores de cada arco, seleccionando aquella con el menor valor. El recorrido continúa de la misma forma con los nodos hijos del siguiente nivel hasta llegar a los nodos hoja.

Este recorrido es utilizado por el algoritmo IIMDT tanto para construir el AD con el conjunto de entrenamiento, como para clasificar un objeto nuevo. En el caso del recorrido para construir el AD, cuando el objeto llega a una hoja IIMDT almacena el objeto en la hoja; en el caso de la clasificación de un nuevo objeto, se le asigna al objeto la etiqueta de clase asociada a la hoja. El recorrido de un objeto en un AD se puede observar en la Figura 4.3.

4.3. Análisis de complejidad temporal del algoritmo IIMDT

En la etapa de construcción de un AD, IIMDT realiza dos pasos fundamentales, recorrer el AD con los objetos de entrenamiento y expandir nodos que tienen almacenados s objetos, por lo cual se analiza la complejidad de estos dos pasos.

Para un conjunto de entrenamiento de m objetos, divididos en k clases, la complejidad de recorrer el AD con cada objeto depende del número de niveles en el AD. En un AD, en el peor caso, se tiene como máximo $O(m/s)$ niveles, dado que IIMDT utiliza s objetos para cada expansión, y cada expansión genera un nuevo nivel del AD. Por lo tanto, recorrer el AD con los m objetos es $O(m * m/s) = O(m^2)$ en el peor caso.

En el proceso de expansión de un nodo, IIMDT crea a lo más k arcos. Para cada arco, un subconjunto de los s objetos almacenados en el nodo es usado para calcular la media de cada atributo, dado que se utilizan sólo los objetos de la clase asociada al arco para hacer este procedimiento. Estas medias serán asociadas como valores representativos en el arco. Expandir un nodo toma $O(k * (s/k)) = O(s)$ y el número máximo de expansiones que puede hacer IIMDT es $O(m/s)$, entonces el proceso completo de expansiones

que hace IIMDT es $O(s * (m/s) = O(m)$.

Finalmente, la complejidad del proceso de construcción de un AD utilizando el algoritmo IIMDT es la suma de recorrer el AD con todos los objetos del conjunto de entrenamiento y la de hacer la expansión de todos los nodos que tengan s objetos almacenados, quedando:

$$O(m^2 + m) = O(m^2); \text{ en el peor caso.}$$

La complejidad de IIMDT es superior a la complejidad de algunos algoritmos de generación de ADs para grandes conjuntos de datos, como ICE [YAR99] y BOAI [YWYC08], e igual a la complejidad de otros algoritmos, como VFDT [ZLC07]. Sin embargo, la complejidad de estos algoritmos depende de la forma de los datos, no sólo del tamaño del conjunto de entrenamiento. Por esta razón, en la Sección 4.5 se mostrará un análisis experimental del tiempo empleado por nuestro algoritmo y otros algoritmos que construyen ADs para grandes conjuntos de datos, con la finalidad de mostrar la diferencia entre ICE, VFDT, BOAI e IIMDT (el costo efectivo para construir un AD, es decir el tiempo de procesamiento real que emplean los algoritmos para construir el AD).

4.4. Análisis de complejidad espacial del algoritmo IIMDT

El análisis de complejidad espacial se basa en el espacio que IIMDT requiere para construir un AD. IIMDT sólo tiene que tener almacenado en memoria el objeto que esté procesando al momento y el AD construido con los objetos anteriores.

En el AD construido por IIMDT, cada arco de un nodo tiene asociado sólo un objeto (una combinación de valores obtenida de los objetos que se almacenaron en ese nodo antes de expandirlo). En el peor de los casos, después de expandir un nodo, éste tiene k arcos y el número máximo de expansiones que puede hacer IIMDT es m/s. El número de objetos almacenados en el AD en el peor caso es:

$$O(k * (m/s))$$

Pero para grandes conjuntos de datos, es común que $k \ll m$ y, en nuestro algoritmo, $s \ll m$ entonces podemos considerar:

$$O(k * (m/s)) = O(m)$$

En el peor caso, la complejidad en espacio de IIMDT es igual a la complejidad de otros algoritmos, como ICE [YAR99], VFDT [ZLC07] y BOAI [YWYC08]. Sin embargo, el espacio requerido por nuestro algoritmo puede reducirse de acuerdo al valor que se le asigne al parámetro s, por ejemplo, si a s le damos un valor de 1000, entonces el espacio será proporcional a la milesima parte de m.

4.5. Resultados experimentales

Los experimentos mostrados en esta sección se realizaron con diversos objetivos. Primero se realiza un estudio experimental del parámetro s para analizar el comportamiento de IIMDT cuando el valor del parámetro varía. También se realizaron experimentos variando el número de objetos y de atributos en el conjunto de entrenamiento, para analizar cómo se afecta el desempeño de IIMDT con estas variaciones, en estos experimentos se incluye una comparación contra los algoritmos de generación de ADs para grandes conjuntos de datos más recientes, ICE [YAR99], VFDT [DH00] y BOAI [YWYC08]. Por último, debido a que la complejidad espacial de IIMDT, analizada en la Sección 4.4, es del mismo orden que la complejidad de los algoritmos contra los que se compara, se realizaron experimentos que muestran una comparación del uso de memoria en la construcción del AD.

En todos los experimentos se evaluó el tiempo de procesamiento (incluyendo el tiempo de construcción del AD y el tiempo de clasificación, además para IIMDT se incluyó también el tiempo de preprocesamiento del conjunto de entrenamiento). También se evaluó la calidad de clasificación.

Para todos los experimentos se usó validación cruzada de 10 pliegues y en las gráficas de resultados se incluyen los intervalos de confianza del 95 %. Sin embargo, en algunos casos los intervalos de confianza no son visibles en las gráficas ya que son muy pequeños.[1]

Todos los experimentos fueron realizados en una computadora Pentium 4 a 3.06 GHz, con 2 GB de memoria RAM, corriendo Linux Kubuntu 7.10. Para programar nuestro algoritmo se utilizó el lenguaje de programación C.

[1]Adicionalmente, en el Apéndice A se muestra una evaluación de la significancia estadística entre nuestros algoritmos y los utilizados para compararlo.

4.5.1 Conjuntos de datos

Los conjuntos de datos utilizados para realizar los experimentos se describen en la Tabla 4.1. Los tres primeros son conjuntos de datos reales. Poker fue obtenido del repositorio de datos UCI [AN07] y contiene posibles combinaciones de cartas en un juego de poker, se eligió este conjunto ya que es de los que más objetos tienen en el repositorio UCI. SpecObj y GalStar se obtuvieron del repositorio Sloan Digital Sky Survey [JAM08]. SpecObj contiene 5 parámetros de datos observacionales espectroscópicos obtenidos de diversos objetos en el universo, por ejemplo de galaxias, cuásares cercanos y lejanos a nuestra galaxia, estrellas. GalStar contiene 30 parámetros de datos observacionales obtenidos con la técnica de fotometría de galaxias y estrellas. El resto de los conjuntos de datos utilizados son conjuntos sintéticos generados aleatoriamente siguiendo una distribución normal con medias (M) y desviaciones estándar (DE) diferentes para cada clase. Se eligió el número de atributos en estos conjuntos sintéticos para probar con conjuntos de datos con pocos atributos y también con un mayor número de atributos.

Tabla 4.1: Conjuntos de datos usados en los experimentos

Conjunto	# Clases	# Objetos	# Atributos
Poker	2	923,707	10
SpecObj	6	884,054	5
GalStar	2	4,000,000	30
Sintética_1	2	4,000,000	5
Sintética_2	3	4,000,000	5
Sintética_3	5	4,000,000	5
Sintética_4	2	4,000,000	40
Sintética_5	3	4,000,000	40
Sintética_6	5	4,000,000	40

Los valores utilizados para generar los conjuntos de datos sintéticos 1, 2 y 3 de la Tabla 4.1 se encuentran descritos en la Tabla 4.2.

Para generar los conjuntos de datos sintéticos 4, 5 y 6 de la Tabla 4.1 se tomaron en cuenta los valores de la Tabla 4.2. Para cada clase, se tomaron los dos primeros valores de M y DE, los cuales fueron repetidos hasta completar el número de atributos deseado, en este caso, como se generaron conjuntos de datos de 40 atributos, los valores fueron repetidos 20 veces.

Tabla 4.2: Medias y desviaciones estándar para los conjuntos de datos sintéticos

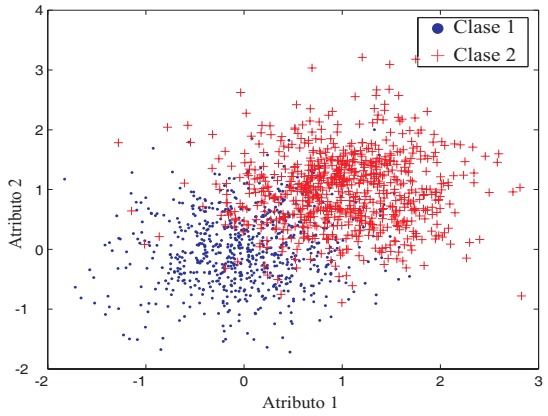
2-clases	$M_1 = [0,0,0,0,0]$ $M_2 = [1,1,1,1,1]$	$DE_1 = [0{,}6, 0{,}6, 0{,}6, 0{,}6, 0{,}6]$ $DE_2 = [0{,}6, 0{,}6, 0{,}6, 0{,}6, 0{,}6]$
3-clases	$M_1 = [0{,}2, 0{,}5, 0{,}2, 0{,}5, 0{,}2]$ $M_2 = [0{,}3, 0{,}9, 0{,}3, 0{,}9, 0{,}3]$ $M_3 = [0{,}7, 0{,}9, 0{,}7, 0{,}9, 0{,}7]$	$DE_1 = [0{,}15, 0{,}15, 0{,}15, 0{,}15, 0{,}15]$ $DE_2 = [0{,}15, 0{,}15, 0{,}15, 0{,}15, 0{,}15]$ $DE_3 = [0{,}10, 0{,}10, 0{,}10, 0{,}10, 0{,}10]$
5-clases	$M_1 = [0{,}2, 0{,}5, 0{,}2, 0{,}5, 0{,}2]$ $M_2 = [0{,}3, 0{,}9, 0{,}3, 0{,}9, 0{,}3]$ $M_3 = [0{,}7, 0{,}9, 0{,}7, 0{,}9, 0{,}7]$ $M_4 = [0{,}9, 0{,}4, 0{,}9, 0{,}4, 0{,}9]$ $M_5 = [0{,}5, 0{,}0, 0{,}5, 0{,}0, 0{,}5]$	$DE_1 = [0{,}15, 0{,}15, 0{,}15, 0{,}15, 0{,}15]$ $DE_2 = [0{,}15, 0{,}15, 0{,}15, 0{,}15, 0{,}15]$ $DE_3 = [0{,}10, 0{,}10, 0{,}10, 0{,}10, 0{,}10]$ $DE_4 = [0{,}15, 0{,}15, 0{,}15, 0{,}15, 0{,}15]$ $DE_5 = [0{,}10, 0{,}10, 0{,}10, 0{,}10, 0{,}10]$

Para mostrar la distribución de los conjuntos de datos sintéticos generados, se crearon conjuntos de datos con los primeros 2 atributos de la Tabla 4.2. Estos conjuntos se muestran en la Figura 4.5.

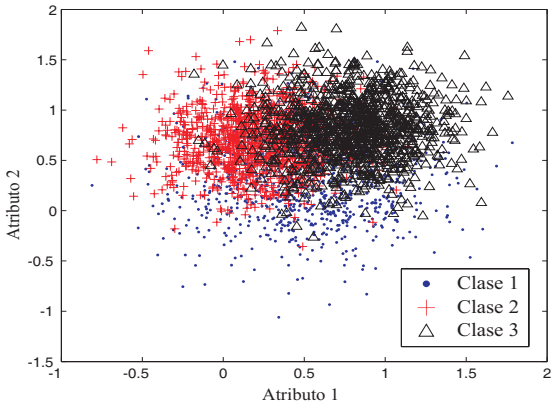
Los conjuntos de datos sintéticos y el conjunto GalStar, son conjuntos de datos balanceados (poseen el mismo número de objetos en cada clase). Por su parte, Poker y SpecObj son conjuntos de datos desbalanceados. Poker tiene el 54.3 % de objetos en la clase 1 y el 45.7 % en la clase 2. SpecObj tiene 1.15 % de objetos en clase 1, 8.25 % en clase 2, 76.15 % en clase 3, 9.21 % en clase 4, 0.72 % en clase 5 y 4.52 % en clase 6.

4.5.2 Algoritmos utilizados para comparar el desempeño de IIMDT

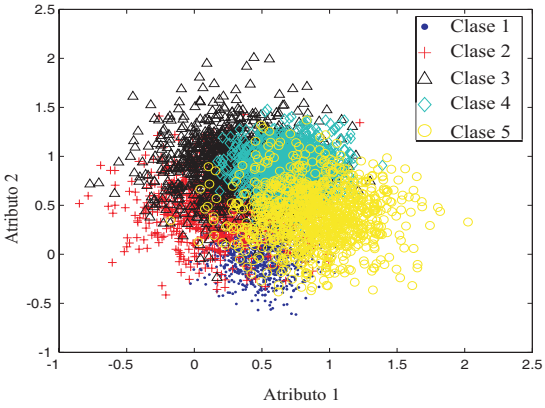
Se utilizaron los tres algoritmos más recientes de generación de AD para grandes conjuntos de datos para comparar el desempeño de nuestro algoritmo. Los algoritmos usados para la comparación son: ICE, implementado por nosotros basándonos en el trabajo de Yoon et al. [YAR99]. Para utilizar este algoritmo, para construir el AD, se estableció el número de epocas en 5 y el tamaño de las submuestras en 10 % del conjunto de entrenamiento por cada época (estos valores fueron establecidos de acuerdo a los experimentos presentados en [YAR99]). El segundo algoritmo utilizado fue VFDT [DH00]. Para este algoritmo obtuvimos la implementación del autor y los experimentos se ejecutaron con los valores predeterminados del programa, que son los sugeridos por los autores en [DH00], estos valores son $n_{min} = 300$, $\delta = 0{,}000001$ y $\tau = 0{,}05$. El últimoalgoritmo usado es BOAI [YWYC08], del cual también se utilizó la implementación del autor. Para este algoritmo,



(a) 2 clases



(b) 3 clases



(c) 5 clases

Figura 4.5: Conjuntos de datos sintéticos utilizando los 2 primeros atributos y 2, 3 y 5 clases.

se estableció en 30 el número de iteraciones a realizar para construir el AD (este valor se eligió de acuerdo a los experimentos mostrados en [YWYC08]).

4.5.3 Parámetro s del algoritmo IIMDT

Debido a que se trabaja con grandes conjuntos de datos, es común que en este tipo de conjuntos exista mucha redundancia en los datos, por lo tanto se propone que nuestro algoritmo sólo utilice un subconjunto de objetos de entrenamiento para hacer más rápido el proceso de expansión de un nodo. Este subconjunto será de tamaño s y para observar el desempeño de nuestro algoritmo cuando el valor del paramétro s varía, se realizó una serie de experimentos con los conjuntos de datos reales descritos en la tabla 4.1. Se tomaron diversos valores para s, 50 y de 100 a 600 con incrementos de 100. Para cada conjunto de datos se crearon un conjunto de entrenamiento y un conjunto de prueba de 100,000 objetos cada uno. El tiempo de procesamiento y la calidad de clasificación obtenidas por IIMDT se muestran en la Figura 4.6.

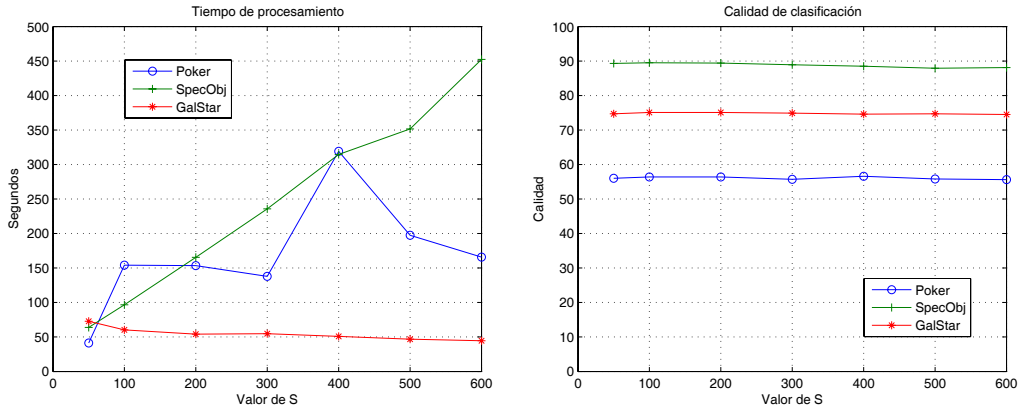


Figura 4.6: Tiempo de procesamiento y calidad de clasificación para IIMDT variando s.

Como se puede observar en la Figura 4.6, la calidad de clasificación, en cada conjunto de datos, es muy similar, no importando el valor que tome el parámetro s. Por otra parte, el tiempo de procesamiento empleado por IIMDT, en GalStar no varía mucho cuando s cambia su valor, en Poker el tiempo varía un poco pero no se incrementa mucho. En SpecObj el tiempo de

procesamiento se incrementa conforme s incrementa su valor, este comportamiento se debe a que SpecObj tiene más clases que Poker y que GalStar, por lo que al recorrer el AD con un objeto, tiene que evaluar más caminos para descender de un nivel a otro (dado que el número de arcos de salida de un nodo está determinado por el número de clases en los objetos del nodo). De esto se puede observar que es mejor elegir un valor pequeño de s, ya que el tiempo de procesamiento es menor que con un valor grande y la calidad de clasificación es similar para todos los valores.

Para realizar los siguientes experimentos con IIMDT, se eligió el valor de $s = 100$, ya que con este valor se mantiene un buen balance entre tiempo y calidad. No se eligió $s = 50$, porque con este valor los ADs construidos eran más grandes que con $s = 100$.

4.5.4 Incrementando el número de objetos en el conjunto de entrenamiento

En esta sección se muestra el desempeño de IIMDT y una comparación contra otros algoritmos de generación de ADs para grandes conjuntos de datos, cuando se incrementa el número de objetos de entrenamiento. Primero se muestran los resultados obtenidos con los conjuntos de datos reales descritos en la Tabla 4.1 y posteriormente los resultados obtenidos con los conjuntos de datos sintéticos.

4.5.4.1 Conjuntos de datos reales

Para mostrar el desempeño de IIMDT en conjuntos de datos reales, se realizaron experimentos con 3 conjuntos con diferente número de clases, de objetos y de atributos (ver Tabla 4.1). Con estos conjuntos de datos, se realizaron experimentos creando diversos conjuntos de entrenamiento, para evaluar el desempeño de IIMDT cuando se incrementa el número de objetos en el conjunto de entrenamiento.

Para Poker se crearon conjuntos de entrenamiento de 50,000 a 500,000 objetos, con incrementos de 50,000. En la Figura 4.7 se pueden observar los resultados obtenidos por IIMDT, ICE, VFDT y BOAI. Como puede observarse, IIMDT y VFDT son mejores que ICE y BOAI, ya que aunque los cuatro algoritmos obtienen una calidad de clasificación similar, IIMDT y VFDT emplean menor tiempo de procesamiento para la construcción del AD y la clasificación de los objetos de prueba. Además, los resultados de ICE y BOAI muestran intervalos de confianza mayores que los de IIMDT y VFDT, lo que

sugiere que el comportamiento de ICE y BOAI no es regular cuando se hace validación cruzada con este conjunto de datos . Para observar la diferencia entre IIMDT y VFDT, se graficaron sólo los resultados de estos algoritmos en la Figura 4.8. Para Poker VFDT es mejor, ya que obtuvo el menor tiempo de procesamiento. En cuanto a la calidad de clasificación, ambos algoritmos obtuvieron una calidad similar.

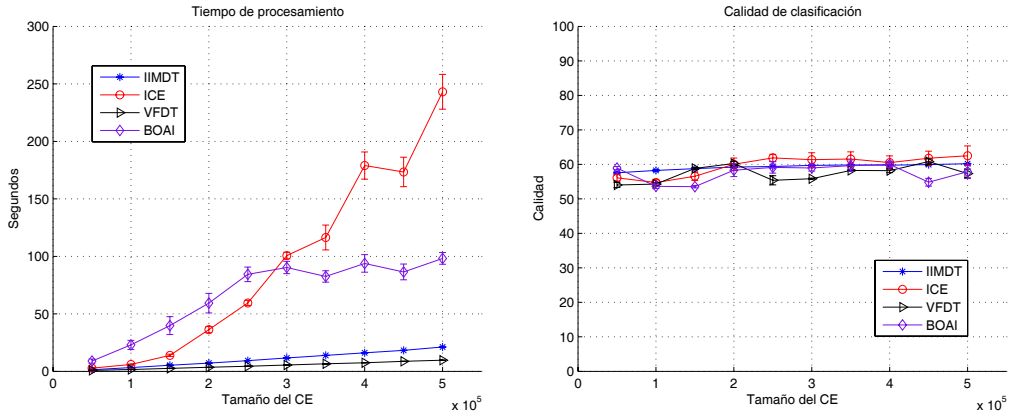


Figura 4.7: Tiempo de procesamiento y calidad de clasificación para Poker.

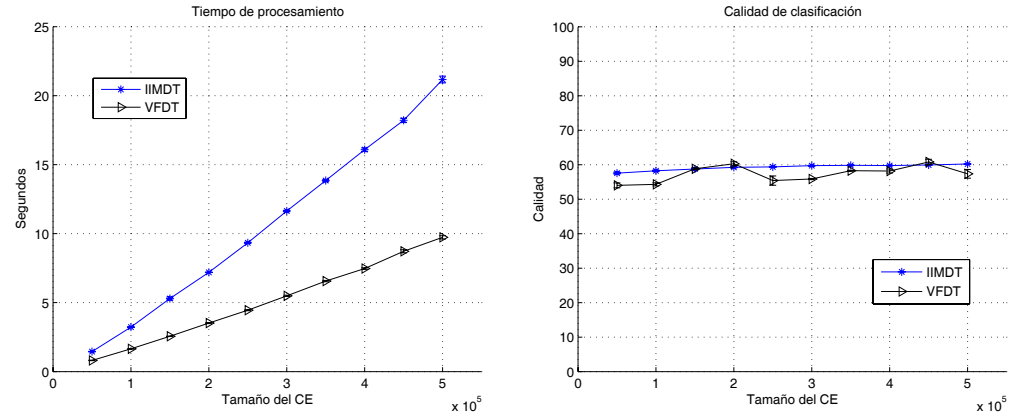


Figura 4.8: Tiempo de procesamiento y calidad de clasificación para Poker con IIMDT y VFDT.

Para SpecObj los conjuntos de entrenamiento que se generaron fueron de 50,000 a 750,000 objetos, con incrementos de 50,000. La Figura 4.9 muestra

los resultados obtenidos con IIMDT, ICE y VFDT. BOAI no se incluye en este experimento, ya que es un algoritmo para problemas de sólo 2 clases. Para SpecObj, IIMDT es el mejor algoritmo, ya que es el que tarda menor tiempo para construir el AD y es el que tiene mejor calidad de clasificación.

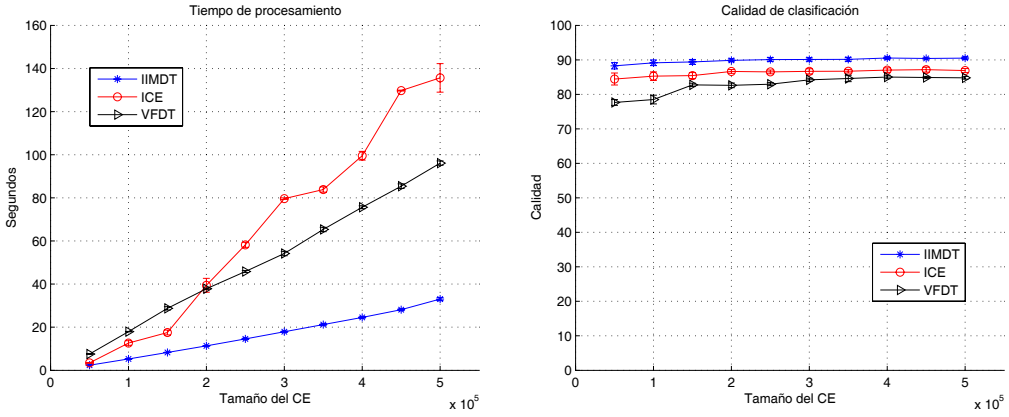


Figura 4.9: Tiempo de procesamiento y calidad de clasificación para SpecObj.

Con GalStar se hicieron conjuntos de entrenamiento de 500,000 a 4,000,000 de objetos, con incrementos de 500,000. En la Figura 4.10 sólo se muestran los resultados para IIMDT, ICE y VFDT. BOAI no se muestra porque con este conjunto de datos, el algoritmo no pudo procesar conjuntos de entrenamiento más grandes de 200,000 objetos. Como se puede observar en la Figura 4.10 la calidad de clasificación de IIMDT, ICE y VFDT es similar. Por otro lado, existe una gran diferencia entre el tiempo de procesamiento de ICE con respecto al tiempo de IIMDT y VFDT, por lo cual no se puede apreciar correctamente la diferencia entre IIMDT y VFDT. Para observar esta diferencia gráficamos sólo los resultados de IIMDT y VFDT en la Figura 4.11. De la Figura 4.11 se puede notar que ambos algoritmos obtienen una calidad de clasificación similar, sin embargo, IIMDT es más rápido que VFDT.

4.5.4.2 Conjuntos de datos sintéticos

Para cada uno de los conjuntos de datos sintéticos descritos en la Tabla 4.1, se crearon diversos conjuntos de entrenamiento para probar el desempeño

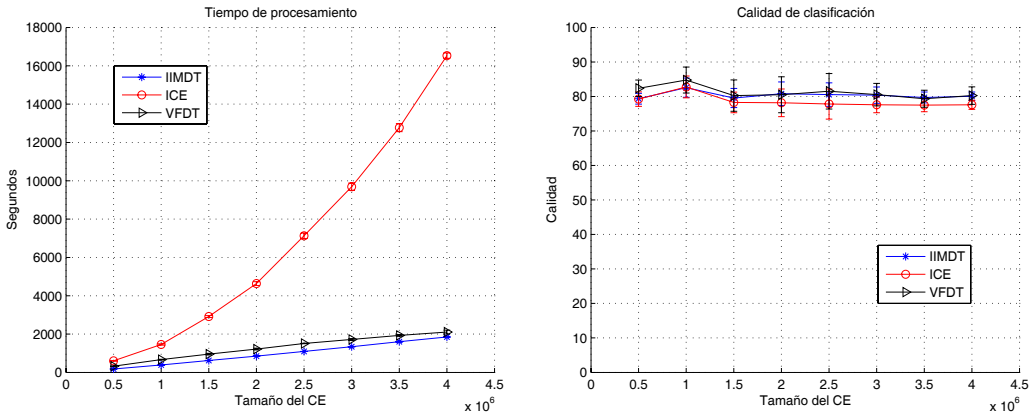


Figura 4.10: Tiempo de procesamiento y calidad de clasificación para GalStar.

de nuestro algoritmo cuando se incrementa el número de objetos en el conjunto de entrenamiento. Los conjuntos de entrenamiento van de 500,000 a 4,000,000 de objetos con incrementos de 500,000 objetos.

La Figura 4.12 muestra los resultados de IIMDT, ICE, VFDT y BOAI para los conjuntos de datos sintéticos de 5 atributos, con 2, 3 y 5 clases. BOAI sólo se muestra en el conjunto de entrenamiento de 2 clases, ya que este algoritmo está diseñado para trabajar con este tipo de problemas. Además, BOAI sólo pudo procesar hasta 3,500,000 objetos, con el conjunto de entrenamiento más grande presentó una falla de memoria y no pudo construir el AD. De la Figura 4.12 podemos observar que, para los conjuntos de entrenamiento de 2 clases nuestro algoritmo es el que menor tiempo de procesamiento emplea para construir el AD y clasificar los objetos de prueba. En cuanto a la calidad de clasificación, IIMDT, ICE y VFDT obtuvieron resultados muy similares, mientras que BOAI obtuvo bajos resultados. Con los conjuntos de entrenamiento de 3 y 5 clases se puede observar que nuestro algoritmo fue más rápido que ICE y VFDT, y en cuanto a la calidad de clasificación, IIMDT obtuvo resultados similares a VFDT, siendo ambos algoritmos mejores que ICE.

BOAI no se muestra en los resultados de los conjuntos de entrenamiento de 40 atributos ya que no pudo procesar ninguno de los conjuntos de entrenamiento, el conjunto de entrenamiento más grande de 40 atributos que pudo procesar fue de 200,000 objetos con 2 clases. Los resultados para los conjuntos de entrenamiento de 40 atributos de 2, 3 y 5 clases se muestran en la Figura

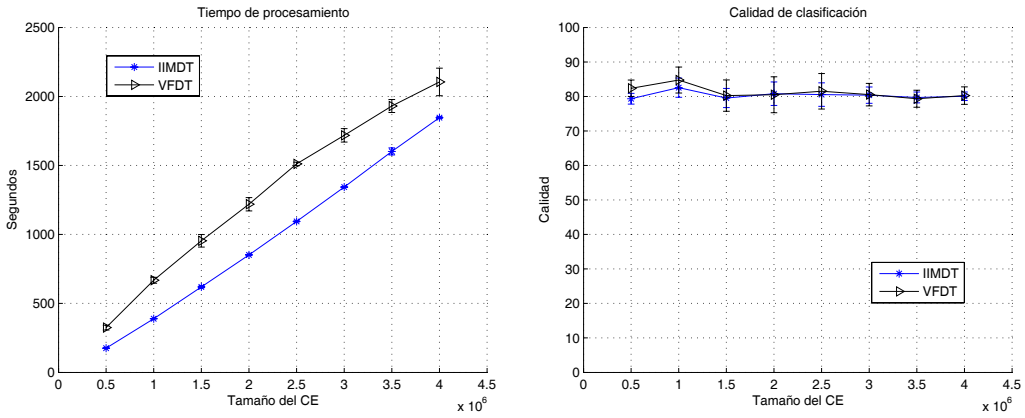
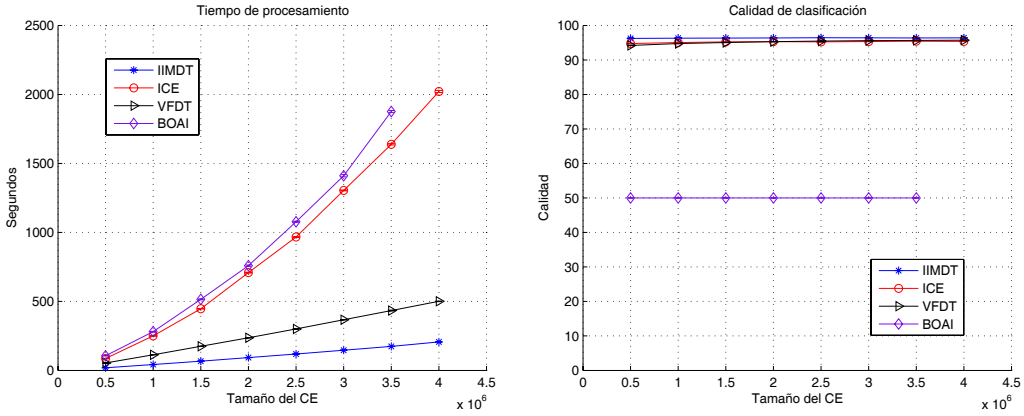


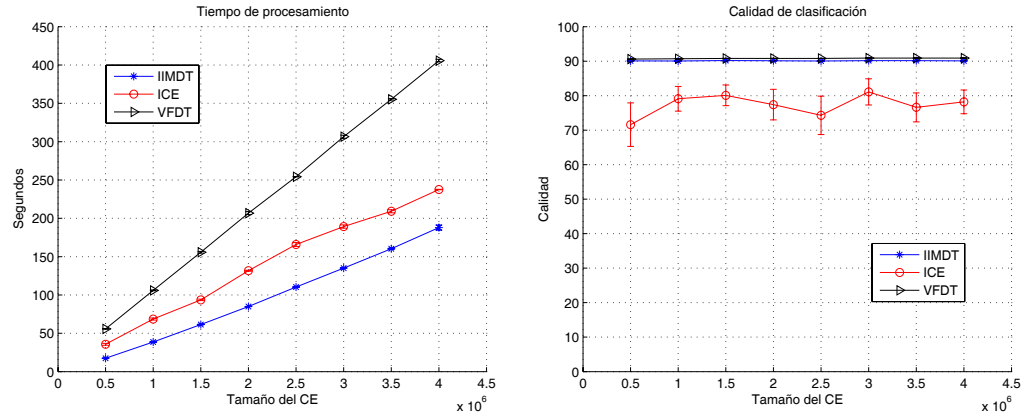
Figura 4.11: Tiempo de procesamiento y calidad de clasificación para GalStar con IIMDT y VFDT.

4.13. Como puede observarse, IIMDT es el que menor tiempo de procesamiento emplea en los tres casos. En cuanto a la calidad de clasificación, para los conjuntos de entrenamiento de 2 clases, los tres algoritmos obtienen una calidad similar, y para los conjuntos de entrenamiento de 3 y 5 clases, IIMDT y VFDT obtienen resultados similares y ambos algoritmos obtienen mejores resultados que ICE.

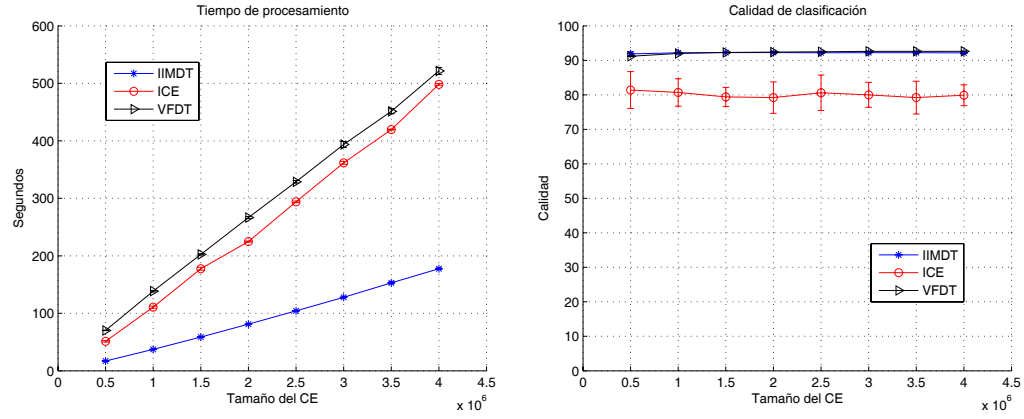
De los experimentos realizados con conjuntos de datos reales y sintéticos, incrementando el número de objetos en el conjunto de entrenamiento, se observó que nuestro algoritmo presenta un buen desempeño. Aunque el tiempo de procesamiento de IIMDT se incrementa cuando el número de objetos se incrementa, en todos los conjuntos de datos nuestro algoritmo fue más rápido en la construcción del AD que ICE, VFDT y BOAI (excepto en Poker, en donde VFDT fue ligeramente más rápido que IIMDT). En los experimentos se puede observar que IIMDT es más rápido que los otros algoritmos ya que no emplea tiempo en elegir atributos de prueba cada vez que se va a expandir un nodo, como es el caso de VFDT y BOAI; ni emplea tiempo en obtener un subconjunto de objetos para generar el AD, como es el caso de ICE. Además, IIMDT obtuvo una calidad de clasificación competitiva con la obtenida por ICE, BOAI y VFDT, incluso en algunos casos mejor.



(a) 2 clases

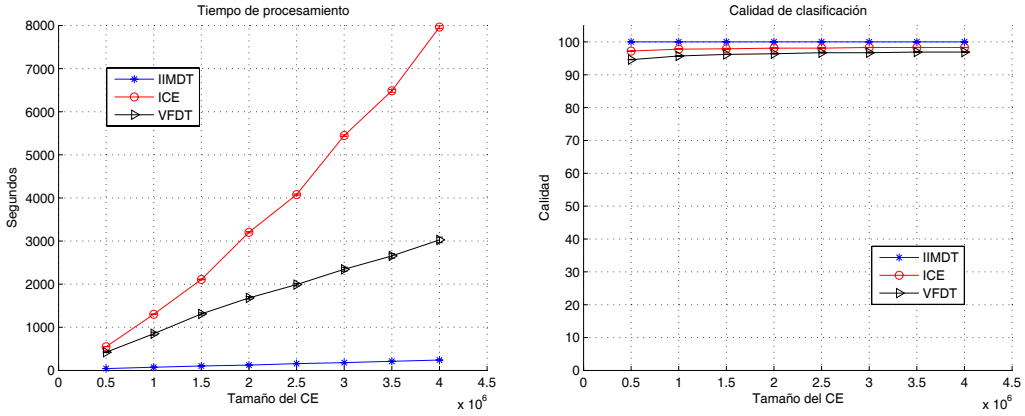


(b) 3 clases

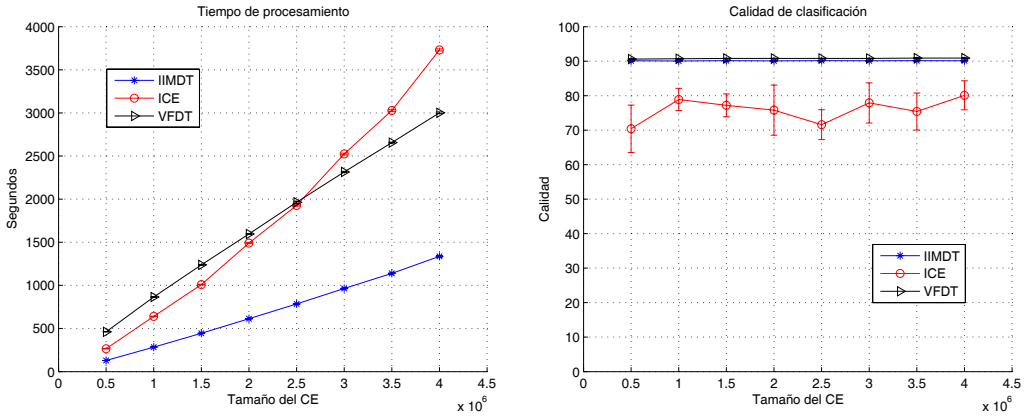


(c) 5 clases

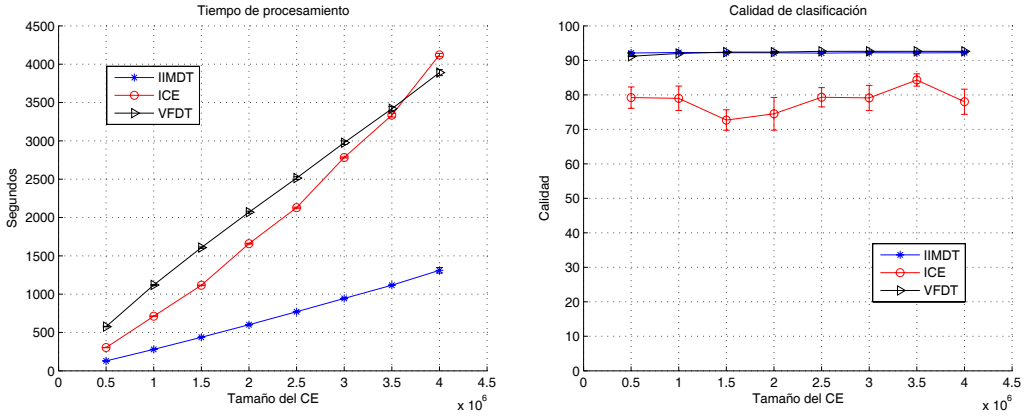
Figura 4.12: Tiempo de procesamiento y calidad de clasificación para IIMDT, ICE, VFDT y BOAI con conjuntos de entrenamiento sintéticos de 5 atributos y de 2, 3 y 5 clases.



(a) 2 clases



(b) 3 clases



(c) 5 clases

Figura 4.13: Tiempo de procesamiento y calidad de clasificación para IIMDT, ICE, VFDT y BOAI con conjuntos de entrenamiento sintéticos de 40 atributos y de 2, 3 y 5 clases.

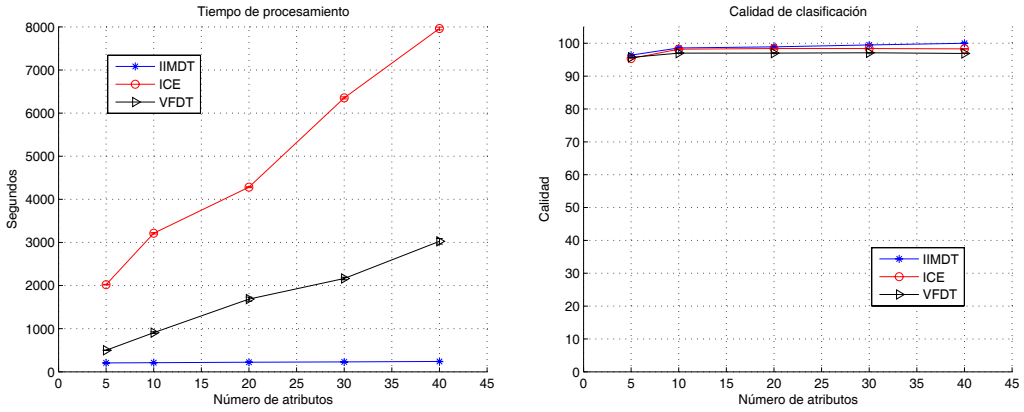
4.5.5 Incrementando el número de atributos en el conjunto de entrenamiento

Esta sección presenta los experimentos realizados para analizar el comportamiento de nuestro algoritmo en comparación con ICE y VFDT, cuando se incrementa el número de atributos en el conjunto de entrenamiento.

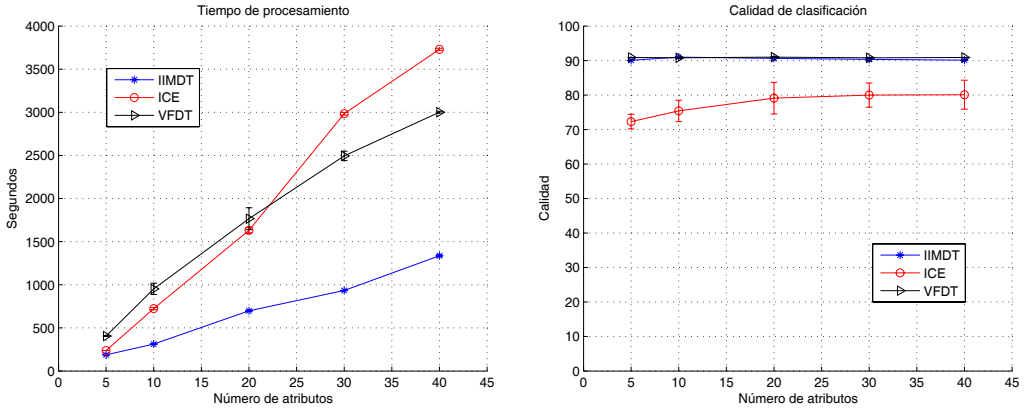
En este experimento, para cada conjunto de datos sintético (de 2, 3 y 5 clases), se crearon conjuntos de entrenamiento de 5 y de 10 a 40 atributos con incrementos de 10 atributos, con 4,000,000 de objetos cada uno. BOAI no se incluye en estos experimentos, ya que no pudo procesar ningún conjunto de entrenamiento de este tamaño.

La Figura 4.14 muestra los resultados obtenidos por los algoritmos IIMDT, ICE y VFDT. Como puede observarse, para los conjuntos de entrenamiento de 2 clases, con nuestro algoritmo el tiempo de procesamiento varía muy poco cuando se incrementa el número de atributos en el conjunto de entrenamiento, requiriendo en todos los casos menor tiempo que ICE y VFDT. Este comportamiento se debe a que ICE y VFDT tienen que evaluar todos los atributos cada vez que un nodo es expandido. Por su parte, nuestro algoritmo no selecciona atributos de prueba (siempre usa todos), por lo que IIMDT realiza la expansión de un nodo rápidamente. Con respecto a la calidad de clasificación, IIMDT, ICE y VFDT obtienen resultados similares, sin importar el número de atributos en el conjunto de entrenamiento. Para los conjuntos de entrenamiento de 3 y 5 clases, nuestro algoritmo emplea un menor tiempo para la construcción del AD que ICE y VFDT. Además, cuando se incrementa el número de atributos en el conjunto de entrenamiento, ICE y VFDT incrementan su tiempo de procesamiento a diferencia de IIMDT. Para estos conjuntos de entrenamiento, la calidad de clasificación de IIMDT y VFDT es similar para todos los conjuntos de entrenamiento y mejor que la obtenida por ICE.

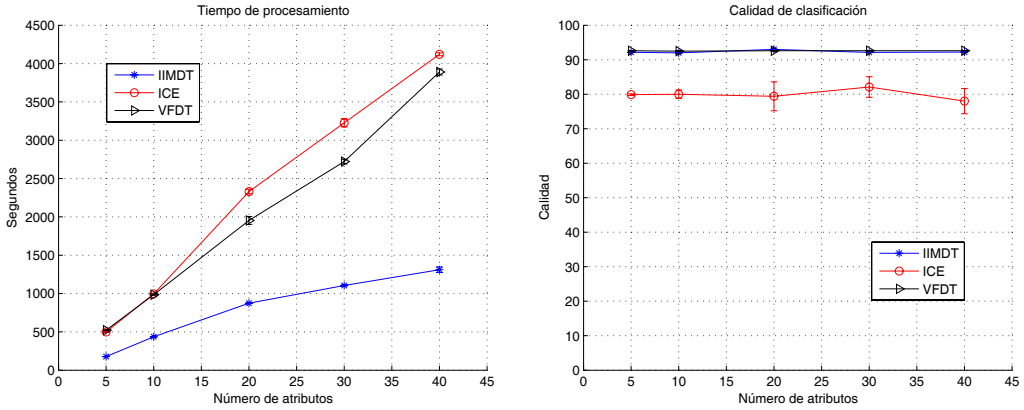
Como puede observarse en los experimentos anteriores, el tiempo de procesamiento de IIMDT se incrementa lentamente conforme se incrementa el número de atributos en el conjunto de entrenamiento, en comparación con ICE y VFDT, cuyos tiempos de procesamiento se incrementan más rápidamente. Además, nuestro algoritmo mantiene una calidad de clasificación similar, independientemente del número de atributos del conjunto de entrenamiento.



(a) 2 clases



(b) 3 clases



(c) 5 clases

Figura 4.14: Tiempo de procesamiento y calidad de clasificación para IIMDT, ICE y VFDT con conjuntos de entrenamiento sintéticos de 2, 3 y 5 clases, cuando el número de atributos se incrementa.

4.5.6 Uso de memoria para la construcción del AD

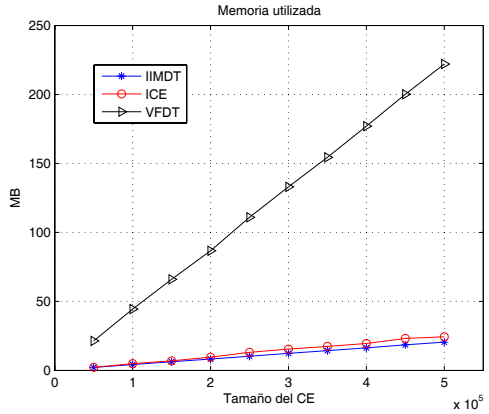
Como se analizó en la Sección 4.4, la complejidad espacial de IIMDT es del mismo orden que la complejidad espacial de ICE, VFDT y BOAI. Por esta razón, se realizó una serie de experimentos con los conjuntos de datos reales (descritos en la Tabla 4.1) para mostrar la cantidad de memoria que IIMDT, ICE y VFDT utilizan en la construcción de los ADs y así analizar la diferencia que existe entre estos algoritmos. BOAI no se muestra en estos experimentos ya que, como se pudo apreciar en los experimentos anteriores, diversos conjuntos de entrenamiento no pudieron ser procesados por este algoritmo, debido a que la memoria fue insuficiente para almacenar los conjuntos de entrenamiento.

La cantidad de memoria utilizada por los algoritmos fue medida utilizando la herramienta *memory* [Gar09] de Linux. La Figura 4.15 muestra la cantidad de memoria utilizada por los algoritmos para Poker, SpecObj y GalStar.

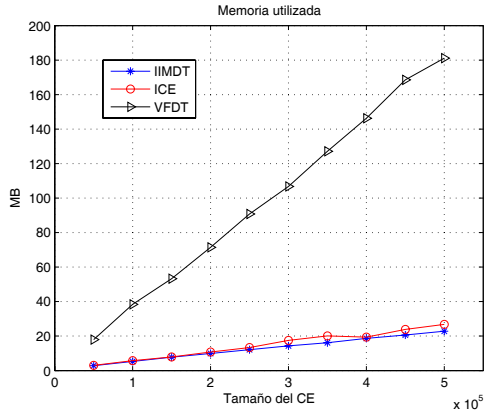
Como puede observarse, IIMDT e ICE utilizan similar cantidad de memoria, pero nuestro algoritmo, a diferencia de ICE, utiliza todo el conjunto de entrenamiento para generar el AD. Ambos algoritmos utilizan menos memoria que VFDT, ya que para elegir un atributo de prueba, cada vez que expande un nivel del AD, este algoritmo debe mantener en memoria el conjunto de entrenamiento completo.

4.6. Discusión

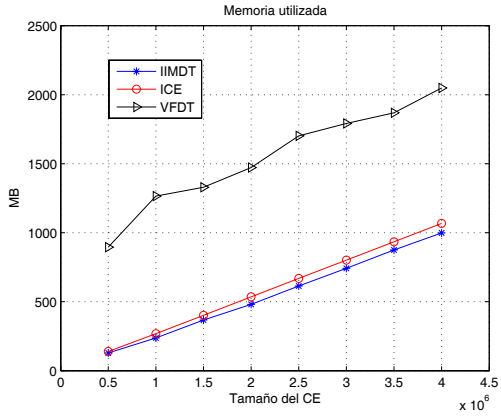
En este capítulo se propuso el algoritmo IIMDT, el cual, de manera incremental, construye ADs multivaluados para grandes conjuntos de datos numéricos. De acuerdo a los experimentos anteriores se pudo observar que IIMDT emplea un tiempo de procesamiento (incluyendo el tiempo de preprocesamiento, de construcción del AD y de clasificación de los objetos de prueba) menor al de ICE, VFDT y BOAI, los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos. Tomando en cuenta todos los conjuntos de datos, en promedio, IIMDT es hasta 13.78, 5.85 y 7.03 veces más rápido que ICE, VFDT y BOAI, respectivamente, cuando se incrementa el número de objetos en el conjunto de entrenamiento. Por su parte, cuando se incrementa el número de atributos en el conjuntos de entrenamiento, IIMDT es hasta 15.29 y 6.67 veces más rápido que ICE y VFDT, respectivamente. Además, la calidad de clasificación de nuestro algoritmo es similar a la de estos algoritmos, independientemente del número de atributos



(a) Poker



(b) SpecObj



(c) GalStar

Figura 4.15: Cantidad de memoria utilizada por IIMDT, ICE y VFDT con Poker, SpecObj y GalStar.

y de objetos en el conjunto de entrenamiento. Esta similitud en calidad de clasificación se muestra en el Anexo 1 de esta tesis, en donde se presenta un análisis de significancia de los resultados obtenidos entre nuestro algoritmo y los utilizados para compararlo.

Con respecto al parámetro s de IIMDT, se mostró experimentalmente que asignándole un valor pequeño se obtienen buenos resultados. El valor de $s = 100$ resulta una buena opción ya que mantiene un buen compromiso entre velocidad y calidad. Además, se observó que IIMDT utiliza menos memoria para construir un AD en comparación con ICE, VFDT y BOAI, lo que le permite procesar grandes conjuntos de datos de manera más eficiente.

Finalmente, podemos afirmar que con IIMDT se cumple el primer objetivo particular de esta tesis. IIMDT es un algoritmo para grandes conjuntos de datos numéricos, que genera ADs multivaluados, que usan todo el conjunto de atributos como atributos de prueba en sus nodos internos. IIMDT es más rápido que ICE, VFDT y BOAI, los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos. Además, IIMDT obtiene una calidad de clasificación competitiva con estos algoritmos.

Capítulo 5

ADs multivaluados utilizando subconjuntos de atributos

En este capítulo se introduce el algoritmo IIMDTS, el cual es una extensión del algoritmo IIMDT. La diferencia entre estos algoritmos es que IIMDTS elige un subconjunto de atributos diferente para caracterizar a cada nodo interno, de aquí la **S** al final del nombre. Se propone esta modificación ya que IIMDT utiliza la mayor parte del tiempo de procesamiento en recorrer el AD con cada objeto del conjunto de entrenamiento. Debido a que para descender de un nivel a otro del AD, cada objeto tiene que compararse, usando todos los atributos, con la combinación de valores de cada arco, si los objetos están descritos en términos de una gran cantidad de atributos, el tiempo de procesamiento para esta operación puede ser muy grande. Por esta razón, se propone un algoritmo de generación de ADs multivaluados para grandes conjuntos de datos numéricos, que selecciona subconjuntos de atributos de prueba diferentes para los nodos internos.

Un AD construido por IIMDTS será similar a uno construido por IMDT. La diferencia es que los nodos internos estarán caracterizados por un subconjunto de atributos de prueba y cada arco de salida tendrá asociada una combinación de valores para los atributos de prueba eligidos para ese nodo.

5.1. Selección de los atributos de prueba

Actualmente existen diversos métodos para la selección de atributos de prueba en ADs multivaluados (ver Sección 2.2). Sin embargo, aplicar cualquiera de estos métodos resulta en un alto costo computacional, ya

que tienen que revisar diversos subconjuntos de atributos antes de elegir al mejor, lo que hace que estos métodos no sean una buena opción cuando se quiere generar ADs para grandes conjuntos de datos.

Para hacer una rápida selección de atributos de prueba, se propone utilizar uno de los criterios de selección de atributos más comunes en la generación de ADs, la Proporción de la Ganancia de Información. Para expandir un nodo se propone aplicar la Proporción de la Ganancia de Información a cada atributo, pero utilizando sólo los objetos almacenados en el nodo a expandir, para elegir a los mejores n atributos (n es un parámetro de IIMDTS). De esta forma, la selección del subconjunto de atributos de prueba es muy rápida, ya que nuestro algoritmo sólo utiliza los s objetos almacenados en el nodo.

5.2. Construcción del AD

El proceso de construcción del AD es similar al de IIMDT, incluyendo el preprocesamiento que se hace al conjunto de entrenamiento, descrito en la Sección 4.1. Cada objeto recorre el AD hasta llegar a una hoja, donde es almacenado. La diferencia está en la forma de expansión o de actualización de un nodo.

Si los s objetos almacenados en el nodo pertenecen a dos o más clases, se expande el nodo. El primer paso en la expansión de un nodo es encontrar el subconjunto de atributos de prueba que divida de la manera más homogénea posible a los objetos almacenados en el nodo. Para obtener el subconjunto de atributos de prueba, se aplica la selección de atributos de prueba descrita en la Sección 5.1, en donde los n atributos con mayor ganancia de información son elegidos para caracterizar al nodo a expandir. Una vez elegido el subconjunto de atributos de prueba, se crea un arco de salida por cada clase de objetos en el nodo y a cada arco se le asocia una combinación de valores. Para obtener la combinación de valores de un arco, se calcula la media de los valores de cada atributo de prueba de los objetos almacenados en el nodo que pertenecen a la clase asociada al arco. Finalmente, se eliminan los objetos almacenados en el nodo, con lo cual se evita almacenar todos los objetos de entrenamiento en memoria. La Figura 4.1 ilustra también el proceso de expansión realizado por IIMDTS, la única diferencia es que para elegir la combinación de valores que se asocia a cada arco, IIMDTS sólo toma en cuenta los atributos de prueba elegidos.

Por otra parte, si los s objetos almacenados en el nodo a expandir

pertenecen todos a la misma clase, entonces se mantiene el nodo como una hoja y sólo se actualiza la combinación de valores asociada al arco de entrada del nodo. Para realizar esta actualización se calcula, para cada atributo de prueba que aparece en el nodo padre, la media de los valores de ese atributo en los objetos almacenados en el nodo. Posteriormente para cada atributo de prueba del nodo de entrada, se promedia la media calculada con el valor asociado al arco de entrada. Después de calcular este valor para cada atributo de prueba, se eliminan los objetos almacenados en el nodo. La Figura 4.2 muestra el proceso de actualización que sigue IIMDTS, pero este algoritmo sólo toma en cuenta los atributos de prueba que están asociados al arco de entrada del nodo.

Cuando todos los objetos de entrenamiendo se han procesado, se asigna a cada hoja, del AD construido, la clase mayoritaria de los objetos que quedaron almacenados en la hoja. En caso de que una hoja esté vacía, entonces se le asigna a la hoja la clase con la cual se creó el arco de entrada de la hoja.

Las Figuras 5.1 y 5.2 muestran el algoritmo IIMDTS y la función que procesa un nodo, cuando éste tiene almacenados s objetos. Como se puede observar, el algoritmo de la Figura 5.1 es similar al de IIMDT, la diferencia entre IIMDT e IIMDTS está en el proceso de expansión de un nodo. IIMDTS permite elegir subconjuntos de atributos de prueba para caracterizar a un nodo interno. Este proceso se puede observar en la Figura 5.2, en donde para expandir un nodo IIMDTS obtiene la Proporción de la Ganancia de Información de cada atributo, elige los n mejores atributos y con ellos expande el nodo.

5.3. Recorrido del AD

El recorrido de un objeto en un AD construido por IIMDTS es similar que el recorrido en IIMDT. La diferencia se encuentra cuando un objeto desciende de un nodo a otro. En IIMDTS el objeto sólo tomará en cuenta a los atributos de prueba que están en el nodo para decidir por cuál arco descender.

IIMDTS
Entrada:
CE: Conjunto de entrenamiento
s: Número máximo de objetos almacenados en un nodo
n: Número de atributos de prueba en un nodo interno
Salida:
AD: Árbol de decisión
Paso 1: Reorganizar el CE
Paso 2: Crear nodo raíz *R*
Paso 3: Para cada objeto *O* en *CE, hacer*
ActualizaAD(O,*R*)
Fin *Para*
Paso 4: Asignar a cada hoja la clase mayoritaria de los objetos en ella
Fin **IIMDTS**

ActualizaAD(O*, NODO*)
Si número de objetos almacenados en *NODO* < *s*, entonces
Almacenar objeto *O* en *NODO*
Incrementar el número de objetos en NODO
Si número de objetos almacenados en *NODO* = *s*, entonces
ProcesaNodo(*NODO*)
Incrementar el número de objetos en NODO
Fin *Si*
Sino /* número de objetos almacenados en *NODO* > *s* */
Para cada arco R_j *en NODO, hacer*
Obtener la diferencia D_j entre los valores de *O* y los de R_j
Fin *Para*
Hijo = Nodo ligado al arco con la menor diferencia *D*
ActualizaAD(*O, Hijo*)
Fin *Si*
Fin *ActualizaAD*

Figura 5.1: Algoritmo IIMDTS.

```
ProcesaNodo(NODO)
    Si NODO tiene objetos de más de una clase, entonces
        Para cada atributo Xi en CE, hacer
            Obtener la proporción de la ganancia de información de Xi
        Fin Para
        Elegir los n atributos que tengan la mayor ganancia como atributos de prueba
        Para cada clase Ci en NODO, hacer
            Crear un arco Ri
            Para cada atributo Aj elegido, hacer
                Obtener la media Mij entre los valores de los objetos en NODO de clase Ci
                Asignar la media Mij al arco Ri
            Fin Para
            Crear nodo asociado a Ri
        Fin Para
        Eliminar los objetos almacenados en el nodo expandido
    Sino
        Para cada atributo Ai elegido, hacer
            Obtener la media Mi entre los valores de los objetos en NODO
            Promediar Mi con la media del atributo Ai asociada al arco de entrada de NODO
        Fin Para
        Eliminar los objetos almacenados en el nodo actualizado
    Fin Si
Fin ProcesaNodo
```

Figura 5.2: Procesamiento de un nodo con s objetos con el algoritmo IIMDTS.

5.4. Análisis de complejidad del algoritmo IIMDTS

Para poder analizar la complejidad temporal del algoritmo IIMDTS, se tienen que analizar los pasos que se llevan a cabo para la construcción del AD, recorrer el AD con todos los objetos de entrenamiento y expandir todos los nodos que tengan almacenados s objetos.

Para un conjunto de entrenamiento de m objetos, divididos en k clases y descritos por x atributos, el recorrido que hacen todos los objetos de entrenamiento es al igual que el hecho en IIMDT, entonces recorrer el AD con los m objetos es, en el peor caso, $O(m^2)$. Por su parte, en el proceso de expansión interviene un nuevo parámetro en IIMDTS, elegir los n mejores atributos que caracterizarán al nodo, usando sólo los s objetos almacenados en el nodo. Esta elección es $O(x*s)$ y una vez que el subconjunto de atributos de prueba se ha elegido, IIMDTS crea a lo más k arcos y para cada arco, un subconjunto de los s objetos almacenados en el nodo es usado para calcular la media de cada atributo de prueba. Estas medias serán asociadas como valores representativos en el arco. Calcular estos valores es $O(k*(s/k)) = O(s)$, entonces expandir un nodo es $O((x*s)+s) = O(x*s)$. El número máximo de expansiones que puede hacer IIMDTS es $O(m/s)$, entonces el proceso completo de expansiones que hace IIMDTS es $O(x*s*(m/s)) = O(x*m)$, sin embargo, ya que usualmente $x \ll m$ para grandes conjuntos de datos, entonces la complejidad es $O(m)$.

Como puede observarse, los dos procesos principales que lleva a cabo IIMDTS (el recorrido del AD con los objetos de entrenamiento y las expansiones en el AD) tienen la misma complejidad que en IIMDT. Por lo tanto, la complejidad de IIMDTS es de igual orden que la complejidad de IIMDT.

Con respecto a la complejidad espacial de IIMDTS, aunque en este algoritmo se tiene la opción de elegir un subconjunto de n atributos en cada nodo interno, en el peor caso $n = x$, por lo tanto la complejidad espacial de IIMDTS es de igual orden que la de IIMDT.

5.5. Resultados experimentales

Los experimentos realizados con IIMDTS se dividen en cuatro partes. Primero se realizó un estudio experimental con los parámetros que intervienen en el proceso de construcción del AD (s y n) con la finalidad de analizar el comportamiento de IIMDTS cuando el valor de estos paráme-

tros varía. Posteriormente se realizaron experimentos variando el número de objetos y de atributos en el conjunto de entrenamiento, para analizar el desempeño de nuestro algoritmo. Además, en estos experimentos se incluyó una comparación contra los algoritmos ICE, VFDT y BOAI y contra el algoritmo anterior, IIMDT. Por último, ya que la complejidad espacial de IIMDTS es de igual orden que la complejidad de los algoritmos utilizados para comparar, se hicieron experimentos que muestran el uso de memoria de los algoritmos.

Los conjuntos de datos utilizados para estos experimentos son los descritos en la Tabla 4.1 y las especificaciones, de cómo se hicieron los experimentos y del equipo utilizado, son las mismas que en IIMDT (ver Sección 4.5).

5.5.1 Paramétros s y n del algoritmo IIMDTS

En esta sección se realizó un estudio experimental para analizar cómo afecta la variación de los parámetros s y n al desempeño de IIMDTS. Para esto, se utilizaron los conjuntos de datos reales descritos en la Tabla 4.1. Para cada conjunto de datos se creó un conjunto de entrenamiento de 100,000 objetos. Se evaluaron diferentes valores para cada parámetro. En el caso del parámetro s se evaluaron los valores 50 y de 100 a 600 con incrementos de 100. Para n se evaluaron diferentes valores de acuerdo al número de atributos de cada conjunto de datos.

Para Poker se evaluaron los valores 1 y de 2 a 10 con incrementos de 2, para el parámetro n. Los resultados se muestran en la Figura 5.3, en donde se puede apreciar que tanto para s como para n, los valores más pequeños son con los que se obtienen mejores resultados. Con estos valores se requiere menor tiempo de procesamiento y la calidad de clasificación es muy similar a la calidad que se obtiene con valores más grandes.

Debido a que SpecObj tiene pocos atributos, se utilizaron todos los posibles valores de n para este experimento (1, 2, 3, 4, 5). En la Figura 5.4 se pueden observar los resultados obtenidos con este conjunto de datos. Respecto a la calidad de clasificación se puede notar que IIMDTS obtiene similares resultados con cualquier valor que se le asigne a s y n. En cuanto al tiempo de procesamiento, éste se incrementa conforme el valor de n se incrementa, y con s se puede notar que para el valor de 100 hay un valle en la gráfica, lo que indica que con este valor el algoritmo muestra un mejor desempeño que con otros valores.

Con GalStar, para el parámetro n, se evaluaron los valores 1 y de 5 a 30

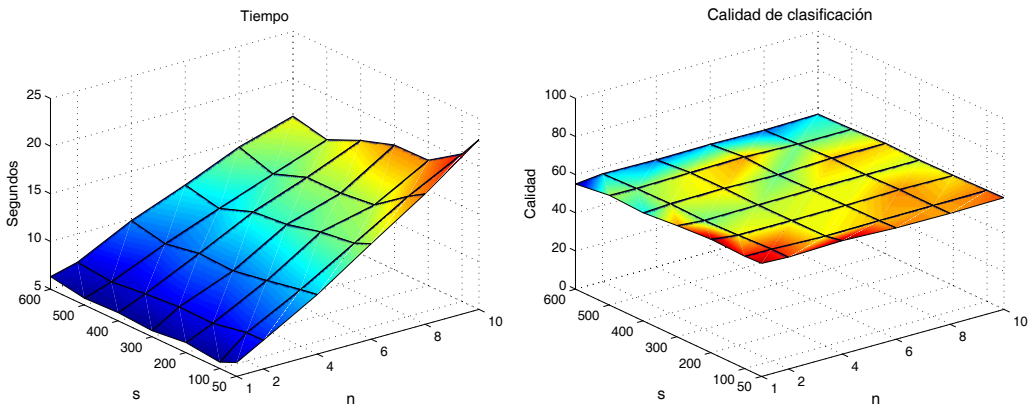


Figura 5.3: Tiempo de procesamiento y calidad de clasificación para IIMDTS variando s y n con Poker.

con incrementos de 5. Con este conjunto de datos también se obtiene una calidad de clasificación similar para todos los valores de s y n. Además, el tiempo de procesamiento empleado por IIMDTS se incrementa conforme el valor de n se incrementa y, así como ocurre con Spec, la gráfica del tiempo de procesamiento presenta un valle en los resultados obtenidos con $s = 100$.

De este estudio experimental podemos observar que al utilizar valores pequeños para s y n se obtiene una buena calidad de clasificación, ya que la calidad siempre se conservó similar sin importar el valor asignado a cada parámetro. Respecto al tiempo de procesamiento, entre más grande sea el valor asignado a los parametros, el tiempo de procesamiento se incrementa. Por esta razón, para los siguientes experimentos se utilizaron valores pequeños para s y n. Para s se utilizó el valor de 100, ya que con este valor se obtuvieron los mejores resultados. Para n se utilizó, para los conjuntos de datos con menos de 10 atributos, valores de 1 y 2, y para los conjuntos de datos con más de 10 atributos, 1 y 5.

5.5.2 Incrementando el número de objetos en el conjunto de entrenamiento

Esta sección muestra los resultados obtenidos con IIMDTS para los conjuntos de datos reales y sintéticos descritos en la Tabla 4.1, cuando se incrementa el número de objetos en el conjunto de entrenamiento. Se realizaron estos experimentos para analizar el comportamiento de nuestro algoritmo

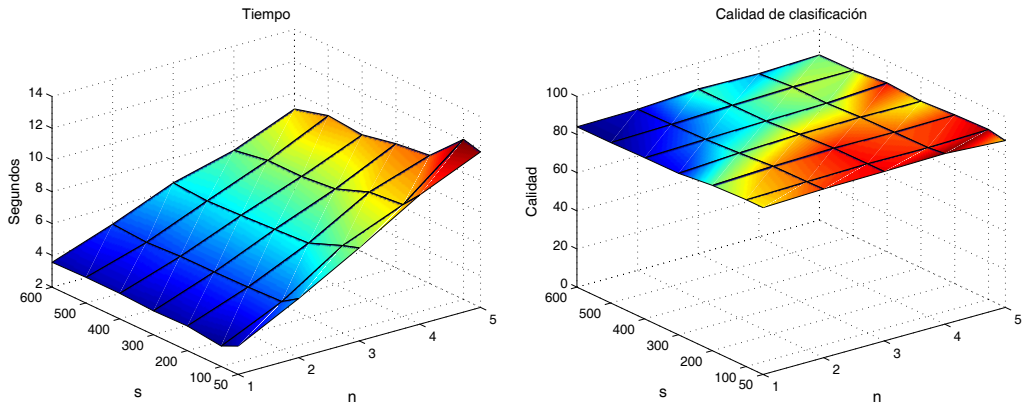


Figura 5.4: Tiempo de procesamiento y calidad de clasificación para IIMDTS variando s y n con SpecObj.

cuando se incrementa el número de objetos. Además, se presenta una comparación de IIMDTS contra los algoritmos de generación de ADs para grandes conjuntos de datos descritos en la Sección 4.5.2 y contra IIMDT.

5.5.2.1 Conjuntos de datos reales

Para efectuar estos experimentos se utilizaron los conjuntos de entrenamiento descritos en la Sección 4.5.4.1. La Figura 5.6 muestra los resultados obtenidos con Poker, utilizando IIMDTS (IIMDTS-1 cuando $n = 1$ e IIMDTS-2 cuando $n = 2$), ICE, VFDT, BOAI e IIMDT. Como puede observarse, todos los algoritmos obtienen una calidad de clasificación similar pero IIMDTS, VFDT e IIMDT son más rápidos que ICE y BOAI. Debido a que en la Figura 5.6 no se puede apreciar la diferencia entre IIMDTS, VFDT e IIMDT, se graficaron en la Figura 5.7 los resultados de estos algoritmos. VFDT es el algoritmo que menor tiempo emplea, sin embargo IIMDT-1 logra superar a IIMDT y se acerca al tiempo de procesamiento de VFDT.

Con SpecObj, IIMDTS supera a IIMDT y al resto de los algoritmos, ya que IIMDTS es 3, 9.5 y 13.5 veces más rápido que IIMDT, VFDT e ICE, respectivamente. Además, todos los algoritmos obtienen una calidad de clasificación similar con este conjunto de datos. Estos resultados pueden observarse en la Figura 5.8.

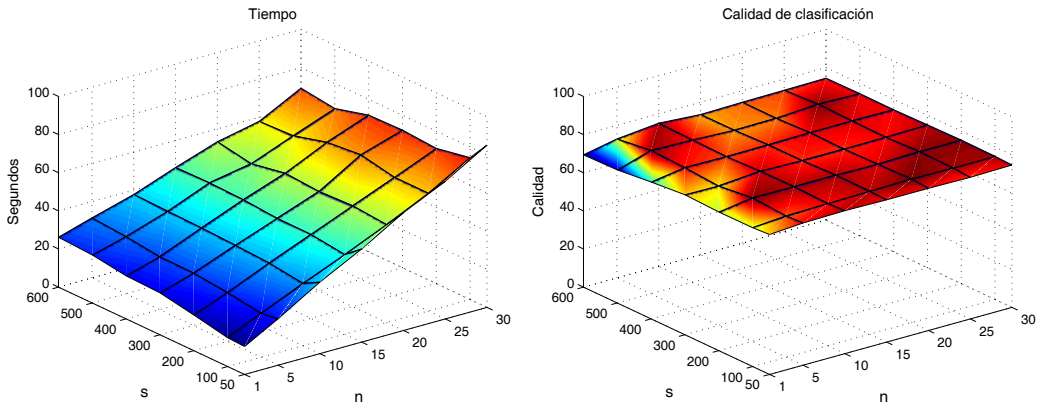


Figura 5.5: Tiempo de procesamiento y calidad de clasificación para IIMDTS variando s y n con GalStar.

En la Figura 5.9 se muestran los resultados obtenidos con IIMDTS, ICE, VFDT e IIMDT utilizando GalStar. Como puede observarse IIMDTS, VFDT e IIMDT son claramente mejores que ICE, ya que los tres algoritmos emplean un tiempo de procesamiento considerablemente menor al empleado por ICE. Por esta razón, en la Figura 5.10 se presentan los resultados para GalStar sin el algoritmo ICE, para poder apreciarlos mejor. Como puede observarse IIMDTS es mejor que VFDT e IIMDT, ya que aunque los tres algoritmos obtienen una calidad de clasificación similar, IIMDTS es 2.7 y 3.2 veces más rápido que IIMDT y VFDT, respectivamente.

5.5.2.2 Conjuntos de datos sintéticos

En este experimento se usaron los conjuntos de entrenamiento descritos en la Sección 4.5.4.2. Para IIMDTS, con los conjuntos de entrenamiento de 5 atributos, se utilizaron los valores 1 y 2 para el parámetro n, y para los conjuntos de entrenamiento de 40 atributos se utilizaron los valores 1 y 5, de acuerdo al análisis experimental realizado en la Sección 5.5.1. La Figura 5.11 muestra los resultados con los conjuntos de entrenamiento de 5 atributos de 2, 3 y 5 clases. Como se puede observar, en los tres casos, IIMDTS (eligiendo 1 o 2 atributos de prueba en sus nodos internos) fue más rápido que ICE, VFDT, BOAI e IIMDT. Con respecto a la calidad de clasificación, para los conjuntos de entrenamiento de 2 clases, IIMDTS, ICE, VFDT e IIMDT

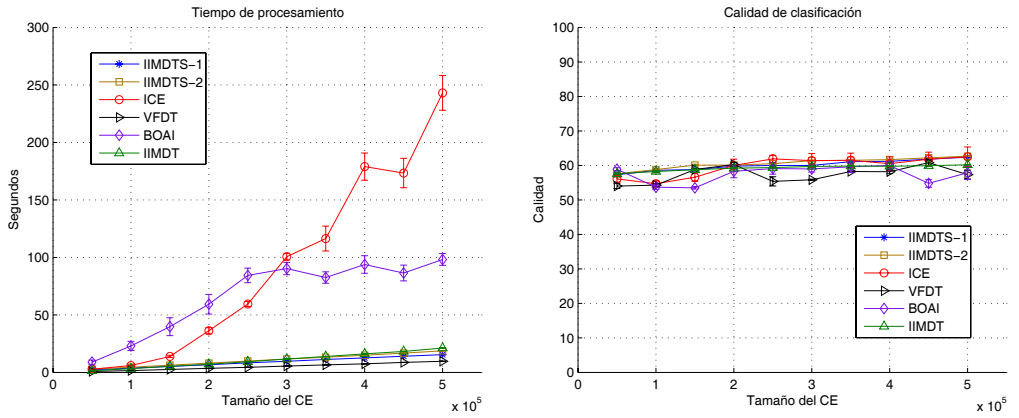


Figura 5.6: Tiempo de procesamiento y calidad de clasificación para Poker.

obtuvieron similares resultados, y todos ellos fueron superiores a BOAI. En el caso de los conjuntos de entrenamiento de 3 y 5 clases, IIMDTS, VFDT e IIMDT tuvieron una calidad similar, siendo estos algoritmos superiores a ICE.

Por su parte, en los conjuntos de entrenamiento de 40 atributos y 2 clases, IIMDTS e IIMDT fueron más rápidos que ICE y VFDT, y la calidad de clasificación de todos los algoritmos resultó muy similar. Para los conjuntos de entrenamiento de 3 y 5 clases, IIMDTS tuvo un comportamiento mejor que el resto de los algoritmos, ya que fue más rápido que ICE, VFDT e IIMDT, teniendo una calidad similar a la de VFDT e IIMDT y superior a la de ICE. Estos resultados se pueden apreciar en la Figura 5.12.

A partir de estos experimentos, tanto con los conjuntos de datos reales como con los conjuntos de datos sintéticos, podemos observar que IIMDTS mantiene un buen desempeño, independientemente del número de objetos en el conjunto de entrenamiento. El tiempo de procesamiento se incrementa poco cuando el número de objetos se incrementa, además, en general IIMDTS fue más rápido que el resto de los algoritmos contra los que se comparó. En cuanto a la calidad de clasificación, IIMDTS obtiene una calidad muy similar a la de los otros algoritmos.

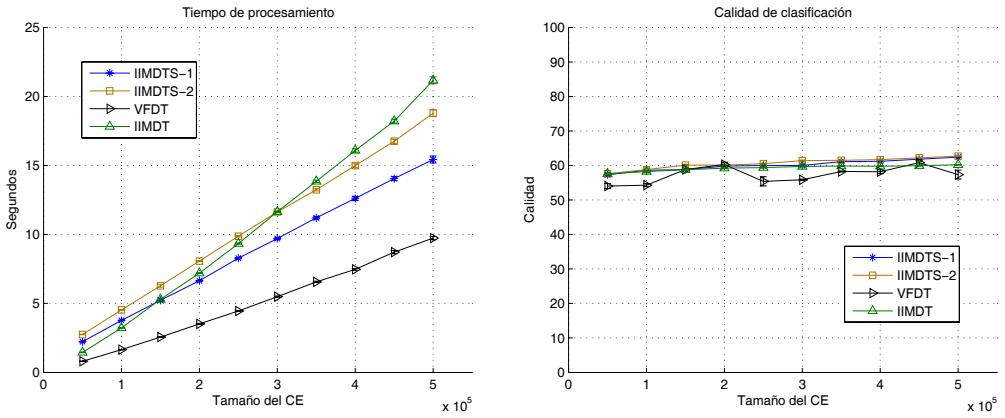


Figura 5.7: Tiempo de procesamiento y calidad de clasificación para Poker con IIMDTS, VFDT e IIMDT.

5.5.3 Incrementando el número de atributos en el conjunto de entrenamiento

Para realizar estos experimentos se utilizaron los conjuntos de entrenamiento descritos en la Sección 4.5.5. Dado que los valores de n elegidos para realizar los experimentos anteriores fueron 1, 2 y 1, 5 (según el número de atributos que tuviera el conjunto de entrenamiento que se fuera a procesar), estos experimentos se realizaron usando los tres valores de n (1, 2 y 5).

En la Figura 5.13 se muestran los resultados obtenidos. Como puede observarse, IIMDTS (con los tres valores para n) mantiene similares resultados cuando se incrementa el número de atributos en el conjunto de entrenamiento. El incremento en el tiempo de procesamiento de IIMDTS es muy poco en comparación con el incremento que tiene el tiempo de procesamiento empleado por ICE y VFDT. Respecto a la calidad de clasificación, se puede observar que es muy similar, independientemente del número de atributos en el conjunto de entrenamiento.

5.5.4 Uso de memoria para la construcción del AD

En la Sección 5.4 se observó que la complejidad de IIMDTS es del mismo orden que la complejidad que IIMDT, lo cual quiere decir que IIMDTS tiene una complejidad espacial del mismo orden que la de los algoritmos ICE,

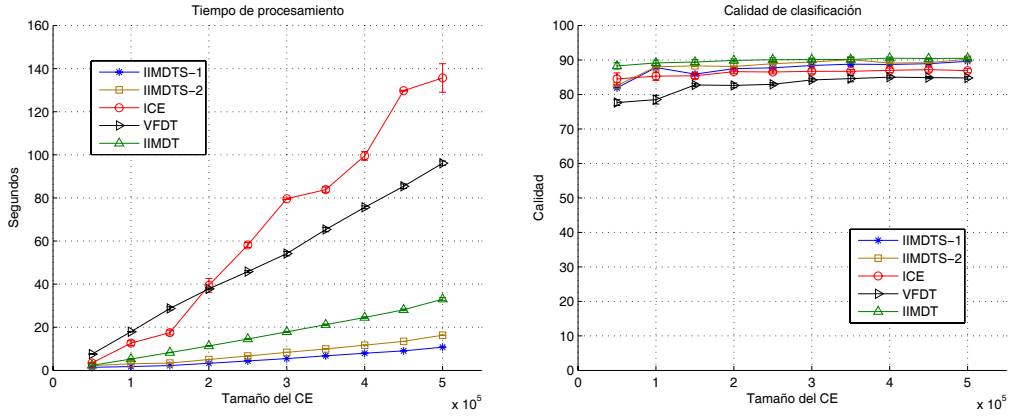


Figura 5.8: Tiempo de procesamiento y calidad de clasificación para SpecObj.

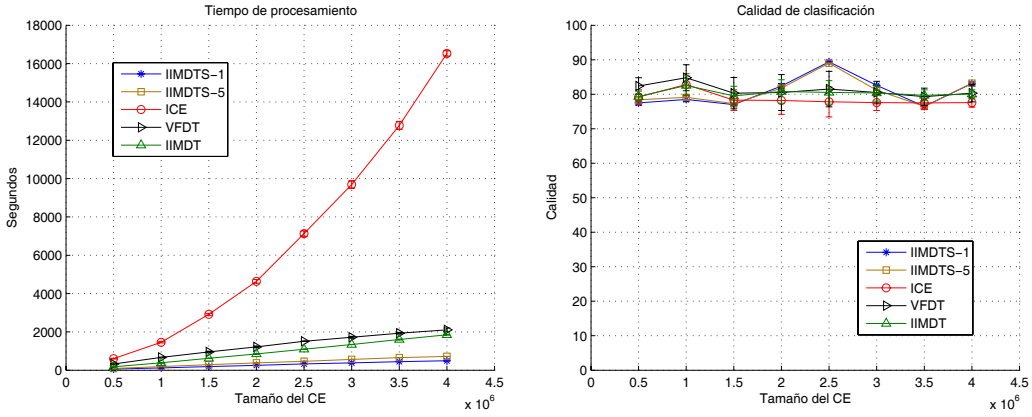


Figura 5.9: Tiempo de procesamiento y calidad de clasificación para GalStar.

VFDT y BOAI. Para ver la diferencia de la cantidad de memoria utilizada por cada algoritmo, se realizaron experimentos entre IIMDTS y estos algoritmos. Para realizar estos experimentos se utilizaron los conjuntos de datos reales descritos en la Tabla 4.1. Para medir la cantidad de memoria utilizada por cada algoritmo se utilizó la herramienta *memory* [Gar09] de Linux. Los resultados se muestran en la Figura 5.14.

De la Figura 5.14 se puede observar que IIMDTS, ICE e IIMDT, utilizan similar cantidad de memoria, pero nuestros algoritmos a diferencia de ICE utilizan todo el conjunto de entrenamiento para crear el AD. Además, los tres algoritmos utilizan menos memoria que VFDT, que es el más rápido de los algoritmos previos para grandes conjuntos de datos.

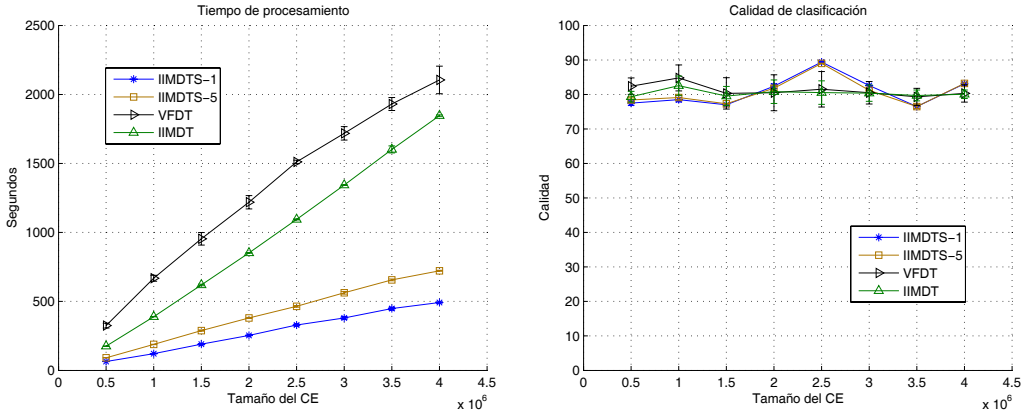
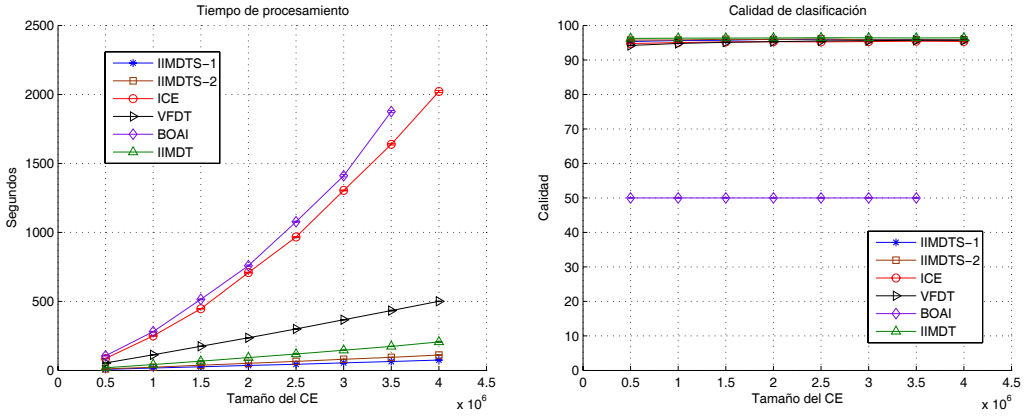


Figura 5.10: Tiempo de procesamiento y calidad de clasificación para GalStar con IIMDTS, VFTD e IIMDT.

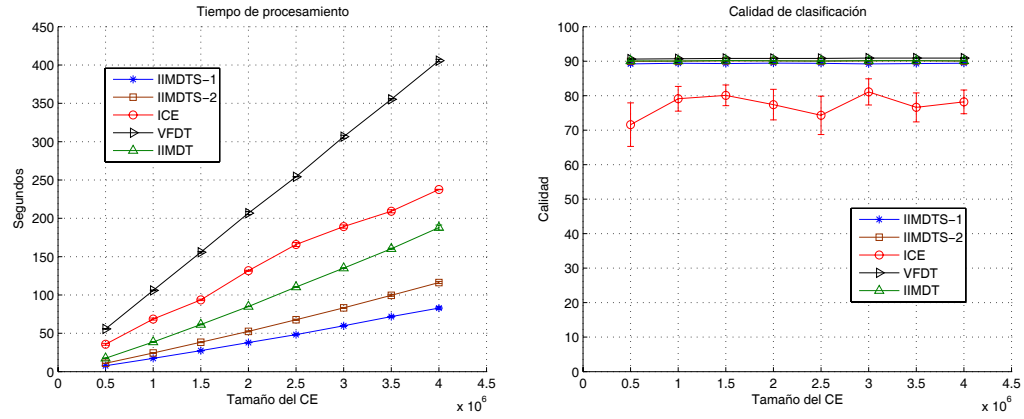
5.6. Discusión

En este capítulo se propuso el algoritmo IIMDTS, el cual construye ADs multivaluados para grandes conjuntos de datos numéricos, pero a diferencia del algoritmo anterior, IIMDTS utiliza, como atributos de prueba en los nodos internos, subconjuntos del conjunto original de atributos. De los experimentos podemos observar que IIMDTS mantiene un buen desempeño cuando se incrementa el número de objetos y atributos en el conjunto de entrenamiento. El tiempo de procesamiento empleado por IIMDTS es menor al empleado por los algoritmos ICE, VFDT y BOAI (los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos) obteniendo una calidad de clasificación similar a la de estos algoritmos. De acuerdo a los resultados sobre todos los conjuntos de datos, cuando se incrementa el número de atributos en el conjunto de entrenamiento, IIMDTS es, en promedio, hasta 19.12, 8.30 y 14.89 veces más rápido que ICE, VFDT y BOAI, respectivamente. Cuando se incrementa el número de atributos en el conjunto de entrenamiento, IIMDTS es hasta 28.24 y 15.86 veces más rápido que ICE y VFDT, respectivamente.

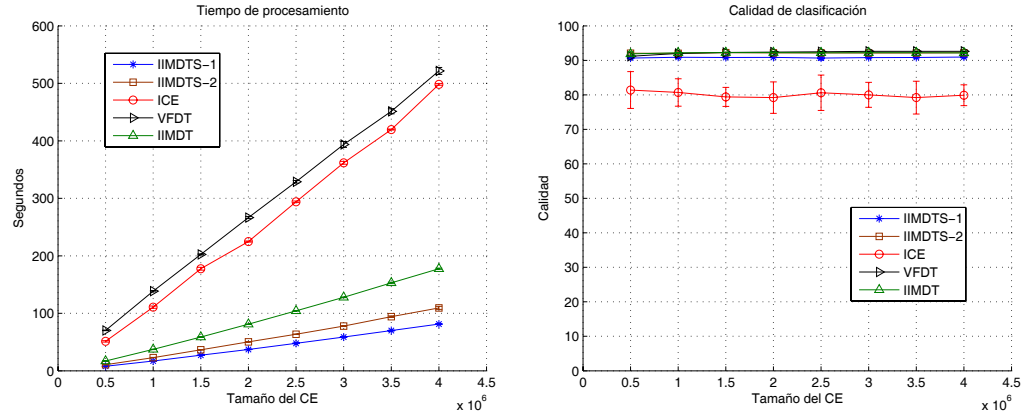
IIMDTS tiene dos parámetros (s, número máximo de objetos que puede almacenar una hoja, y n, número de atributos a elegir en cada nodo a expandir), sin embargo, analizando experimentalmente estos parámetros, se puede observar que asignándoles valores pequeños, IIMDTS emplea menor tiempo de procesamiento que con valores grandes. Es por esto que en los experimentos se uso $s = 100$ y valores pequeños para n.



(a) 2 clases

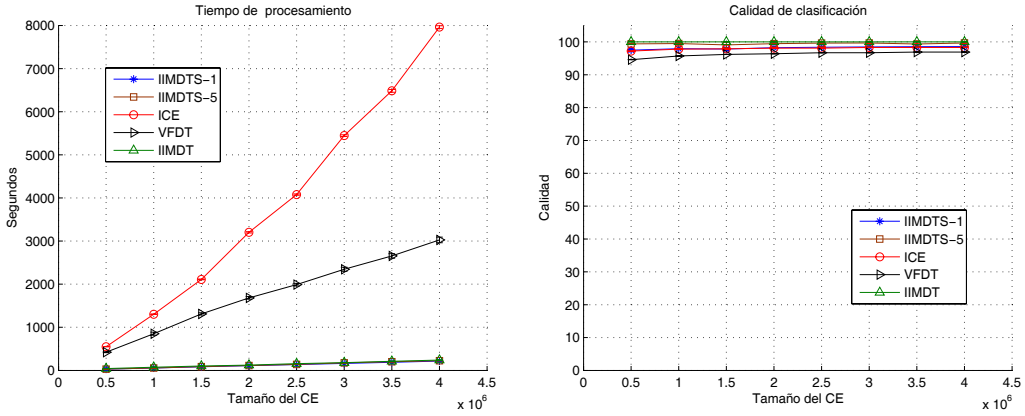


(b) 3 clases

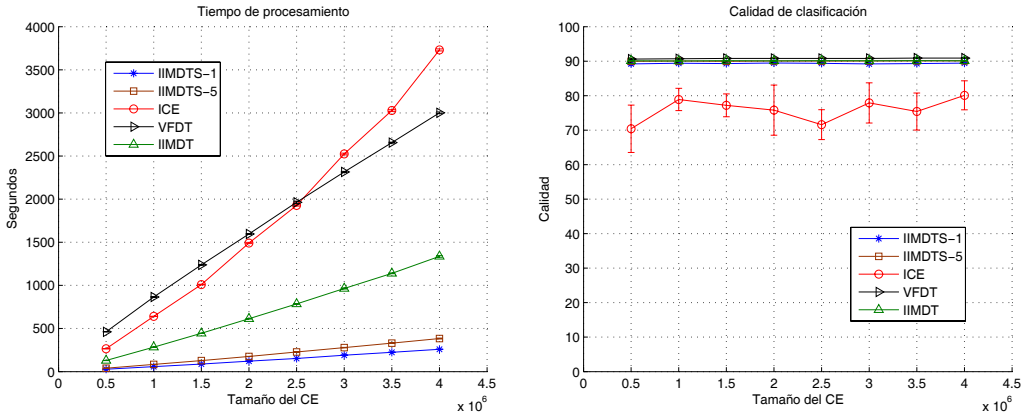


(c) 5 clases

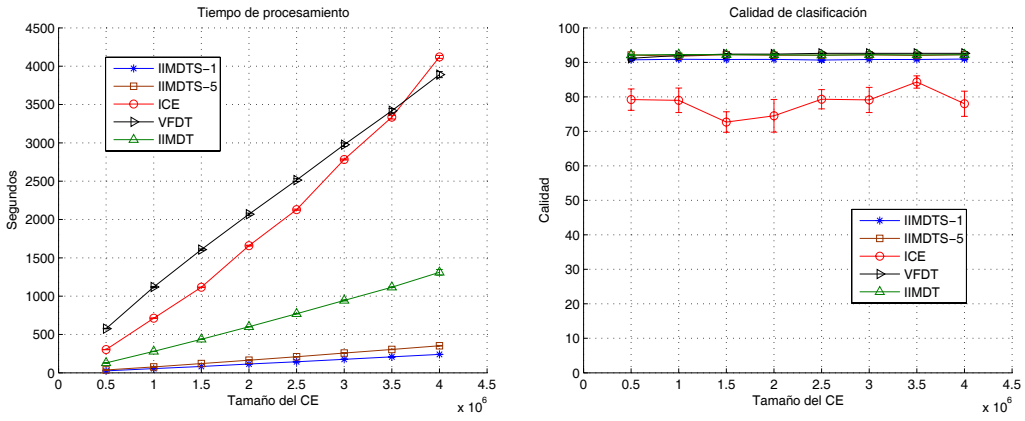
Figura 5.11: Tiempo de procesamiento y calidad de clasificación para IIMDTS, ICE, VFDT, BOAI e IIMDT con conjuntos de entrenamiento sintéticos de 5 atributos y de 2, 3 y 5 clases.



(a) 2 clases

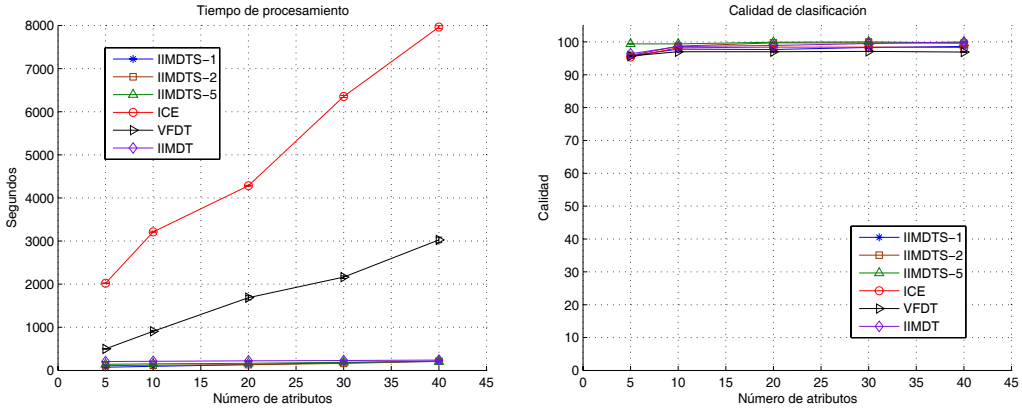


(b) 3 clases

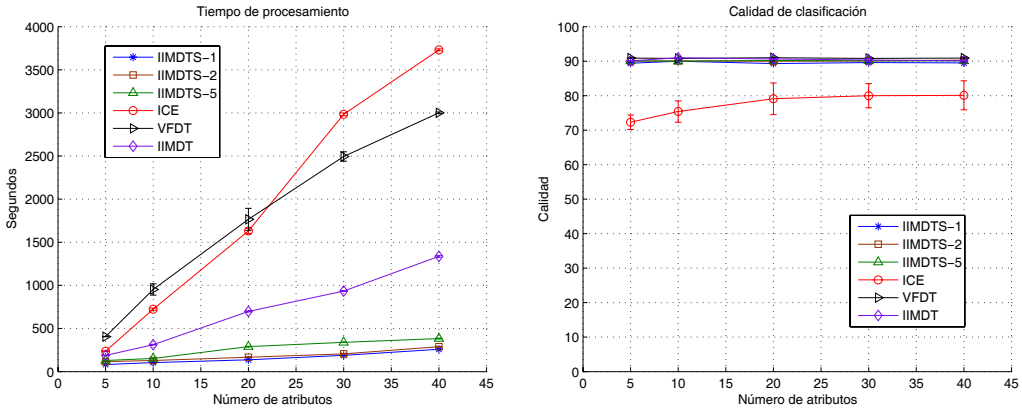


(c) 5 clases

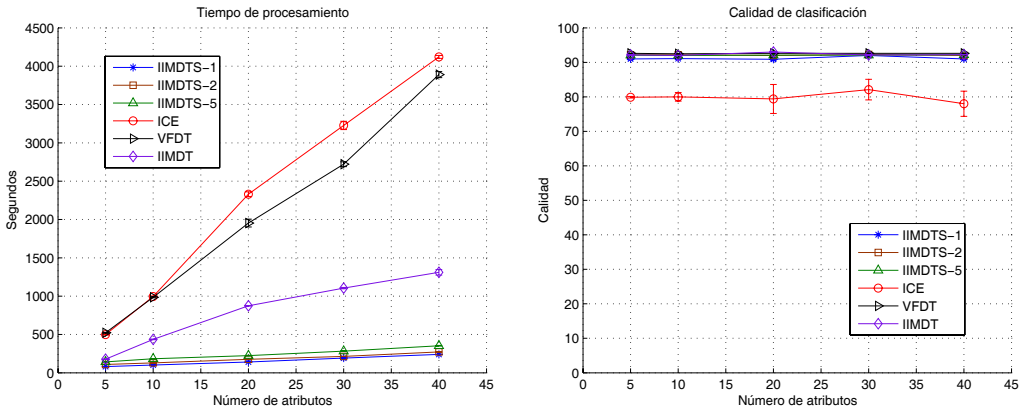
Figura 5.12: Tiempo de procesamiento y calidad de clasificación para IIMDTS, ICE, VFDT e IIMDT con conjuntos de entrenamiento sintéticos de 40 atributos y de 2, 3 y 5 clases.



(a) 2 clases

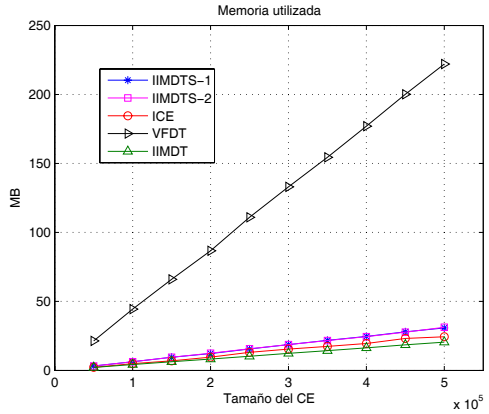


(b) 3 clases

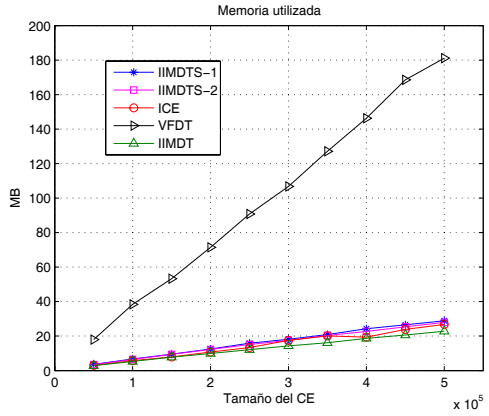


(c) 5 clases

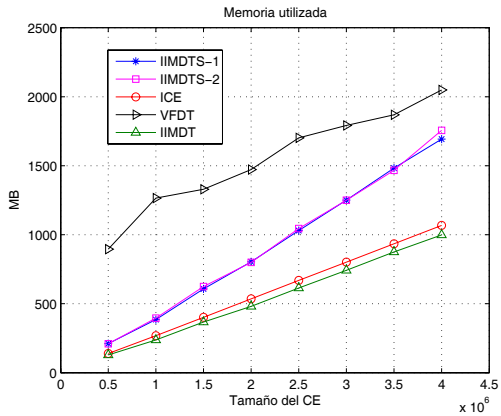
Figura 5.13: Tiempo de procesamiento y calidad de clasificación para IIMDTS, ICE y VFDT con conjuntos de entrenamiento sintéticos de 2, 3 y 5 clases, cuando el número de atributos se incrementa.



(a) 2 clases



(b) 3 clases



(c) 5 clases

Figura 5.14: Cantidad de memoria utilizada por IIMDTS, ICE y VFDT con Poker, SpecObj y GalStar.

Con respecto a la memoria utilizada por nuestro algoritmo, se pudo observar que IIMDTS requiere menos memoria que VFDT, siendo éste el más rápido de los algoritmos previos, el cual también procesa todos los objetos del conjunto de entrenamiento.

Con el algoritmo IIMDTS se cumple el segundo objetivo particular de esta tesis. Este algoritmo genera ADs multivaluados (incluso puede generar ADs univaluados), con subconjuntos de atributos en los nodos internos, para grandes conjuntos de datos con atributos numéricos. IIMDTS cumple además el objetivo de ser un algoritmo más rápido que los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos (ICE, VFDT y BOAI), obteniendo una calidad de clasificación competitiva.

Capítulo 6

ADs univaluados para grandes conjuntos de datos numéricos

Este capítulo presenta el algoritmo DTFS (por sus siglas en inglés ***D**ecision **T**rees using a **F**ast **S**plitting attribute selection*) el cual construye ADs univaluados para grandes conjuntos de datos numéricos. Se propone este algoritmo debido a que en el algoritmo IIMDTS se propuso elegir subconjuntos de atributos de prueba para expandir un nodo, aplicando el Criterio de Proporción de la Ganancia de Información como tradicionalmente se utiliza en los algoritmos de generación de ADs. Sin embargo, esta forma de obtener los atributos de prueba para un nodo, no corresponde con la forma de expandir el nodo, ya que una vez que se han obtenido los atributos de prueba, se calculan nuevos valores para caracterizar a los arcos de salida del nodo a expandir. Por lo tanto, se propone una nueva forma de aplicar el Criterio de Proporción de Ganancia de Información para elegir el atributo de prueba, acorde con la forma de expansión y recorrido del AD generado.

El AD construido por DTFS tendrá en cada nodo interno sólo un atributo de prueba y cada arco de salida del nodo tendrá sólo un valor asociado.

6.1. Selección del atributo de prueba

Como ya se había mencionado, uno de los criterios de selección de atributos de prueba más utilizado en algoritmos de generación de ADs es el Criterio de Proporción de Ganancia de Información. Sin embargo, este criterio se vuelve muy costoso cuando se trabaja con grandes conjuntos de datos.

Para poder elegir al mejor atributo en un nodo, de cada atributo, por cada par de valores diferentes se encuentra un punto de corte y con éste se mide la ganancia de información del atributo, por lo que para cada atributo se deben evaluar diversas particiones usando los objetos de entrenamiento que están en el nodo. Además, los algoritmos tradicionales deben tener en memoria a todos los objetos de entrenamiento, los cuales pueden ser demasiados, tantos que podría no ser posible almacenarlos en memoria.

Debido a que DTFS trabaja con grandes conjuntos de datos, se propone una manera distinta de seleccionar el atributo de prueba usando el Criterio de Proporción de Ganancia de Información. Para elegir al atributo de prueba se propone sólo tomar en cuenta a los s objetos almacenados en el nodo, ya que el preprocesamiento que se aplicó al conjunto de entrenamiento al inicio del algoritmo permite tener diversidad de objetos en los nodos a expandir. La idea consiste en elegir, como atributo de prueba, al atributo y a un conjunto de valores de prueba que mejor reconstruyan la partición dada por las clases de los objetos que están almacenados en el nodo. Para poder reconstuir esa partición se elegirá un número de valores de prueba igual al número de clases de los objetos del nodo.

El proceso que se sigue para seleccionar el atributo de prueba es el siguiente. Para cada atributo se obtiene, para cada clase presente en la hoja, la media de los valores del atributo que aparecen en los objetos de la hoja y que pertenecen a dicha clase. Estas medias obtenidas serán los valores de prueba que estarán asociados a ese atributo. Una vez obtenidos los valores de prueba para todos los atributos, con cada atributo y su respectivo conjunto de valores de prueba (medias) se particiona el conjunto de objetos de la hoja asociando cada objeto con el valor de prueba más cercano, de acuerdo al atributo en cuestión. Posteriormente, las particiones generadas con cada atributo y sus respectivos valores de prueba son evaluadas mediante la Proporción de Ganancia de Información como en [Qui93]. Se propone usar las medias obtenidas de las clases como valores de prueba para los atributos, para no tener que probar todos los posibles puntos de corte que se puedan calcular con los valores del atributo. De esta manera, para encontrar al mejor atributo, sólo un subconjunto de valores de prueba es evaluado para cada atributo, lo que permite que la selección del atributo de prueba sea rápida.

Finalmente, el atributo, junto con su conjunto de valores de prueba, que produzca la mejor partición, es decir el que obtenga el máximo valor de Ganancia de Información, es elegido como atributo de prueba para el nodo a expandir. En caso de que dos atributos obtengan el máximo valor con el criterio, se elige el primer atributo que lo obtuvo.

6.2. Construcción del AD

El proceso de construcción de un AD usando DTFS es similar al que se propuso con IIMDT, incluyendo el preprocesamiento que se efectúa al conjunto de entrenamiento (descrito en la Sección 4.1). La diferencia en DTFS es que para expandir un nodo se elige sólo a un atributo de prueba.

La construcción del AD inicia con un nodo raíz vacío (hoja) donde se almacenan los primeros s objetos del conjunto de entrenamiento y se expande. Posteriormente cada uno de los objetos de entrenamiento recorre el AD hasta llegar a una hoja en donde es almacenado. Cuando una hoja tiene s objetos almacenados, el nodo es expandido o actualizado de acuerdo al número de clases de los objetos almacenados en el nodo.

Cuando se tienen objetos de 2 o más clases en el nodo a expandir, primero se encuentra el atributo de prueba y el conjunto de valores de prueba, de acuerdo al criterio descrito en la Sección 6.1. Cuando se ha elegido el atributo de prueba, se crean tantos arcos como clases de objetos haya en el nodo, y a cada arco se le asocia el valor de prueba proporcionado por el criterio de selección, que corresponda a la clase con la cual se creó el arco. Después de la expansión, los s objetos son eliminados y el nodo se marca como nodo interno.

Por otro lado, si los s objetos almacenados en el nodo pertenecen a la misma clase, sólo se actualiza el valor de prueba asociado al arco de entrada del nodo. Se obtiene la media de los valores del atributo de prueba (asociado al nodo padre) en los objetos almacenados en el nodo. Este valor obtenido y el valor de prueba del arco de entrada son promediados para obtener el nuevo valor de prueba que estará asociado al arco de entrada del nodo. Finalmente, los s objetos del nodo son eliminados.

La construcción del AD termina cuando todos los objetos de entrenamiento se han procesado, entonces se le asignan a las hojas la clase mayoritaria de los objetos almacenados en ellas, o bien, si una hoja no tiene objetos almacenados, se le asigna la clase con la cual se obtuvo el valor de prueba asociado al arco de entrada de dicha hoja. Las Figuras 6.1 y 6.2 muestran la función de construcción del AD y la función de expansión o actualización de un nodo, respectivamente.

El algoritmo de la Figura 6.1 es similar al de los dos algoritmos anteriores. La diferencia está en que, cuando un nodo va a ser expandido, DTFS calcula un conjunto de valores de prueba para cada atributo y con la ganancia de información determina el atributo de prueba.

DTFS
Entrada:
CE: Conjunto de entrenamiento
s: Número máximo de objetos almacenados en un nodo
Salida:
AD: Árbol de decisión
Paso 1: Reorganizar el CE
Paso 2: Crear nodo raíz *R*
Paso 3: Para cada objeto *O* en *CE, hacer*
ActualizaAD(O,*R*)
Fin *Para*
Paso 4: Asignar a cada hoja la clase mayoritaria de los objetos en ella
Fin DTFS

ActualizaAD(O*, NODO*)
Si número de objetos almacenados en *NODO* < *s*, entonces
Almacenar objeto *O* en *NODO*
Incrementar el número de objetos en NODO
Si número de objetos almacenados en *NODO* = *s*, entonces
ProcesaNodo(*NODO*)
Incrementar el número de objetos en NODO
Fin *Si*
Sino /* número de objetos almacenados en *NODO* > *s* */
Para cada arco R_j *en NODO, hacer*
Obtener la diferencia D_j entre los valores de *O* y los del R_j
Fin *Para*
Hijo = Nodo ligado al arco con la menor diferencia *D*
ActualizaAD(*O, Hijo*)
Fin *Si*
Fin *ActualizaAD*

Figura 6.1: Algoritmo DTFS.

ProcesaNodo(*NODO*)
Si NODO tiene objetos de más de una clase, entonces
Para cada atributo X_i en *CE, hacer*
Para cada clase C_j en *NODO, hacer*
Obtener la media M_{ij} entre los valores de los objetos en *NODO* de clase C_j
Fin *Para*
Obtener la proporción de la ganancia de información de X_i usando M_i
Fin *Para*
Elegir el atributo *i* con la mayor ganancia como atributo de prueba
Para cada clase C_j en *NODO, hacer*
Crear un arco R_j
Asignar la media M_{ij} al arco R_j
Crear nodo asociado a R_j
Fin *Para*
Eliminar los objetos almacenados en el nodo expandido
Sino
Para cada atributo A_i del CE, *hacer*
Obtener la media M_i entre los valores de los objetos en *NODO*
Promediar M_i con la media del atributo A_i asociada al arco de entrada de *NODO*
Fin *Para*
Eliminar los objetos almacenados en el nodo actualizado
Fin *Si*
Fin *ProcesaNodo*

Figura 6.2: Procesamiento de un nodo con s objetos con el algoritmo DTFS.

6.3. Recorrido del AD

El recorrido que sigue un objeto en un AD construido por DTFS se inicia en el nodo raíz, descendiendo entre los nodos internos hasta llegar a una hoja. Al igual que en IIMDT e IIMDTS, los objetos descienden de un nodo a otro, eligiendo el arco que tenga la menor diferencia entre el valor del atributo de prueba en el objeto que está recorriendo el AD y el valor de prueba que está asociado a cada arco del nodo por el cual se va a descender.

6.4. Análisis de complejidad del algoritmo DTFS

El análisis de la complejidad temporal de DTFS se llevó a cabo teniendo en cuenta los dos pasos que se realizan en la construcción del AD, el recorrido del AD con todos los objetos de entrenamiento y la expansión de todos los nodos que tengan almacenados s objetos.

Dado un conjunto de entrenamiento de m objetos, dividos en k clases y descritos por x atributos, para la complejidad temporal del algoritmo DTFS se puede observar que el recorrido que siguen los m objetos de entrenamiento, en el peor caso, es el mismo que se sigue en IIMDTS, $O(m^2)$. Por su parte, en el proceso de expansión de un nodo, primero se calcula el conjunto de valores de prueba de cada atributo, esto es $O(k * (s/k)) = O(s)$ y posteriormente se aplica la Proporción de la Ganancia de Información a cada atributo utilizando sólo el conjunto de valores de prueba calculado para el atributo, esto es $O(x)$. Entonces expandir un nodo es $O(s + x)$, y dado que el número máximo de expansiones es $O(m/s)$, la complejidad temporal del total de expansiones en DTFS es $O((s+x)*(m/s)) = O((s*m/s)+(x*m/s)) = O(m+((x*m)/s)) = O(m)$. Lo que resulta en un orden de complejidad igual al de IIMDTS.

Para la complejidad espacial de DTFS, se toma en cuenta el número máximo de expansiones que se pueden realizar, el cual está determinado, como en IIMDT, por el parámetro s, entonces la complejidad espacial de DTFS es de igual orden que la de IIMDT.

6.5. Resultados experimentales

La primera parte de esta sección, muestra un estudio experimental del parámetro s para analizar el comportamiendo de nuestro algoritmo al variar

el valor de este parámetro. La segunda parte de la sección muestra experimentos para analizar el comportamiento de DTFS cuando varía el número de objetos en el conjunto de entrenamiento. Posteriormente, se muestra el comportamiento de DTFS cuando se incrementa el número de atributos en el conjunto de entrenamiento. Finalmente, se presentan los resultados del uso de memoria de los algoritmos DTFS, ICE, VFDT e IIMDTS.

Las especificaciones para realizar los experimentos y los conjuntos de datos utilizados son los descritos en la Sección 4.5.

6.5.1 Parámetro s del algoritmo DTFS

Para estudiar el comportamiento de nuestro algoritmo cuando varía el valor del parámetro s, se crearon conjuntos de entrenamiento de 100,000 objetos de cada conjunto de datos real de la Tabla 4.1. Los valores evaluados para s fueron 50 y de 100 a 600 con incrementos de 100.

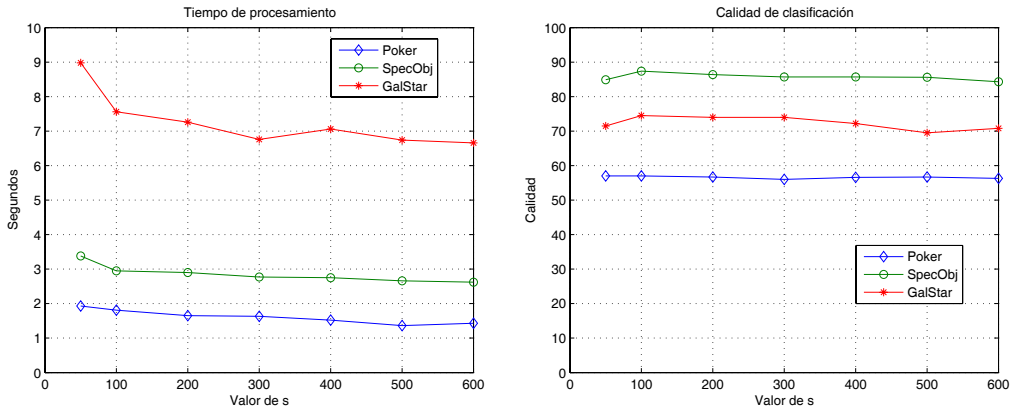


Figura 6.3: Tiempo de procesamiento y calidad de clasificación para DTFS variando s.

La Figura 6.3 muestra el tiempo de procesamiento y la calidad de clasificación obtenida con el algoritmo DTFS cuando varía el valor del parámetro s. Como se puede observar, el tiempo de procesamiento y la calidad de clasificación disminuyen poco cuando el valor de s aumenta. De acuerdo a estos experimentos, se decidió elegir el valor de $s = 100$ para los siguientes experimentos, ya que con este valor se obtuvó la mejor calidad de clasificación. En cuanto al tiempo de procesamiento no hay mucha diferencia con los tiempos de procesamiento que obtuvieron los otros

valores. No se eligió el valor de $s = 600$ que es con el que DTFS obtiene el menor tiempo de procesamiento, ya que con este valor la calidad de clasificación disminuye.

6.5.2 Incrementando el número de objetos en el conjunto de entrenamiento

Esta sección está divida en dos partes, la primera muestra los resultados obtenidos con los conjuntos de datos reales y la segunda los obtenidos con los conjuntos de datos sintéticos de 5 y 40 atributos (conjuntos de datos descritos en la Tabla 4.1).

6.5.2.1 Conjuntos de datos reales

Para cada conjunto de datos real se utilizaron los conjuntos de entrenamiento descritos en la Sección 4.5.4.1. La Figura 6.4 muestra los resultados obtenidos con Poker. Como puede observarse, DTFS, VFDT e IIMDTS son mejores que ICE y BOAI, ya que son los que menor tiempo de procesamiento emplean para la construcción del AD, mientras que todos los algoritmos obtienen una calidad de clasificación similar. Para mostrar más claramente la diferencia entre DTFS, VFDT e IIMDTS, se graficaron los resultados de estos algoritmos en la Figura 6.5, en donde se puede observar que DTFS es más rápido que VFDT e IIMDTS, aún cuando IIMDTS elige sólo un atributo de prueba en sus nodos internos (IIMDTS-1 en la figura).

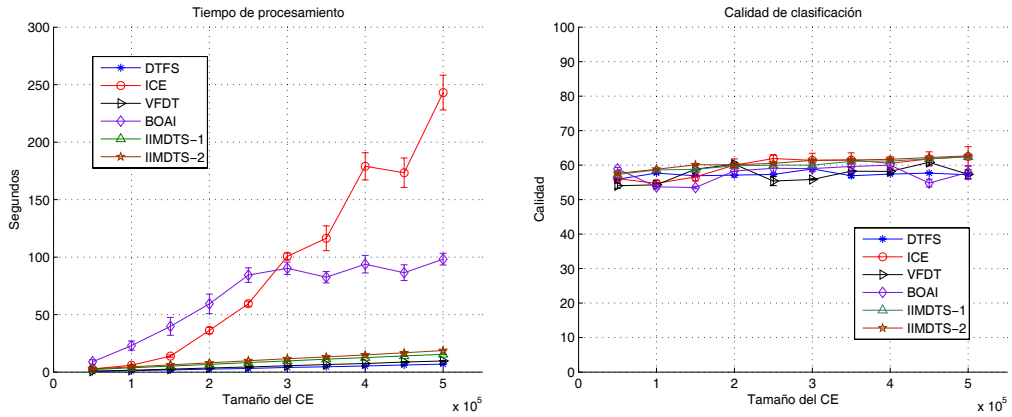


Figura 6.4: Tiempo de procesamiento y calidad de clasificación para Poker.

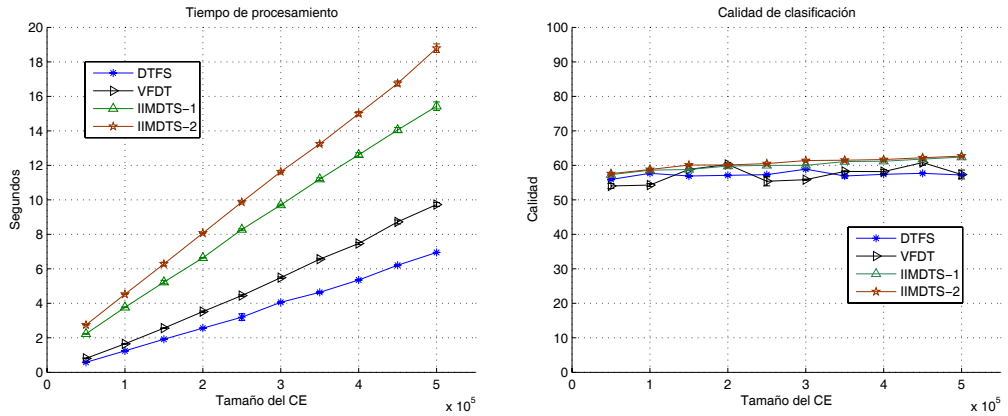


Figura 6.5: Tiempo de procesamiento y calidad de clasificación para Poker con DTFS, VFDT e IIMDTS.

Con Spec, DTFS obtuvo similares resultados a IIMDTS-1, y ambos algoritmos fueron los mejores para este conjunto de datos. Los dos tuvieron casi el mismo tiempo de procesamiento y similar calidad de clasificación. Estos resultados pueden observarse en la Figura 6.6.

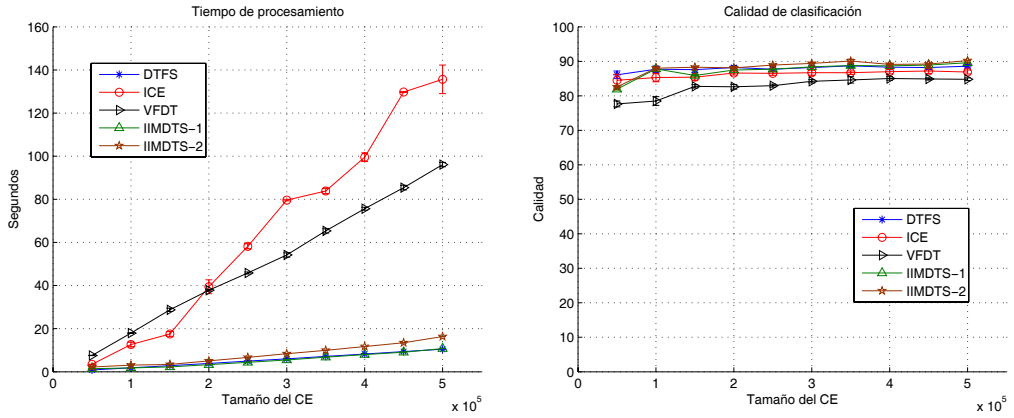


Figura 6.6: Tiempo de procesamiento y calidad de clasificación para SpecObj.

Los resultados obtenidos con GalStar se muestran en la Figura 6.7. Como puede observarse DTFS e IIMDTS son los que menor tiempo de procesamiento emplean y la calidad de clasificación de todos los algoritmos es muy similar.

Sin embargo, en la Figura 6.7 no se puede apreciar la diferencia entre DTFS, VFDT e IIMDTS, debido a que el tiempo de ICE es 57 veces mayor que el de DTFS. Por lo tanto, la Figura 6.8 muestra los resultados excluyendo a ICE.

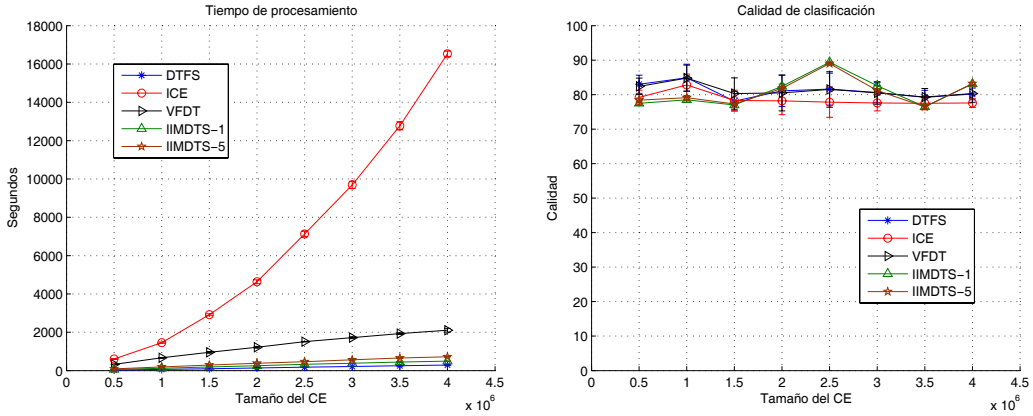


Figura 6.7: Tiempo de procesamiento y calidad de clasificación para GalStar.

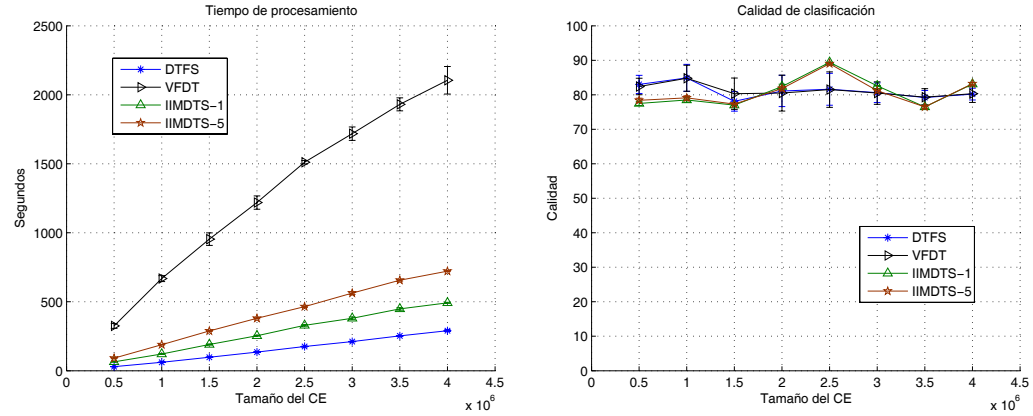


Figura 6.8: Tiempo de procesamiento y calidad de clasificación para GalStar excluyendo a ICE.

En la Figura 6.8 se puede observar que DTFS es el algoritmo más rápido. DTFS es 2, 2.5 y 8.5 veces más rápido que IIMDTS-1, IIMDTS-5 y VFDT, respectivamente. Con respecto a la calidad de clasificación, todos los algoritmos tienen una calidad similar.

6.5.2.2 Conjuntos de datos sintéticos

Utilizando los conjuntos de entrenamiento creados anteriormente para los conjuntos de datos sintéticos, se realizaron los siguientes experimentos con la finalidad de analizar el comportamiento de DTFS cuando se incrementa el número de objetos en el conjunto de entrenamiento. Los resultados con los conjuntos de entrenamiento de 5 atributos se muestran en la Figura 6.9. Como puede observarse, para los tres conjuntos de entrenamiento (de 2, 3 y 5 clases), DTFS fue más rápido que ICE y VFDT, sin embargo IIMDTS fue un poco más rápido que DTFS. En cuanto a la calidad de clasificación, con los conjuntos de entrenamiento de 2 clases, todos los algoritmos obtuvieron una calidad similar, a excepción de BOAI que obtuvo una calidad inferior. Con los conjuntos de entrenamiento de 3 y 5 clases, la calidad de clasificación de DTFS, VFDT e IIMDTS fue superior a la de ICE.

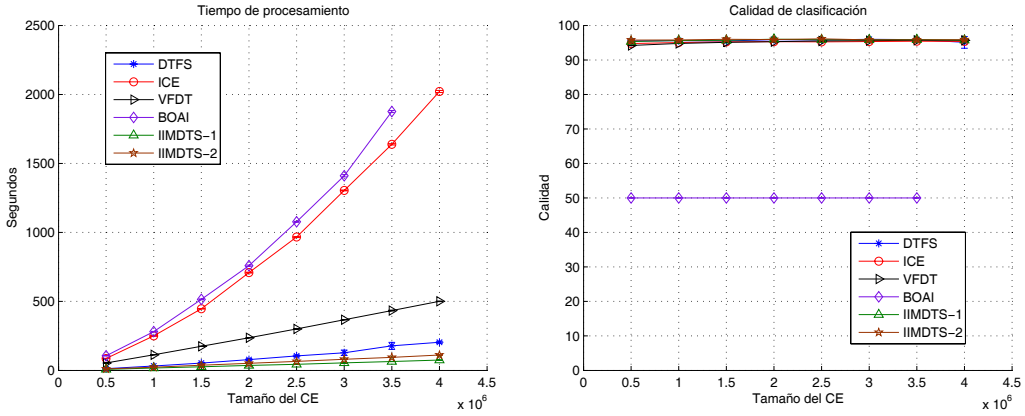
La Figura 6.10 presenta los resultados obtenidos con los conjuntos de entrenamiento de 40 atributos con 2, 3 y 5 clases. Como puede observarse, DTFS fue más rápido que ICE y VFDT en todos los casos. En estos experimentos, el tiempo empleado por DTFS fue similar al empleado por IIMDTS. Respecto a la calidad de clasificación, DTFS obtuvo resultados similares al resto de los algoritmos con los conjuntos de entrenamiento de 2 clases, y fue superior a ICE en los conjuntos de entrenamiento de 3 y 5 clases.

A partir de los experimentos realizados con los conjuntos de datos reales y los conjuntos de datos sintéticos, cuando se incrementa el número de objetos en el conjunto de entrenamiento, se puede observar que DTFS mantiene un comportamiento estable. El tiempo de procesamiento de DTFS no se incrementa mucho cuando varía el número de objetos de entrenamiento y la calidad de clasificación que obtiene es muy similar en todos los casos.

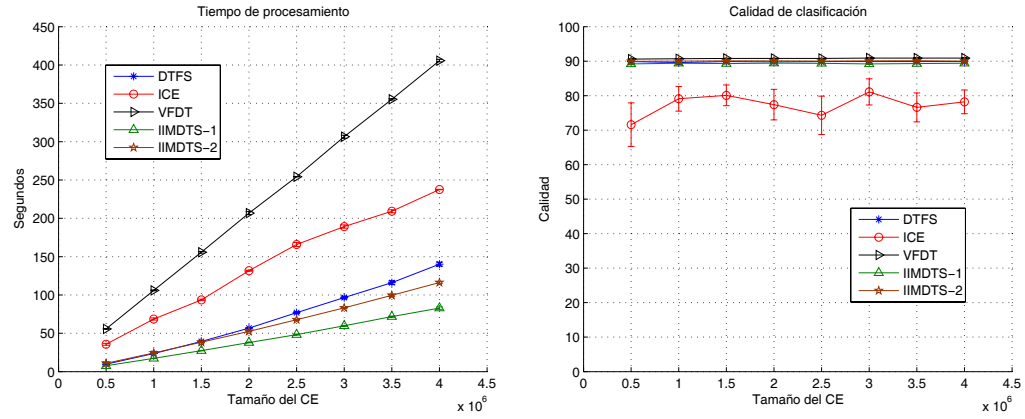
6.5.3 Incrementando el número de atributos en el conjunto de entrenamiento

Para realizar este experimento se utilizaron los conjuntos de datos sintéticos descritos anteriormente (de 5 a 40 atributos con incrementos de 5 atributos, de 2, 3 y 5 clases cada uno).

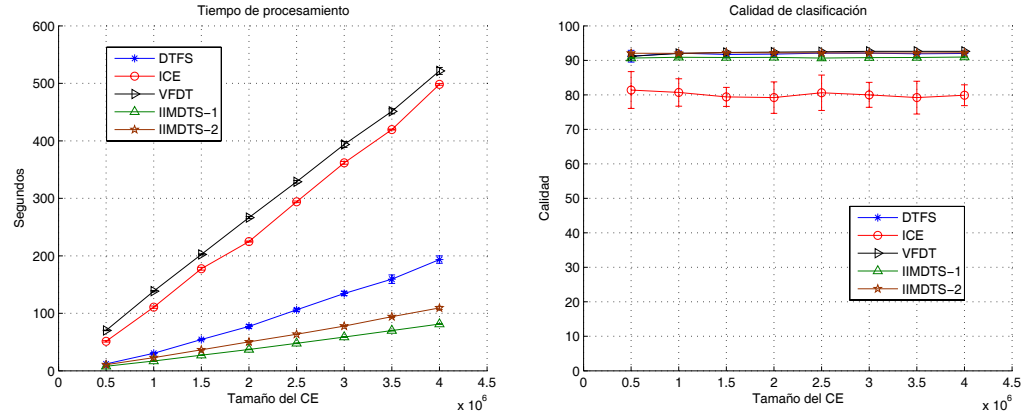
La Figura 6.11 muestra el desempeño para los conjuntos de entrenamiento sintéticos de 2, 3 y 5 clases con los algoritmos DTFS, ICE, VFDT e IIMDTS cuando se incrementa el número de atributos en el conjunto de entrenamiento.



(a) 2 clases

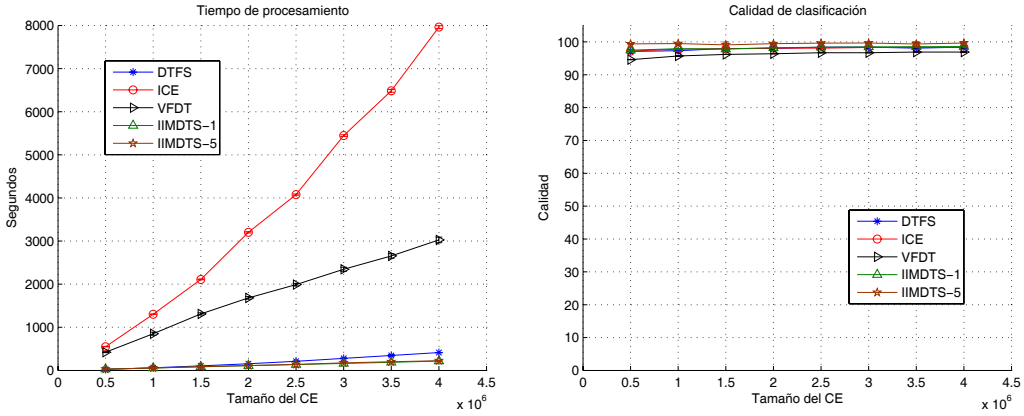


(b) 3 clases

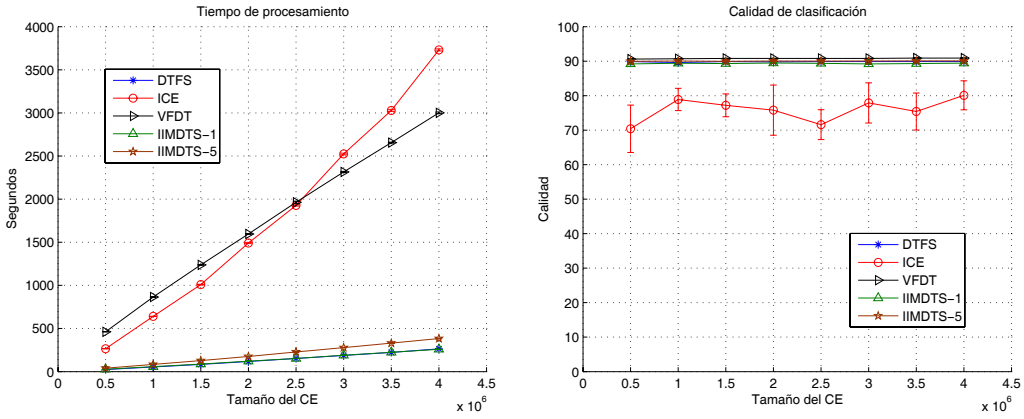


(c) 5 clases

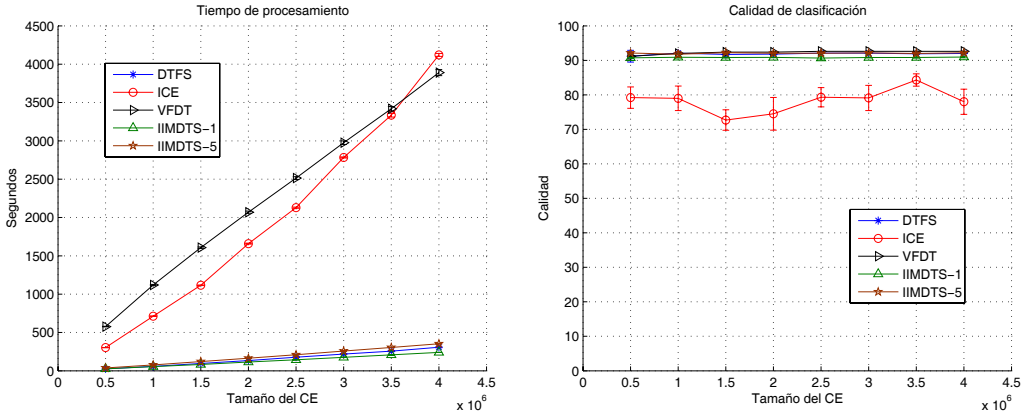
Figura 6.9: Tiempo de procesamiento y calidad de clasificación para DTFS, ICE, VFDT, BOAI e IIMDTS con conjuntos de entrenamiento sintéticos de 5 atributos y de 2, 3 y 5 clases.



(a) 2 clases



(b) 3 clases



(c) 5 clases

Figura 6.10: Tiempo de procesamiento y calidad de clasificación para DTFS, ICE, VFDT e IIMDTS con conjuntos de entrenamiento sintéticos de 40 atributos y de 2, 3 y 5 clases.

BOAI no se muestra en estos resultados ya que no pudo procesar ningún conjunto de entrenamiento de este tamaño.

Como puede observarse en la Figura 6.11, el tiempo de procesamiento de DTFS e IIMDTS se conserva similar cuando se incrementa el número de atributos en el conjunto de entrenamiento. Por su parte, ICE y VFDT incrementan su tiempo de procesamiento conforme se incrementa el número de atributos. En cuanto a la calidad de clasificación, independientemente del número de atributos, DTFS, VFDT e IIMDTS obtienen similares resultados, siendo los tres algoritmos superiores a ICE.

6.5.4 Uso de memoria del algoritmo DTFS

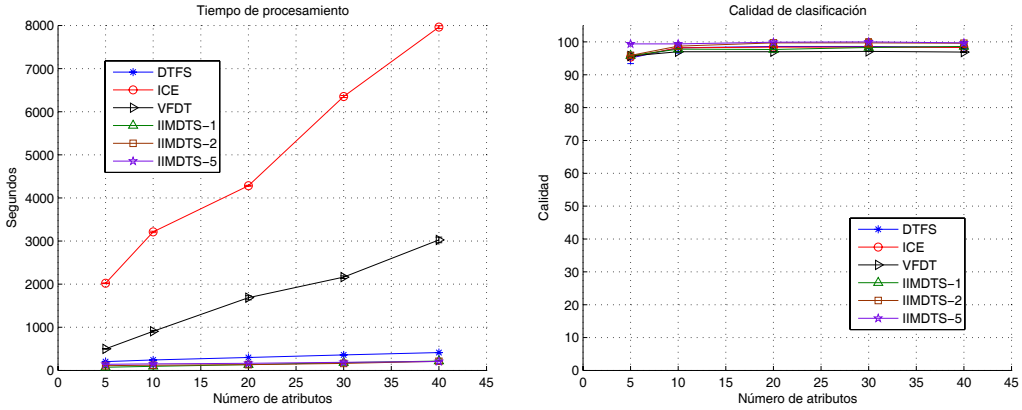
Estos experimentos se realizaron con el objetivo de mostrar la diferencia en uso de memoria que existe entre ICE, VFDT, IIMDTS y DTFS, ya que la complejidad espacial de los cuatro es del mismo orden. BOAI no se muestra en estos experimentos ya que, como se mencionó anteriormente, este algoritmo presentó fallas de memoria cuando el conjunto de entrenamiento a procesar tenia una gran cantidad de objetos.

La Figura 6.12 muestra los resultados obtenidos con DTFS, ICE, VFDT e IIMDTS con los tres conjuntos de datos reales descritos en la Tabla 4.1, utilizando la herramienta *memory* [Gar09] de Linux para obtener la cantidad de memoria que emplea cada algoritmo.

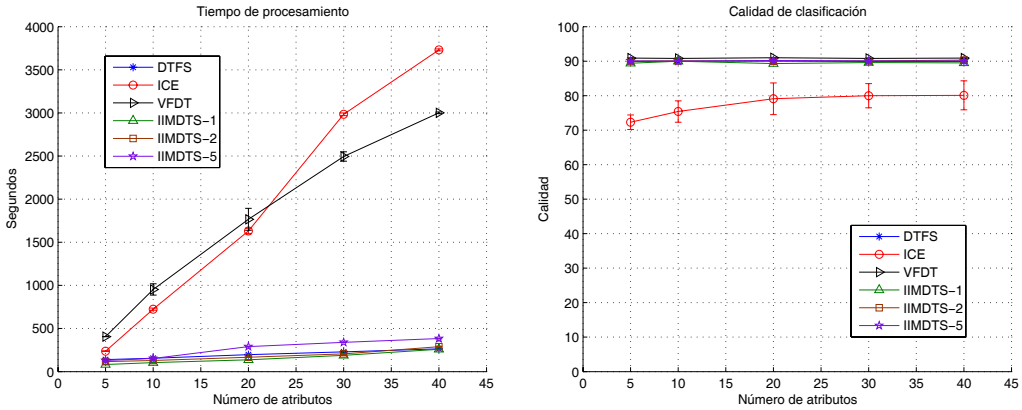
De la Figura 6.12 se puede observar que DTFS utiliza menos memoria que VFDT (el más rápido de los algoritmos del estado del arte), aún cuando ambos algoritmos generan ADs univaluados. Además, DTFS utiliza similar cantidad de memoria que IIMDTS y un poco más que ICE, sin embargo ICE no utiliza todo el conjunto de entrenamiento para generar el AD.

6.6. Discusión

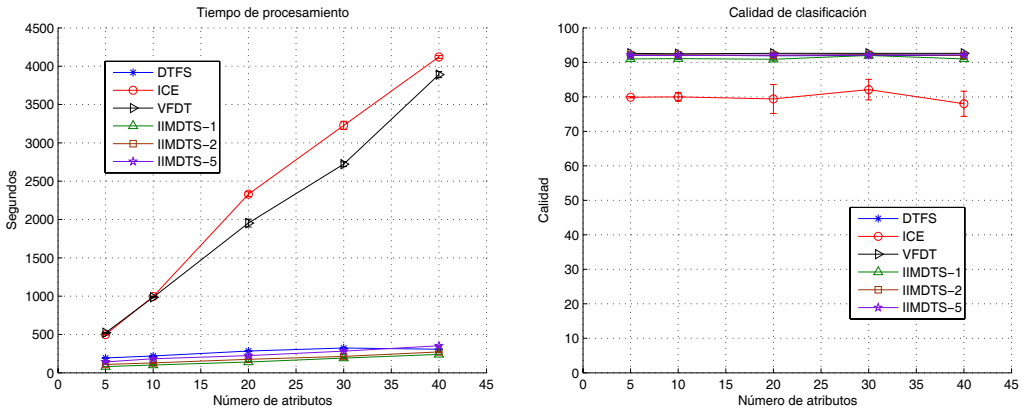
En este capítulo se propuso el algoritmo DTFS, el cual construye ADs univaluados para grandes conjuntos de datos numéricos. Este algoritmo obtiene buenos resultados independientemente del número de objetos y atributos en el conjunto de entrenamiento. Respecto al tiempo de procesamiento que emplea para la construcción del AD, de acuerdo a los experimentos cuando se incrementa el número de objetos en el conjunto de entrenamiento, DTFS es,



(a) 2 clases

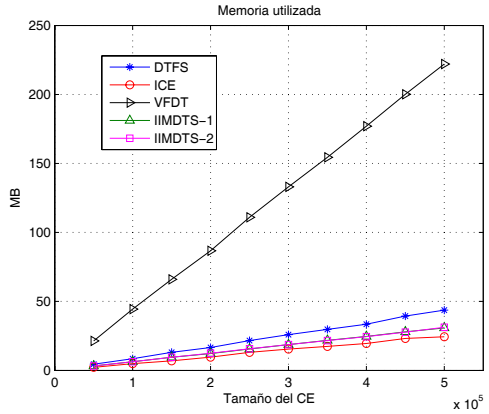


(b) 3 clases

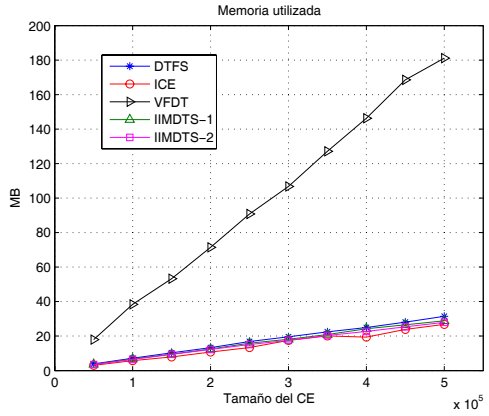


(c) 5 clases

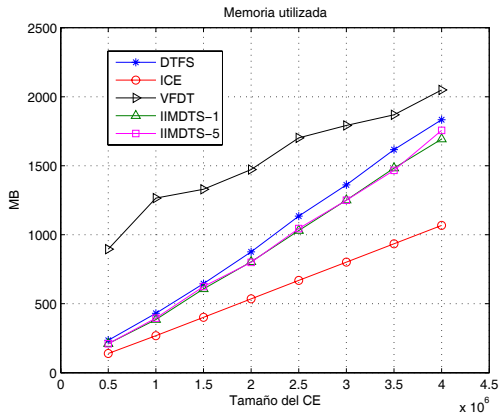
Figura 6.11: Tiempo de procesamiento y calidad de clasificación para DTFS, ICE, VFDT e IIMDTS con conjuntos de entrenamiento sintéticos de 2, 3 y 5 clases, cuando el número de atributos se incrementa.



(a) 2 clases



(b) 3 clases



(c) 5 clases

Figura 6.12: Cantidad de memoria utilizada por DTFS, ICE, VFDT e IIMDTS con Poker, SpecObj y GalStar.

en promedio, hasta 21.53, 7.35 y 19.01 veces más rápido que ICE, VFDT y BOAI (los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos), respectivamente. Por su parte, cuando se incrementa el número de atributos en el conjunto de entrenamiento, DTFS es hasta 16.03 y 10.10 veces más rápido que ICE y VFDT, respectivamente. Con respecto a la calidad de clasificación, DTFS obtiene resultados similares a los obtenidos por ICE, VFDT y BOAI, independientemente del número de atributos y de objetos en el conjunto de entrenamiento.

En cuando al parámetro utilizado por DTFS, se observó que utilizando valores pequeños se obtienen buenos resultados. Con el valor de $s = 100$, utilizado en los experimentos con DTFS, es posible obtener buenos resultados, ya que el tiempo de procesamiento empleado por nuestro algoritmo fue menor con este valor que con valores más grandes y la calidad de clasificación fue similar para todos los valores de s. Con respecto a la memoria utilizada por nuestro algoritmo, se pudo observar que utiliza menor cantidad de memoria que VFDT, el más rápido de los algoritmos previos, construyendo ambos algoritmos ADs univaluados de calidad similar, utilizando todo el conjunto de entrenamiento.

Comparando los resultados obtenidos con DTFS contra los obtenidos con IIMDTS (cuando éste último elige subconjuntos de 1 atributo en los nodos internos), se observó que la diferencia entre los tiempos de procesamiento de ambos algoritmos es pequeña, mientras que la calidad de clasificación obtenida con los dos algoritmos es similar.

Capítulo 7

ADs univaluados para grandes conjuntos de datos mezclados

En este capítulo se introduce un algoritmo de generación de ADs univaluados para grandes conjuntos de datos, descritos simultáneamente por atributos numéricos y no númericos (datos mezclados).

La primera idea que se utilizó para resolver este problema, fue adaptar los algoritmos que generan ADs, propuestos en los Capítulos 4, 5 y 6, para que permitieran el manejo de datos mezclados. Para obtener la combinación de valores de prueba asociada a un arco de salida cuando un nodo se expande, se aplicó la moda y la mediana. Para comparar los objetos al recorrer el AD se utilizaron las funciones de comparación para datos mezclados D [JMTAM02] y HVDM [WM00]. Sin embargo, con estas modificaciones los resultados obtenidos no fueron satisfactorios, dado que la calidad de clasificación fue inferior que la de los algoritmos utilizados para comparar.

Por esta razón, se desarrolló otro algoritmo para construir ADs para grandes conjuntos de datos mezclados, DTLT (por sus siglas en inglés, *Decision Trees for Large Training sets*). Este algoritmo construye ADs univaluados y al igual que los algoritmos propuestos en los capítulos anteriores procesa todo el conjunto de entrenamiento de manera incremental sin almacenarlo completo en memoria.

La estructura del AD construido es igual a la estructura de un AD construido por un algoritmo de generación de ADs tradicional, es decir, el AD tiene nodos internos y terminales (hojas). Cada nodo interno tiene un atributo de prueba, si el atributo de prueba es numérico, tendrá dos arcos de salida. Por el contrario, si el atributo de prueba es no numérico, entonces el número de arcos de salida dependerá del número de valores diferentes que

pueda tomar el atributo. Por su parte, cada hoja tendrá asociada la etiqueta de una clase.

7.1. Selección del atributo de prueba

La selección del atributo de prueba se hace utilizando sólo los objetos almacenados en el nodo. Debido a que se trata de grandes conjuntos de datos es común que exista mucha redundancia en los datos, por lo tanto es posible utilizar sólo los objetos almacenados para seleccionar el atributo de prueba. Además utilizar pocos objetos para seleccionar el atributo de prueba permite que la selección sea rápida.

La selección del atributo de prueba consiste en aplicar la Proporción de Ganancia de Información a cada atributo para elegir el atributo que divida lo más homogeneamente posible a los objetos almacenados en el nodo, es decir, aquel con mayor ganancia de información. La ganancia de información de un atributo numérico se obtiene como en el algoritmo C4.5 [Qui93], es decir, se ordenan los valores del atributo, se obtiene un punto de corte por cada par de valores adyacentes (que sean diferentes y de distinta clase) promediando ambos valores, y con cada punto de corte se evalúa la ganancia de información del atributo. El punto de corte que mejor ganancia obtenga para ese atributo, será elegido para crear los arcos de salida del nodo, en caso de que ese atributo sea el de mayor ganancia.

En el caso de los atributos no numéricos, también se obtiene la ganancia de información como en el algoritmo C4.5 [Qui93], es decir, la ganancia se evalúa utilizando los valores diferentes que puede tomar el atributo. Si un atributo no numérico es el que tiene la mayor ganancia de información, los valores diferentes del atributo serán utilizados para crear los arcos de salida del nodo.

7.2. Construcción del AD

La construcción del AD se inicia con un nodo raíz vacío y sin descendientes (nodo hoja). Posteriormente, se procesan, de manera incremental, uno a uno los objetos de entrenamiento. Dado que DTLT procesa los objetos de entrenamiento de manera incremental (así como lo hacen los algoritmos de los capítulos anteriores), con el objetivo de permitir diversidad en los nodos hoja, el preprocesamiento del conjunto de entrenamiento, descrito en la Sección 4.1, también es aplicado por DTLT.

Al inicio del algoritmo como sólo se tiene un nodo (el nodo raíz), los primeros objetos procesados serán almacenados en ese nodo. Cuando un nodo tenga s objetos almacenados (s es un parámetro del algoritmo), si los objetos pertenecen a dos o más clases se expande el nodo, pero si pertenecen a una misma clase, el nodo se queda como nodo hoja y los objetos almacenados en él son eliminados.

Cuando un nodo tiene objetos pertenecientes a dos o más clases, el nodo se expande de la siguiente manera. Primero, se elige el atributo de prueba de acuerdo al criterio explicado en la Sección 7.1. Si es un atributo de prueba numérico, entonces junto con él se eligió el punto de corte que mejor separa a los objetos almacenados en el nodo, y se crean dos arcos de salida para el nodo, cada uno ligado a un nuevo nodo vacío. Por el primer arco descenderán los objetos que tengan en el atributo de prueba un valor menor o igual al punto de corte y por el segundo arco descenderán aquellos objetos que tengan en ese atributo un valor mayor al punto de corte elegido. En caso de que sea un atributo de prueba no numérico, entonces se crean tantos arcos de salida como valores diferentes pueda tomar el atributo, y cada arco de salida es ligado a un nuevo nodo vacío. Por cada arco descenderán los objetos que tengan en el atributo de prueba el valor con el cual se creó el arco. Esta forma de crear los arcos de salida es igual a la utilizada en el algoritmo C4.5 [Qui93].

Una vez que se ha elegido el atributo de prueba para el nodo a expandir y se han creado sus arcos de salida, cada objeto almacenado en el nodo se coloca en los nuevos nodos hijos, de acuerdo al valor que tenga el objeto en el atributo de prueba. A cada nuevo nodo (ligado a uno de los arcos de salida del nodo que se expandió) se le asigna una etiqueta de clase temporal (la cual será utilizada al finalizar la construcción del AD, en caso de que la hoja quede vacía). La etiqueta de clase temporal de cada hoja será la clase mayoritaria de los objetos que correspondan a esa hoja. Si la hoja no recibió ningún objeto, entonces la clase temporal será la clase mayoritaria de los objetos usados para expandir el nodo. Después, los s objetos utilizados para la expansión son eliminados, con la finalidad de que sólo sean procesados una vez y además liberar espacio para recibir más objetos.

La Figura 7.1 muestra la expansión de un nodo, cuando tiene almacenados s objetos. Como se puede observar, el nodo contiene objetos de más de una clase, entonces es expandido. Primero se aplica la Proporción de Ganancia de Información a cada atributo para elegir el atributo de prueba X. Si X es numérico, entonces junto con él se elegirá el punto de corte (PC) que mejor divida a los objetos almacenados en el nodo, y se crean dos arcos ligados a

un nodo vacío. Pero si X es no numérico, entonces se crean a arcos (cada uno ligado a un nuevo nodo vacío), suponiendo que X puede tomar a valores diferentes. A cada nodo hijo creado se le asigna una etiqueta de clase temporal, la cual será la clase mayoritaria de los objetos que están almacenados en el nodo padre y que siguen el arco ligado al nodo hijo. En el ejemplo, para el arco con el valor de prueba $V_X \leq PC$, los objetos almacenados en el nodo que siguieron ese arco, la mayoría fueron de la clase triangulo, es por esto que ésta es la clase asignada como clase temporal.

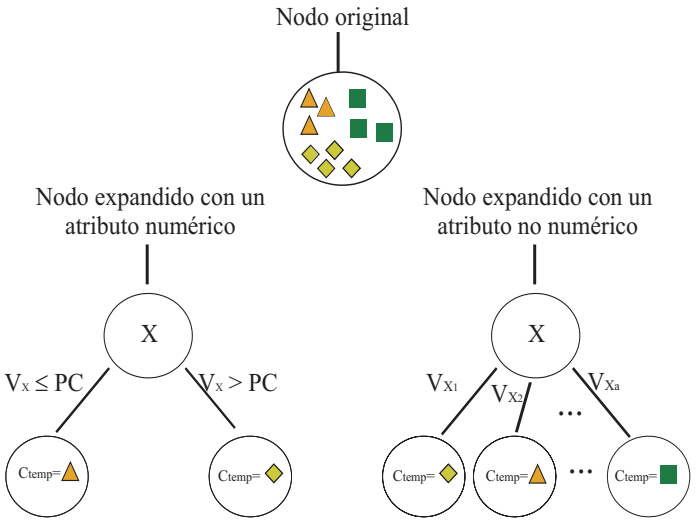


Figura 7.1: Ejemplo de expansión de un nodo con DTLT.

Por otro lado, si un nodo tiene s objetos que pertenezcan a una sola clase, entonces se tiene un nodo homogéneo, por lo cual no es necesario expandirlo, simplemente se eliminan los objetos que están almacenados en el nodo, de tal manera que sea nuevamente un nodo vacío que pueda recibir otros s objetos.

La construcción del AD termina cuando todos los objetos de entrenamiento se han procesado. Entonces se asigna a cada hoja del AD, la clase mayoritaria de los objetos que quedaron almacenados en ella. Si la hoja está vacía, entonces se le asigna, como etiqueta de clase, la clase temporal.

La función para la construcción del AD con DTLT se muestra en la Figura 7.2 y la función para procesar un nodo, cuando tiene s objetos almacenados, se muestra en la Figura 7.1. Como se puede observar en la Figura 7.2, la función para la construcción de un AD con el algoritmo DTLT es similar a

la de los algoritmos propuestos en los Capítulos 4, 5 y 6. Por su parte, en la Figura 7.3 se puede observar que cuando un nodo tiene s objetos, y si éstos pertenecen a dos o más clases, entonces el nodo es expandido, eligiendo un atributo de prueba y creando un conjunto de nodos hijos, el número de nodos a crear dependerá del tipo de atributo de prueba seleccionado.

```
DTLT
   Entrada:
      CE: Conjunto de entrenamiento
      s: Número máximo de objetos almacenados en un nodo
   Salida:
      AD: Árbol de decisión
   Paso 1: Reorganizar el CE
   Paso 2: Crear nodo raíz R
   Paso 3: Para cada objeto O en CE, hacer
                ActualizaAD(O,R)
           Fin Para
   Paso 4: Asignar a cada hoja la clase mayoritaria de los objetos en ella
Fin DTLT


ActualizaAD(O, NODO)
   Si número de objetos almacenados en NODO < s, entonces
      Almacenar objeto O en NODO
      Incrementar el número de objetos en NODO
      Si número de objetos almacenados en NODO = s, entonces
         ProcesaNodo(NODO)
         Incrementar el número de objetos en NODO
      Fin Si
   Sino /* número de objetos almacenados en NODO > s */
      Si atributo de prueba X en NODO es no numérico
         Hijo = Nodo ligado al arco con el valor que tiene O en X
      Sino /* atributo de prueba X en NODO es numérico */
         Si valor de O en X es menor o igual al valor de prueba en NODO
            Hijo = Nodo ligado al arco izquierdo
         Sino /* valor de O en X es mayor */
            Hijo = Nodo ligado al arco derecho
      Fin Si
      ActualizaAD(O, Hijo)
   Fin Si
Fin ActualizaAD
```

Figura 7.2: Algoritmo DTLT.

```
ProcesaNodo(NODO)
    Si NODO tiene objetos de más de una clase, entonces
        Para cada atributo Xi en CE, hacer
            Obtener la proporción de la ganancia de información de Xi
        Fin Para
        Elegir el atributo X con la mayor ganancia como atributo de prueba
        Si X es numérico
            Crear un arco Rizq  /* arco izquierdo */
            Crear un arco Rder  /* arco derecho */
        Sino /* X es no numérico */
            Para cada posible valor Vj en atributo de prueba X, hacer
                Crear un arco Rj
                Asignar V j al arco Rj
            Fin Para
        Fin Si
        Para cada arco Ri en NODO, hacer
            Crear nodo asociado a Rj
        Fin Para
        Eliminar los objetos almacenados en el nodo expandido
    Sino
        Eliminar los objetos almacenados en el nodo actualizado
    Fin Si
Fin ProcesaNodo
```

Figura 7.3: Procesamiento de un nodo con s objetos en DTLT.

7.3. Recorrido del AD

Un objeto empieza a recorrer el AD desde el nodo raíz, y desciende por los nodos internos eligiendo el camino correspondiente a los valores de sus atributos. Para descender de un nodo a otro, se verifica que tipo de atributo de prueba tiene el nodo por el cual se va a descender. Si es un atributo de prueba numérico, entonces el nodo tiene dos arcos de salida. Por el arco de salida izquierdo descienden aquellos objetos que tengan en ese atributo un valor menor o igual al punto de corte, y por el arco de salida derecho, los objetos que tengan un valor mayor al punto de corte. Por otro lado, si el nodo tiene un atributo de prueba no numérico, entonces se sigue el arco de salida que tenga asociado el valor que el objeto tenga en ese atributo. El objeto desciende hasta llegar a una hoja. Como se puede observar, este recorrido es el mismo que se realiza en los ADs tradicionales.

7.4. Análisis de complejidad del algoritmo DTLT

Para analizar la complejidad de DTLT se tomaron en cuenta los procesos principales que realiza el algoritmo para la construcción del AD, recorrer el AD con cada objeto de entrenamiento y expandir los nodos que tengan s objetos almacenados.

Para un conjunto de entrenamiento de m objetos con x atributos, divididos en k clases, la complejidad del recorrido de un objeto de entrenamiento depende del número de niveles que tiene que recorrer en el AD. En un AD construido por DTLT, en el peor caso, se tienen como máximo m/s niveles, por lo tanto recorrer el AD con los m objetos es $O(m * m/s) = O(m^2)$ en el peor caso. En el caso de la expansión de un nodo, cuando tiene s objetos almacenados, se calcula la Proporción de la Ganancia de Información de cada atributo usando los s objetos del nodo, lo cual es $O(s * x)$, para poder elegir el mejor atributo para el nodo. Si tenemos que el número máximo de expansiones que se pueden hacer es $O(m/s)$ (puesto que para cada expansión se usan s objetos), entonces realizar el proceso completo de expansiones es $O(s * x * m/s) = O(x * m)$ y dado que, en grandes conjuntos de datos, comúnmente $x \ll m$ entonces, la complejidad de hacer todas la expansiones es $O(m)$.

Finalmente, la complejidad para la construcción del AD con DTLT es la suma de las complejidades de estos procesos, quedando:

$$O(m^2 + m) = O(m^2); \text{ en el peor caso.}$$

Como se puede observar la complejidad de DTLT es del mismo orden que la complejidad de los algoritmos propuestos anteriormente.

Respecto a la complejidad espacial de DTLT, el análisis se hace de acuerdo al número de nodos que tenga el AD, ya que DTLT no almacena el conjunto de entrenamiento completo. Como ya se mencionó, el número máximo de expansiones que hace DTLT es $O(m/s)$, y cada una produce a lo más a^M nuevos nodos (siendo a^M el número de valores diferentes del atributo que pueda tomar el mayor número de valores diferentes) con lo cual el número máximo de nodos es $O(a^M * m/s)$, y dado que en grandes conjuntos de datos $s \ll m$ y $a^M \ll m$, entonces la complejidad espacial de DTLT es $O(m)$. Siendo esta complejidad del mismo orden que la de los otros algoritmos de generación de ADs propuestos.

7.5. Resultados experimentales

En esta sección, primero se presenta un estudio experimental del parámetro s, para analizar el comportamiento de nuestro algoritmo cuando varía el valor de este parámetro. Luego, se presentan experimentos variando el número de objetos y atributos en el conjunto de entrenamiento, incluyendo una comparación contra los tres algoritmos más recientes para generación de ADs a partir de grandes conjuntos de datos: ICE, VFDT y BOAI (la descripción de cómo se usaron estos algoritmos se presentó en la Sección 4.5.2). Posteriormente, se muestran experimentos para comparar la cantidad de memoria que los algoritmos utilizan para la construcción de los ADs, con la finalidad de analizar la diferencia en la cantidad de memoria usada por cada algoritmo, debido a que, como se mostró en la Sección 7.4, el orden de complejidad espacial de DTLT, ICE, VFDT y BOAI es el mismo. Finalmente, se analizó el comportamiento de DTLT con grandes conjuntos de datos numéricos, con el fin de comparar a DTLT con los algoritmos propuestos anteriormente. Los experimentos se realizaron con las especificaciones descritas en la Sección 4.5.

7.5.1 Conjuntos de datos

La Tabla 7.1 describe los conjuntos de datos mezclados utilizados para realizar los experimentos con DTLT. Forest CoverType es un conjunto de datos real obtenido del Repositorio UCI [AN07]. KDD es un conjunto de

datos utilizado en la Copa KDD de 1999 [Arc99] y el resto de los conjuntos de datos fueron creados utilizando el generador de datos propuesto por Agrawal [RAS93].

Tabla 7.1: Conjuntos de datos mezclados usados en los experimentos

Conjunto	# Clases	# Objetos	# Atr. Num.	# Atr. No Num.
Forest CoverType	7	581,012	10	44
KDD	2	4,800,000	34	7
Agrawal F1	2	6,500,000	6	3
Agrawal F2	2	6,500,000	6	3
Agrawal F7	2	6,500,000	6	3

El generador de conjuntos de datos propuesto por Agrawal crea conjuntos de datos descritos por 9 atributos y 2 clases. Para crear un conjunto de datos, el generador se basa en una función que determina la clase a la que debe pertenecer cada objeto de entrenamiento. El generador tiene 10 funciones [RAS93] disponibles para generar un conjunto de datos. Las funciones que se utilizaron para generar los conjuntos de datos de la Tabla 7.1 fueron la 1, 2 y 7. Se eligieron estas funciones ya que se han utilizado para realizar experimentos con diversos algoritmos de generación de ADs para grandes conjuntos de datos [SAM96, GRG00, GGRL99, YAR99, YWYC08].

7.5.2 Parámetro s del algoritmo DTLT

En esta sección se realizó un estudio experimental para observar el desempeño de nuestro algoritmo cuando el valor del paramétro s varía. Para cada conjunto de datos de la Tabla 7.5 se creó un conjunto de entrenamiento. Con Forest CoverType se creó un conjunto de entrenamiento de 50,000 objetos, con KDD un conjunto de entrenamiento de 500,000 objetos y para cada conjunto de datos de Agrawal un conjunto de entrenamiento de 200,000 objetos. Se evaluaron diversos valores para s, 50 y de 100 a 600 con incrementos de 100. La Figura 7.4 muestra los resultados de calidad de clasificación y tiempo de procesamiento.

Como puede observarse en la Figura 7.4, conforme se incrementa el valor de s, el tiempo de procesamiento de DTLT se incrementa. Con respecto a la calidad de clasificación, se puede notar que con todos los conjuntos de

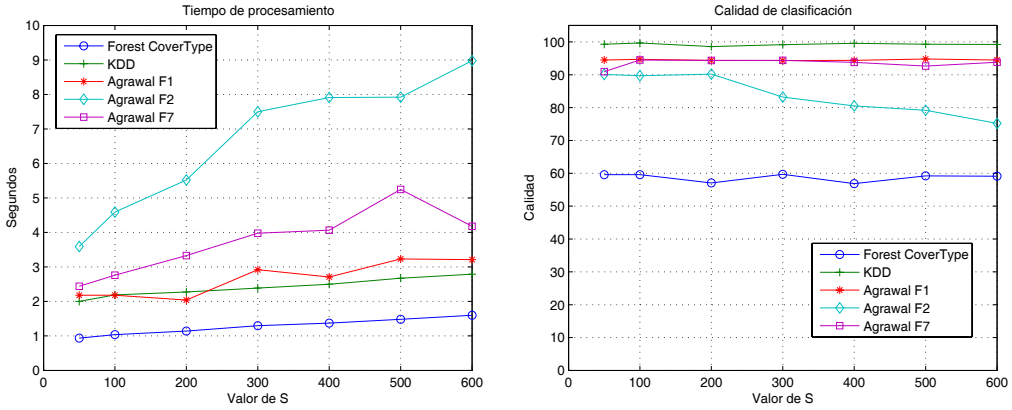


Figura 7.4: Tiempo de procesamiento y calidad de clasificación para DTLT variando s.

datos se obtuvo una calidad similar independientemente del valor de s, con excepción de los resultados obtenidos con el conjunto de datos Agrawal F2, ya que la calidad de clasificación disminuyó conforme se incrementó el valor de s. A partir de estos resultados, se recomienda utilizar valores pequeños para el parámetro s. En los siguientes experimentos, se decidió utilizar el valor de $s = 100$, ya que con este valor se mantiene un buen balance entre el tiempo de procesamiento y la calidad de clasificación. No se eligió $s = 50$ porque con este valor la calidad era un poco menor que con el valor de 100, en la mayoría de los conjuntos de datos.

7.5.3 Incrementando el número de objetos en el conjunto de entrenamiento

En esta sección se muestra el desempeño de nuestro algoritmo y una comparación contra los algoritmos ICE, VFDT y BOAI, cuando se incrementa el número de objetos de entrenamiento. Se realizaron experimentos creando diversos conjuntos de entrenamiento con los conjuntos de datos descritos en la Tabla 7.5, para evaluar el desempeño (incluyendo el tiempo de procesamiento y la calidad de clasificación) de DTLT cuando se incrementa el número de objetos en el conjunto de entrenamiento.

Para Forest CoverType se crearon conjuntos de entrenamiento de 50,000 a 550,000 objetos con incrementos de 50,000 objetos. La Figura 7.5 muestra los resultados con los algoritmos DTLT, ICE y VFDT. BOAI no se muestra en estos resultados, ya que es un algoritmo para problemas de sólo dos clases.

Como puede observarse, DTLT es el mejor algoritmo, ya que es hasta 4.5 y 31.5 veces más rápido que VFDT e ICE, respectivamente. Con respecto a la calidad de clasificación, los tres algoritmos obtienen resultados similares, ya que aunque los promedios de los resultados obtenidos por los algoritmos no son tan similares, los intervalos de confianza son amplios, por lo cual los resultados de los algoritmos se intersectan entre sí.

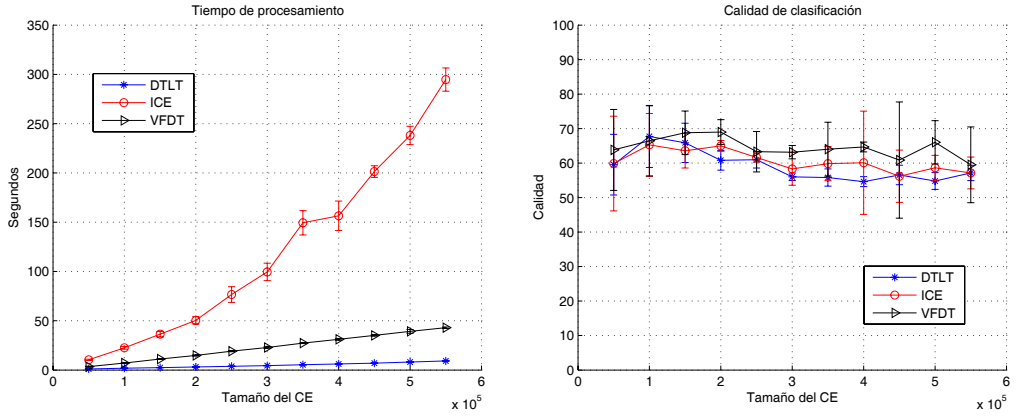


Figura 7.5: Tiempo de procesamiento y calidad de clasificación para Forest CoverType.

Para KDD se crearon conjuntos de entrenamiento de 500,000 a 4,500,000 objetos con incrementos de 500,000 objetos. Los resultados con KDD se muestran en la Figura 7.6. BOAI tampoco es incluido en estos resultados ya que no pudo procesar ninguno de los conjuntos de entrenamiento. Con este conjunto de datos, DTLT es visiblemente más rápido que ICE y VFDT. En cuanto a la calidad de clasificación DTLT y VFDT obtienen resultados similares, y ambos algoritmos son mejores que ICE.

Con los conjuntos de datos Agrawal F1, Agrawal F2 y Agrawal F7 se crearon conjuntos de entrenamiento de 500,000 a 6,500,000 objetos con incrementos de 500,000 objetos. Para estos conjuntos de datos, los resultados de BOAI sólo se muestran hasta el conjunto de entrenamiento de 2,000,000 de objetos, ya que el tiempo de procesamiento empleado por este algoritmo se incrementa muy rápido en comparación al tiempo empleado por los otros algoritmos cuando el número de objetos en el conjunto de entrenamiento se incrementa. La Figura 7.7 muestra los resultados con Agrawal F1, con los algoritmos DTLT, ICE, VFDT y BOAI. En esta figura se puede observar que

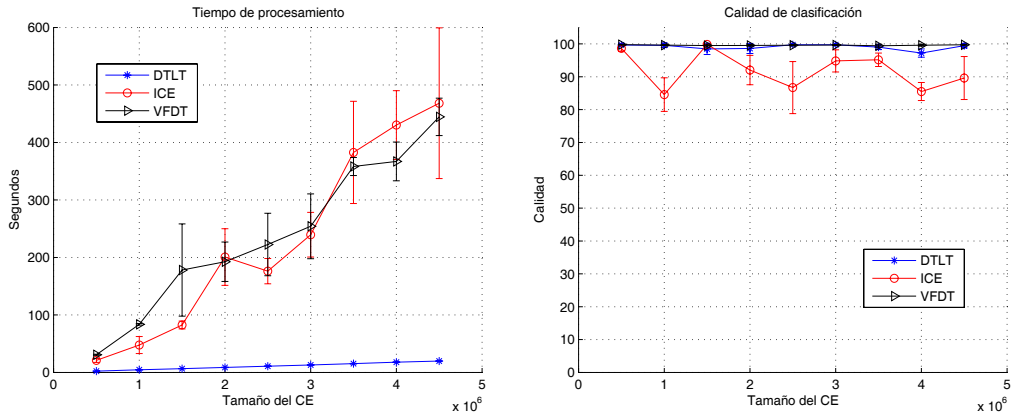


Figura 7.6: Tiempo de procesamiento y calidad de clasificación para KDD.

nuestro algoritmo es mejor que ICE y VFDT. DTLT es hasta 10 y 3.4 veces más rápido que ICE y VFDT, respectivamente. Con respecto a la calidad de clasificación, DTLT y VFDT obtienen resultados similares, siendo ambos algoritmos mejores que ICE y BOAI.

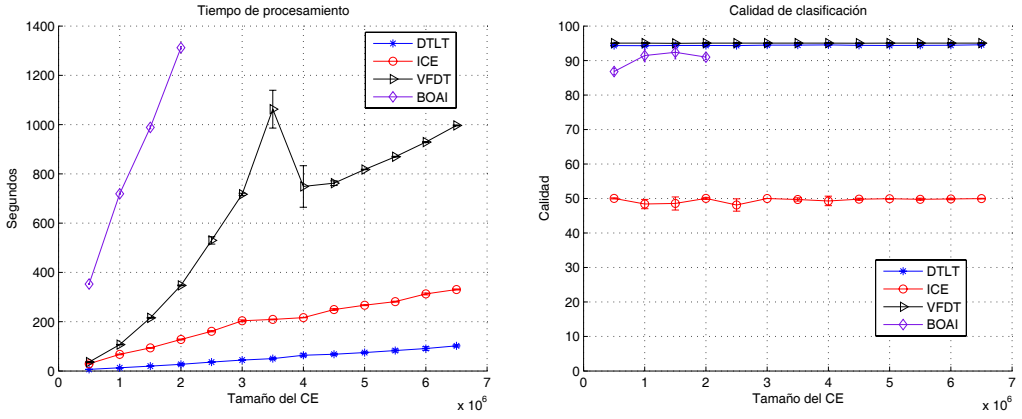


Figura 7.7: Tiempo de procesamiento y calidad de clasificación para Agrawal F1.

Con Agrawal F2, se obtuvieron similares resultados que con Agrawal F1. Estos resultados se pueden observar en la Figura 7.8. Con Agrawal F2, DTLT es hasta 2.7 y 4.7 veces más rápido que ICE y VFDT, obteniendo nuestro algoritmo un calidad de clasificación ligeramente superior a VFDT. Por su parte, DTLT y VFDT obtuvieron una calidad superior a la de ICE y BOAI.

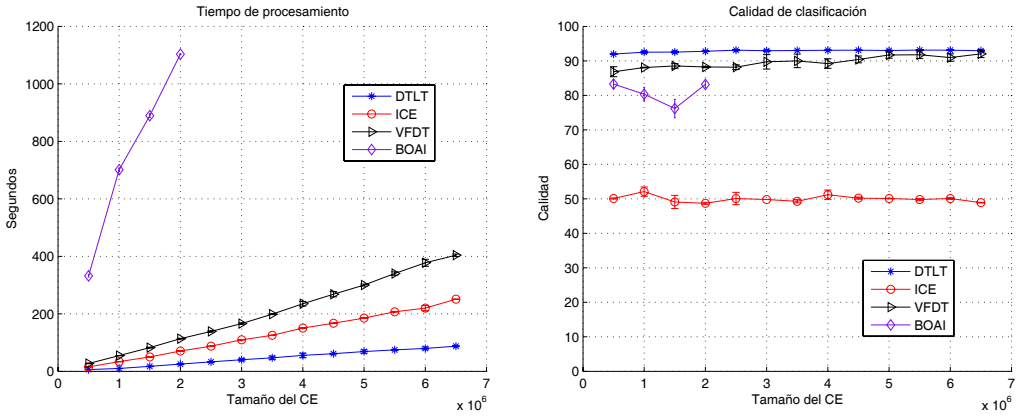


Figura 7.8: Tiempo de procesamiento y calidad de clasificación para Agrawal F2.

La Figura 7.9 muestra los resultados con DTLT, ICE, VFDT y BOAI con Agrawal F7. En esta figura se puede observar que DTLT es hasta 1.75 y 3.6 veces más rápido que ICE y VFDT, y aunque ICE es el segundo algoritmo más rápido, ICE obtiene una calidad de clasificación inferior a la de DTLT. Por su parte, DTLT es más rápido que VFDT y ambos algoritmos obtienen una calidad de clasificación similar.

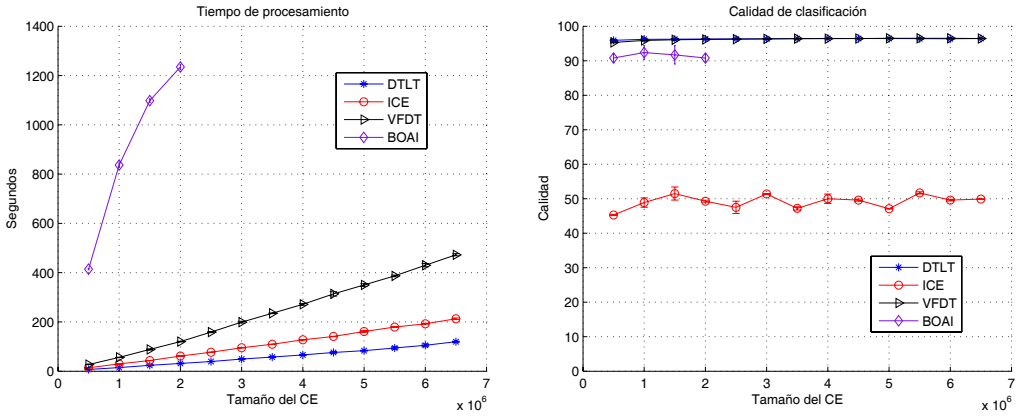


Figura 7.9: Tiempo de procesamiento y calidad de clasificación para Agrawal F7.

A partir de los experimentos realizados, cuando se incrementa el número

de objetos en el conjunto de entrenamiento, se puede observar que DTLT es un algoritmo que mantiene un buen desempeño, independientemente del número de objetos en el conjunto de entrenamiento. Aunque el tiempo de procesamiento de DTLT se incrementa cuando el número de objetos se incrementa, en todos los conjuntos de datos nuestro algoritmo fue más rápido que ICE, VFDT y BOAI. Además, nuestro algoritmo obtuvo una calidad de clasificación competitiva a la obtenida por estos algoritmos.

7.5.4 Incrementando el número de atributos en el conjunto de entrenamiento

En esta sección se analiza experimentalmente el comportamiento de nuestro algoritmo en comparación con ICE y VFDT, cuando se incrementa el número de atributos en el conjunto de entrenamiento.

En este experimento se usó el conjunto de datos KDD, ya que es el conjunto de datos de la Tabla 7.1 que tiene más objetos y atributos. Utilizando el conjunto de entrenamiento de 4,000,000 de objetos, se crearon conjuntos de entrenamiento con diferente número de atributos, de 5 y de 10 a 40 con incrementos de 10 atributos.

En la Figura 7.10 se muestra el tiempo de procesamiento y la calidad de clasificación para DTLT, ICE y VFDT, cuando se incrementa el número de atributos en el conjunto de entrenamiento. BOAI no se muestra en estos resultados ya que no pudo procesar ninguno de los conjuntos de entrenamiento.

Como puede observarse en la Figura 7.10, el tiempo de procesamiento que emplea nuestro algoritmo se incrementa muy poco, a diferencia del tiempo de procesamiento de ICE y VFDT, el cual crece más rápido cuando se incrementa el número de atributos en el conjunto de entrenamiento. Con respecto a la calidad de clasificación, se puede observar que, para los tres algoritmos, la calidad se incrementa conforme el número de atributos se incrementa. Sin embargo, este comportamiento en particular se debe a que KDD necesita de todo el conjunto de atributos (los 41 atributos que tiene en su descripción) para obtener buenos resultados de clasificación.

7.5.5 Uso de memoria para la construcción del AD

En el análisis de la Sección 7.4 se observó que la complejidad espacial de DTLT es del mismo orden que la complejidad espacial de ICE, VFDT y BOAI. Por esta razón, se realizaron experimentos para medir la cantidad

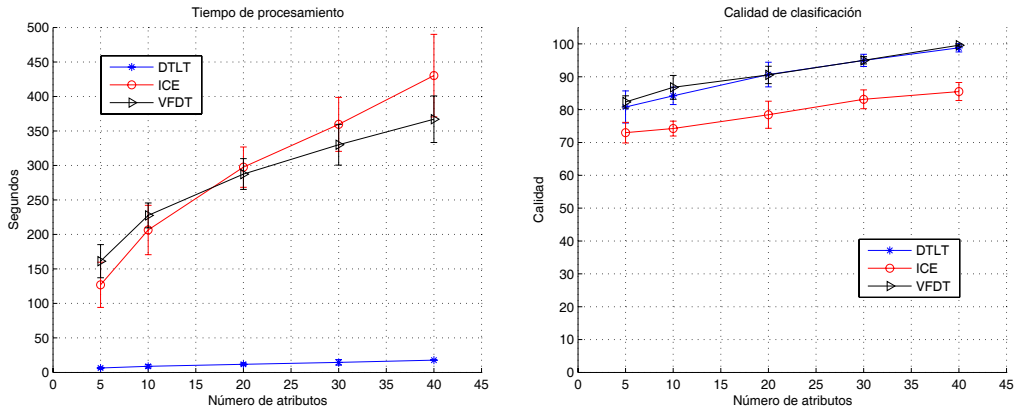


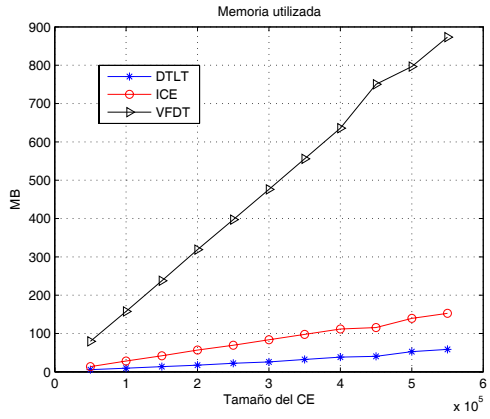
Figura 7.10: Tiempo de procesamiento y calidad de clasificación para DTLT, ICE y VFDT cuando se incrementa el número de atributos en el conjunto de entrenamiento.

de memoria que utilizan estos algoritmos para construir un AD. Utilizando la herramienta *memory* [Gar09] de Linux se obtuvo la cantidad de memoria que emplea cada algoritmo. BOAI no se incluye en estos resultados ya que, como se observó en los experimentos anteriores, no pudo procesar diversos conjuntos de entrenamiento, debido a una falla de memoria.

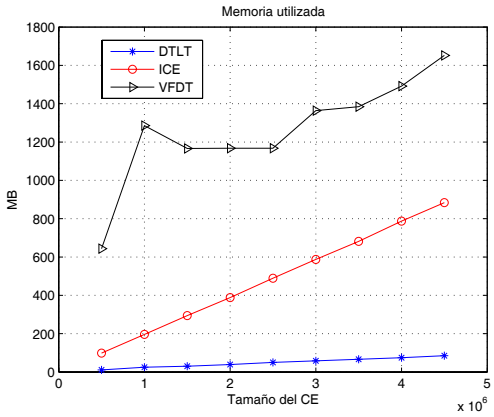
La Figura 7.11 muestra la cantidad de memoria utilizada por cada algoritmo, para Forest CoverType y KDD. Como puede observarse, para Forest CoverType DTLT e ICE son los algoritmos que menor cantidad de memoria emplean para la construcción del AD. Sin embargo, ICE no utiliza el conjunto de entrenamiento completo. Para KDD, nuestro algoritmo es el que menor cantidad de memoria utiliza para construir el AD.

La Figura 7.12 muestra la cantidad de memoria utilizada por DTLT, ICE y VFDT con Agrawal F1, Agrawal F2 y Agrawal F7.

Como puede observarse en la Figura 7.12, con estos conjuntos de datos, DTLT e ICE utilizan similar cantidad de memoria, y ambos algoritmos utilizan menor cantidad de memoria que VFDT, que es el más rápido de los algoritmos previos. Sin embargo, como ya se había mencionado, ICE es un algoritmo que no utiliza todo el conjunto de entrenamiento para la construcción del AD, además en los experimentos anteriores fue el que obtuvo menor calidad de clasificación.

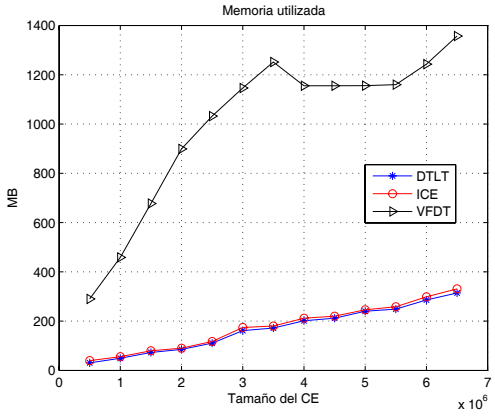


(a) Forest CoverType

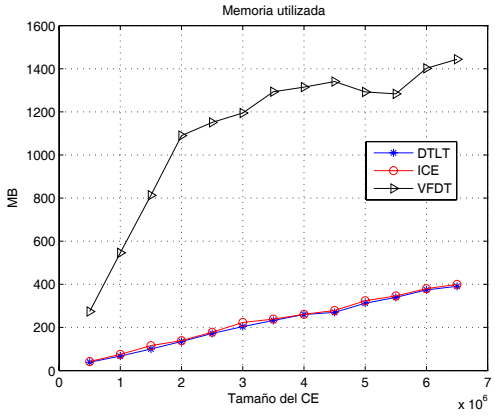


(b) KDD

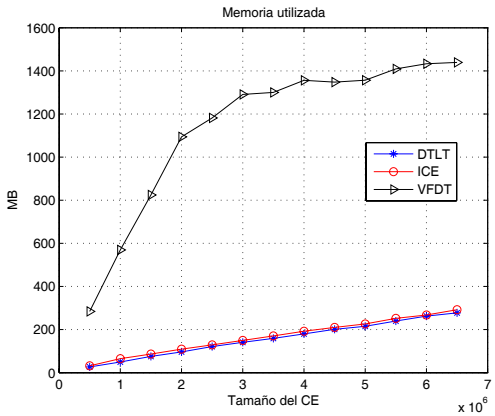
Figura 7.11: Cantidad de memoria utilizada por DTLT, ICE y VFDT con Forest CoverType y KDD.



(a) Agrawal F1



(b) Agrawal F2



(c) Agrawal F7

Figura 7.12: Cantidad de memoria utilizada por DTLT, ICE y VFDT con Agrawal F1, Agrawal F2 y Agrawal F7.

7.5.6 DTLT con grandes conjuntos de datos numéricos

En esta sección se analiza el comportamiento de DTLT con grandes conjuntos de datos numéricos y se muestra una comparación contra el algoritmo DTFS que fue diseñado para generar ADs para grandes conjuntos de datos numéricos, ya que ambos construyen ADs univaluados.

Para realizar este experimento, se utilizaron los conjuntos de datos reales descritos en la Tabla 4.1 (Poker, SpecObj y GalStar) y las particiones, de cada conjunto, descritas en la Sección 4.5.4.1.

La Figura 7.13 muestra los resultados obtenidos con Poker. Como puede observarse en esta figura, el tiempo de procesamiento y la calidad de clasificación de ambos algoritmos es muy similar.

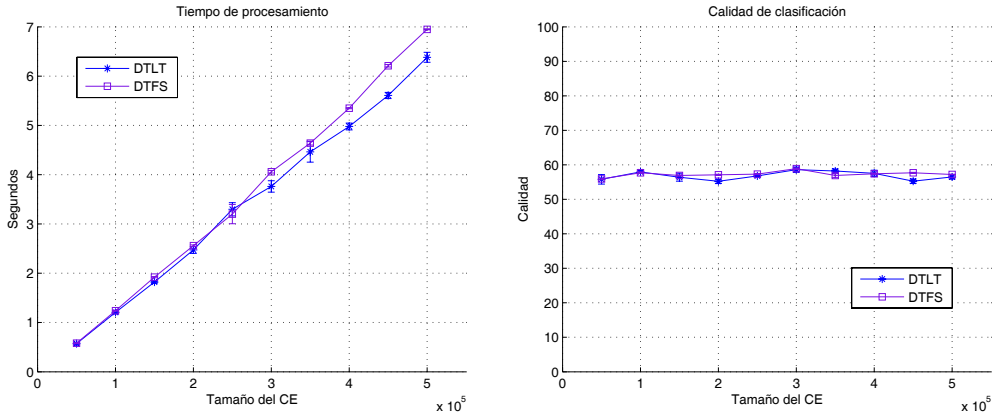


Figura 7.13: Tiempo de procesamiento y calidad de clasificación para DTLT y DTFS con Poker.

Los resultados con SpecObj se muestran en la Figura 7.14, en donde puede observarse que DTFS es hasta 1.33 más rápido que DTLT. Además, DTFS obtiene una calidad de clasificación superior a la obtenida por DTLT.

La Figura 7.15 muestra los resultados obtenidos con el conjunto de datos GalStar. En esta figura puede apreciarse que DTFS es hasta 5.36 veces más rápido que DTLT. Con este conjunto de datos se puede apreciar una mayor diferencia entre los tiempos de procesamiento de ambos algoritmos, ya que GalStar tiene más atributos que Poker y SpecObj, y cada uno de sus atributos puede tomar una gran cantidad de valores diferentes. Con respecto a la calidad de clasificación, DTFS fue mejor que DTLT.

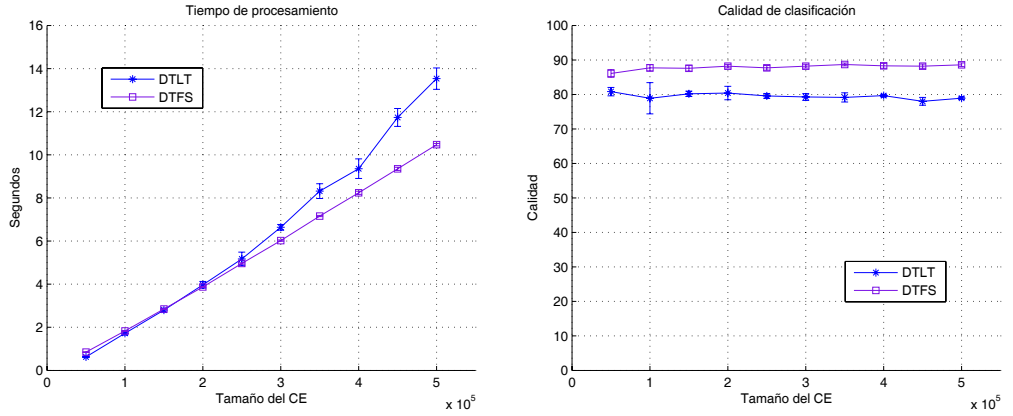


Figura 7.14: Tiempo de procesamiento y calidad de clasificación para DTLT y DTFS con SpecObj.

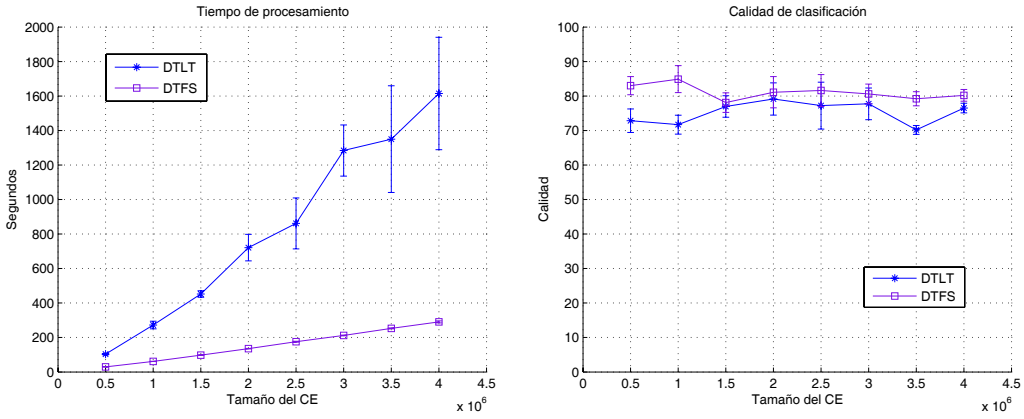


Figura 7.15: Tiempo de procesamiento y calidad de clasificación para DTLT y DTFS con GalStar.

De acuerdo a los resultados obtenidos anteriormente, cuando se trata de grandes conjuntos de datos numéricos, el algoritmo DTFS fue mejor que DTLT, ya que este último emplea mayor tiempo de procesamiento y la calidad de clasificación que obtiene es inferior a la de DTFS.

7.6. Discusión

En este capítulo se propuso el algoritmo DTLT, el cual construye ADs univaluados para grandes conjuntos de datos mezclados de manera incremental, utilizando todo el conjunto de entrenamiento, sin almacenarlo completo en memoria. De los experimentos realizados con este algoritmo, podemos observar que DTLT mantiene un buen desempeño cuando se incrementa el número de objetos y atributos en el conjunto de entrenamiento. Tomando en cuenta los resultados obtenidos con todos los conjuntos de datos de los experimentos, DTLT es, en promedio, hasta 14.99, 9.34 y 37.98 veces más rápido que ICE, VFDT y BOAI (los tres algorimos más recientes de generación de ADs para grandes conjuntos de datos), respectivamente, cuando se incrementa el número de objetos. Por su parte, cuando se incrementa el número de atributos en el conjunto de entrenamiento, DTLT es hasta 21.5 y 18.5 veces más rápido que ICE y VFDT, respectivamente. En cuanto a la calidad de clasificación, nuestro algoritmo obtiene resultados similares a los obtenidos con ICE, VFDT y BOAI, independientemente del número de objetos y atributos en el conjunto de entrenamiento.

Con respecto al parámetro s (s es el número máximo de objetos que puede almacenar un nodo), se observó que con valores pequeños para s, DTLT obtiene mejores resultados que con valores grandes. El valor de $s = 100$ (usado en los experimentos) es una buena opción, ya que con este valor se obtuvieron mejores resultados que con otros valores, manteniendo un buen balance entre tiempo de procesamiento y calidad de clasificación del algoritmo.

De la cantidad de memoria que utiliza DTLT para la construcción del AD, se observó que fue menor que la cantidad de memoria que usan ICE, VFDT y BOAI, aún cuando los cuatro algoritmos tienen el mismo orden de complejidad espacial.

Por otro lado, de acuerdo a los experimentos realizados, cuando se trata de grandes conjuntos de datos numéricos, el algoritmo DTFS fue mejor que DTLT, ya que este último emplea un mayor tiempo de procesamiento para construir el AD. DTFS es, en promedio, hasta 3.34 veces más rápido que DTLT. Esto se debe a que el proceso que sigue DTFS para la selección del atributo de prueba en un nodo a expandir, es más rápido que el de DTLT. DTFS sólo evalua un conjunto de valores de prueba para cada atributo, mientras que DTLT tiene que evaluar todos los posibles puntos de corte de cada atributo. Además, la calidad de clasificación obtenida por DTFS es hasta 10 puntos porcentuales superior a la obtenida por DTLT.

Con el algoritmo DTLT se cumple el tercer objetivo particular de esta

tesis. DTLT es un algoritmo que genera ADs univaluados para grandes conjuntos de datos con atributos numéricos y no numéricos (datos mezclados). Además, DTLT es más rápido que los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos (ICE, VFDT y BOAI) y obtiene una calidad de clasificación competitiva con estos algoritmos.

Capítulo 8

Conclusiones

Los Árboles de Decisión son una herramienta útil para resolver problemas de clasificación supervisada. En la actualidad se han desarrollado diversos algoritmos que generan ADs, incluso algunos que trabajan con grandes conjuntos de datos. Sin embargo, los algoritmos de generación de ADs para este tipo de conjuntos de datos tienen algunas limitaciones, como por ejemplo: el espacio de memoria que requieren, el número de veces que tienen que recorrer el conjunto de entrenamiento para construir el árbol, el no tener la capacidad de actualizarse si el conjunto de datos tiene un flujo continuo de información, o tienen diversos parámetros que pueden ser difíciles de determinar por parte del usuario.

Por estas razones surgió la necesidad de proponer nuevos algoritmos de generación de ADs para grandes conjuntos de datos, los cuales procesaran todo el conjunto de entrenamiento en un tiempo menor que el de los algoritmos existentes y obtuvieran una calidad de clasificación competitiva. En este trabajo se propusieron, analizaron y compararon cuatro algoritmos de generación de ADs para grandes conjuntos de datos, IIMDT, IIMDTS, DTFS y DTLT, los cuales procesan todo el conjunto de entrenamiento y generan el AD más rápido que ICE, VFDT y BOAI, los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos. Además, nuestros algoritmos obtienen una calidad de clasificación competitiva con la de estos algoritmos.

Los primeros tres algoritmos propuestos en esta tesis son para generar ADs para grandes conjuntos de datos numéricos y el cuarto algoritmo es para generar ADs para grandes conjuntos de datos mezclados. Todos los algoritmos propuestos trabajan de manera incremental los objetos del conjunto de entrenamiento. De acuerdo con los resultados obtenidos con estos algo-

ritmos, se puede afirmar que se cumplió con el objetivo de este trabajo de investigación.

Con base en los experimentos de cada algoritmo, se tienen las siguientes conclusiones:

- IIMDT genera ADs multivaluados que utilizan todo el conjunto de atributos para caracterizar a los nodos internos del AD. Los resultados experimentales obtenidos con este algoritmo, muestran que IIMDT es competitivo en calidad y es hasta 13.78, 5.85 y 7.03 veces más rápido que los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos: ICE, VFDT y BOAI, respectivamente.

- IIMDTS tiene las mismas características que IIMDT, con la diferencia de que IIMDTS incorpora una selección de subconjuntos de atributos de prueba de tamaño n para los nodos internos, de tal manera que el recorrido del AD que hagan los objetos sea más rápido y por lo tanto la construcción del AD se acelere. Con base en el análisis experimental de la Sección 5.5.2, se puede concluir que cuando n toma un valor pequeño (1, 2 o 5 según nuestros experimentos) IIMDTS supera a los algoritmos ICE, VFDT y BOAI, ya que IIMDTS es hasta 19.12, 9.21 y 14.89 veces más rapido que estos algoritmos, respectivamente, y la calidad obtenida por los cuatro algoritmos es muy similar. Con respecto a IIMDT, IIMDTS es, en promedio, hasta 3.19 veces más rápido que IIMDT, cuando IIMDTS elige subconjuntos de atributos pequeños (en nuestros experimentos, subconjuntos de 1, 2 o 5 atributos). Sin embargo, si IIMDTS elige subconjuntos de datos con más atributos (por ejemplo, igual al tamaño del conjunto total de atributos) IIMDT es hasta 1.73 veces más rápido que IIMDTS. Con respecto a la calidad de clasificación, ambos algoritmos obtienen resultados similares.

- DTFS es un algoritmo diseñado para generar ADs univaluados para grandes conjuntos de datos numéricos. En este algoritmo se propuso una nueva forma de aplicar el Criterio de Proporción de Ganancia de Información para elegir el atributo de prueba en cada nodo a expandir. Los resultados de este algoritmo muestran que DTFS es hasta 21.53, 8.30 y 19.01 veces más rápido que ICE, VFDT y BOAI, respectivamente, obteniendo una calidad de clasificación similar a la de estos algoritmos. Comparando IIMDTS (cuando este algoritmo toma el valor de $n = 1$) contra DTFS, se pudo notar que la diferencia en el tiempo

de procesamiento de ambos algoritmos es pequeña, y que su calidad de clasificación es similar.

- DTLT construye ADs univaluados iguales a los construidos por algoritmos de generación de ADs tradicionales. De acuerdo a los resultados presentados en este trabajo se observó que DTLT obtiene una calidad de clasificación competitiva con ICE, VFDT y BOAI, pero nuestro algoritmo es hasta 14.99, 9.34 y 37.98 veces más rápido que estos algoritmos, respectivamente. Con respecto a DTFS (que también construye ADs univaluados pero sólo para grandes conjuntos de datos numéricos), comparando ambos algoritmos con grandes conjuntos de datos numéricos, se pudó observar que DTFS es hasta 3.34 veces más rápido que DTLT y además DTFS obtiene una calidad de clasificación superior a la obtenida por DTLT.

- Para todos los algoritmos propuestos se recomienda utilizar un valor pequeño para el parámetro s. De acuerdo a los experimentos, con conjuntos de datos de hasta 4,500,000 de objetos, utilizar $s = 100$ da buenos resultados, ya que se mantiene un buen balance entre tiempo de procesamiento y calidad de clasificación del AD.

La Tabla 8.1 presenta en resumen la razón de incremento en el tiempo de procesamiento empleado por los algoritmos propuestos en esta tesis para generar un AD, contra el empleado por los algoritmos más recientes de generación de ADs para grandes conjuntos de datos. La coincidencia entre cada par de algoritmos muestra, en promedio, el número de veces que nuestros algoritmos superan a ICE, VFDT y BOAI.

Tabla 8.1: Número de veces que los algoritmos propuestos superan a algoritmos recientes.

Algoritmo	ICE	VFDT	BOAI
IIMDT	13.78	5.85	7.03
IIMDTS	19.12	9.21	14.89
DTFS	21.53	8.30	19.01
DTLT	14.99	9.34	37.98

Contribuciones del trabajo de investigación

Las contribuciones de este trabajo de investigación son cuatro algoritmos para generar ADs para grandes conjuntos de datos.

- IIMDT es un algoritmo de generación de ADs multivaluados para grandes conjuntos de datos numéricos, el cual utiliza todo el conjunto de atributos en los nodos internos del AD.
- IIMDTS es un algoritmo de generación de ADs multivaluados para grandes conjuntos de datos numéricos, el cual utiliza subconjuntos de atributos en los nodos internos del AD.
- DTFS es un algoritmo de generación de ADs univaluados para grandes conjuntos de datos numéricos.
- DTLT es un algoritmo de generación de ADs univaluados para grandes conjuntos de datos mezclados (con atributos numéricos y no numéricos).

Todos los algoritmos propuestos en este trabajo de tesis son más rápidos en la construcción del AD que los tres algoritmos más recientes de generación de ADs para grandes conjuntos de datos: ICE [YAR99], VFDT [DH00] y BOAI [YWYC08].

Trabajo futuro

Como trabajo futuro se propone lo siguiente:

- Los algoritmos propuestos en este trabajo requieren de dos parámetros iniciales, el número de objetos que puede almacenar un nodo antes de ser expandido o actualizado, y el número de atributos a elegir para un nodo interno (en el caso del algoritmo IIMDTS). Por lo que, se propone buscar métodos para la selección automática de valores para estos parámetros.
- Proponer otra forma para calcular los valores de prueba asociados a los arcos de salida en un nodo a expandir, de tal manera que se incremente la calidad de clasificación de los algoritmos propuestos en esta investigación.

Publicaciones derivadas del trabajo de investigación

Las publicaciones que se han derivado de esta tesis son:

- Anilu Franco-Arcega, J. Ariel Carrasco-Ochoa, Guillermo Sánchez-Díaz and J.Fco. Martínez-Trinidad. A New Incremental Algorithm for Induction of Multivariate Decision Trees for Large Datasets. In Proc. of the 9th International Conference on Intelligent Data Engineering and Automated Learning - IDEAL. LNCS 5326, pages 282-289, 2008.
- Anilu Franco-Arcega, J. Ariel Carrasco-Ochoa, Guillermo Sánchez-Díaz and J.Fco. Martínez-Trinidad. Multivariate Decision Trees using Different Splitting Attribute Subsets for Large Datasets. In Proc. of the 23th Canandian Conference on Artificial Intelligence, LNAI, Springer Vol. 6085, pages 370-373, 2010.
- Anilu Franco-Arcega, J. Ariel Carrasco-Ochoa, Guillermo Sánchez-Díaz and J.Fco. Martínez-Trinidad. A fast decision tree algorithm for large datasets. Artículo sometido a revisión en la revista: International Journal on Artificial Intelligence Tools. Fecha de envío: 21 de octubre 2009.
- Anilu Franco-Arcega, J. Ariel Carrasco-Ochoa, Guillermo Sánchez-Díaz and J.Fco. Martínez-Trinidad. Decision tree induction using a fast splitting attribute selection for large datasets. Artículo sometido a revisión en la revista: Expert Systems With Applications. Fecha de envío: 27 de abril 2010.
- Anilu Franco-Arcega, J. Ariel Carrasco-Ochoa, Guillermo Sánchez-Díaz and J.Fco. Martínez-Trinidad. Building fast decision trees from large training sets. Artículo sometido a revisión en la revista: Pattern Recognition. Fecha de envío: 28 de junio 2010.

Apéndice A

Significancia estadística

En este apéndice se muestra una evaluación de la significancia estadística de la calidad de clasificación entre los algoritmos propuestos en esta investigación y los algoritmos más recientes de generación de ADs para grandes conjuntos de datos, ICE y VFDT. BOAI no se muestra en estos resultados, debido a que en la mayoría de los conjuntos de entrenamiento no pudo construir el AD debido a fallas de memoria.

La prueba de significancia estadística utilizada fue T-test [Die98]. La Tabla A.1 presenta los resultados con el algoritmo IIMDT, la Tabla A.2 los resultados con IIMDTS, en la Tabla A.3 se muestran los resultados con DTFS y finalmente, la Tabla A.4 muestra los resultados con el algoritmo DTLT. Cada una de las tablas muestran si existe diferencia significativa entre los resultados de cada par de algoritmos. En caso de que la coincidencia entre dos algoritmos, en cualquiera de las tablas, diga NO es que no existe una diferencia significativa entre ellos. En caso de que diga SI, significa que existe una diferencia significativa entre los algoritmos. En este caso se cuantifica también la diferencia, si la diferencia es positiva $(+)$ nuestro algoritmo fue superior al otro algoritmo, si es negativa $(-)$ entonces el otro algoritmo evaluado fue superior al nuestro. La tabla muestra los resultados utilizando los niveles de confianza de 90 %, 95 % y 99 %.

Tabla A.1: Significancia estadística con IIMDT

	90 %		95 %		99 %	
Dataset	ICE	VFDT	ICE	VFDT	ICE	VFDT
Poker	SI (-1.85)	SI (+2.57)	NO	SI (+2.57)	NO	SI (+2.57)
SpecObj	SI (+3.26)	SI (+5.53)	SI (+3.26)	SI (+5.53)	SI (+3.26)	SI (+5.53)
GalStar	SI (+2.47)	NO	SI (+2.47)	NO	SI (+2.47)	NO
Sintética_1	SI (+1.03)	SI (+0.69)	SI (+1.03)	SI (+0.69)	SI (+1.03)	SI (+0.69)
Sintética_2	SI (+13.53)	SI (-0.81)	SI (+13.53)	SI (-0.81)	SI (+13.53)	SI (-0.81)
Sintética_3	SI (+12.33)	SI (-0.44)	SI (+12.33)	SI (-0.44)	SI (+12.33)	SI (-0.44)
Sintética_4	SI (+1.68)	SI (+3.10)	SI (+1.68)	SI (+3.10)	SI (+1.68)	SI (+3.10)
Sintética_5	SI (+10.02)	SI (-0.74)	SI (+10.02)	SI (-0.74)	SI (+10.02)	SI (-0.74)
Sintética_6	SI (+14.29)	SI (-0.35)	SI (+14.29)	SI (-0.35)	SI (+14.29)	SI (-0.35)

Tabla A.2: Significancia estadística con IIMDTS

	90 %		95 %		99 %	
Dataset	ICE	VFDT	ICE	VFDT	ICE	VFDT
Poker	NO	SI (+4.31)	NO	SI (+4.31)	NO	SI (+4.31)
SpecObj	SI (+2.08)	SI (+4.35)	SI (+2.08)	SI (+4.35)	SI SI (+2.08)	SI (+4.35)
GalStar	SI (+2.29)	NO	SI (+2.29)	NO	SI (+2.29)	NO
Sintética_1	SI (+0.40)	NO	SI (+0.40)	NO	SI (+0.40)	NO
Sintética_2	SI (+12.85)	SI (-1.49)	SI (+12.85)	SI (-1.49)	SI (+12.85)	SI (-1.49)
Sintética_3	SI (+11.13)	SI (-1.64)	SI (+11.13)	SI (-1.64)	SI (+11.13)	SI (-1.64)
Sintética_4	SI (+0.25)	SI (+1.67)	SI (+0.25)	SI (+1.67)	SI (+0.25)	SI (+1.67)
Sintética_5	SI (+9.32)	SI (-1.44)	SI (+9.32)	SI (-1.44)	SI (+9.32)	SI (-1.44)
Sintética_6	SI (+13.01)	SI (-1.63)	SI (+13.01)	SI (-1.63)	SI (+13.01)	SI (-1.63)

Tabla A.3: Significancia estadística con DTFS

	90 %		95 %		99 %	
Dataset	ICE	VFDT	ICE	VFDT	ICE	VFDT
Poker	SI (-4.61)	NO	SI (-4.61)	NO	SI (-4.61)	NO
SpecObj	SI (+1.44)	SI (+3.71)	SI (+1.44)	SI (+3.71)	SI (+1.44)	SI (+3.71)
GalStar	SI (+2.60)	NO	SI (+2.60)	NO	SI (+2.60)	NO
Sintética_1	NO	NO	NO	NO	NO	NO
Sintética_2	SI (+13.30)	SI (-1.04)	SI (+13.30)	SI (-1.04)	SI (+13.30)	SI (-1.04)
Sintética_3	SI (+12.12)	SI (-0.65)	SI (+12.12)	SI (-0.65)	SI (+12.12)	SI (-0.65)
Sintética_4	NO	SI (+1.48)	NO	SI (+1.48)	NO	SI (+1.48)
Sintética_5	SI (+9.75)	SI (-1.01)	SI (+9.75)	SI (-1.01)	SI (+9.75)	SI (-1.01)
Sintética_6	SI (+14.00)	SI (-0.64)	SI (+14.00)	SI (-0.64)	SI (+14.00)	SI (-0.64)

Tabla A.4: Significancia estadística con DTLT

	90 %		95 %		99 %	
Dataset	ICE	VFDT	ICE	VFDT	ICE	VFDT
Forest CoverType	NO	NO	NO	NO	NO	NO
KDD	SI (+9.86)	NO	SI (+9.86)	NO	SI (+9.86)	NO
Agrawal F1	SI (+44.91)	SI (-0.39)	SI (+44.91)	SI (-0.39)	SI (+44.91)	SI (-0.39)
Agrawal F2	SI (+44.02)	SI (+0.92)	SI (+44.02)	SI (+0.92)	SI (+44.02)	SI (+0.92)
Agrawal F3	SI (+46.60)	NO	SI (+46.60)	NO	SI (+46.60)	NO

Apéndice B

Comparación con el algoritmo C4.5

En esta sección se presenta una comparación de los algoritmos de generación de ADs para grandes conjuntos de datos, propuestos y utilizados en esta tesis, contra uno de los algoritmos tradicionales de generación de ADs más usado en la literatura, C4.5 [Qui93]. Esta comparación se realizó con la finalidad de mostrar que la calidad de clasificación obtenida por los algoritmos para grandes conjuntos de datos es comparable con la de C4.5 y que los tiempos de procesamiento de los algoritmos diseñados para trabajar con grandes conjuntos de datos son considerablemente menores a los de un algoritmo tradicional.

Las Figuras B.1-B.9 muestran los resultados de los conjuntos de datos numéricos. Las Figuras B.10-B.14 presentan los resultados de los conjuntos de datos mezclados.

Para cada conjunto de datos se crearon conjuntos de entrenamiento de 100,000 a 500,000 objetos con incrementos de 100,000 objetos. Se realizaron estos experimentos sólo con conjuntos de entrenamiento de este tamaño debido a que C4.5 incrementa su tiempo de procesamiento drásticamente cuando el tamaño del conjunto de entrenamiento se incrementa, a diferencia de los otros algoritmos. De los experimentos podemos notar que la calidad de clasificación obtenida por los algoritmos de generación de ADs para grandes conjuntos de datos es similar a la obtenida por C4.5, pero requiriendo un tiempo de procesamiento mucho menor.

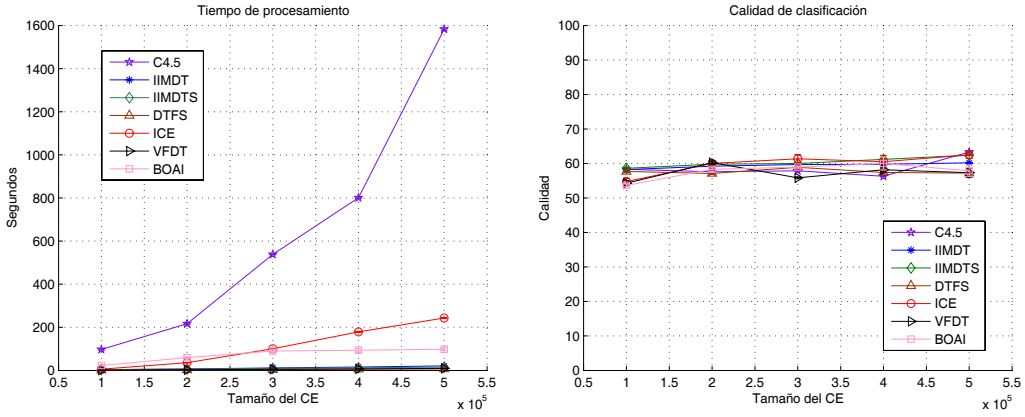


Figura B.1: Tiempo de procesamiento y calidad de clasificación para Poker.

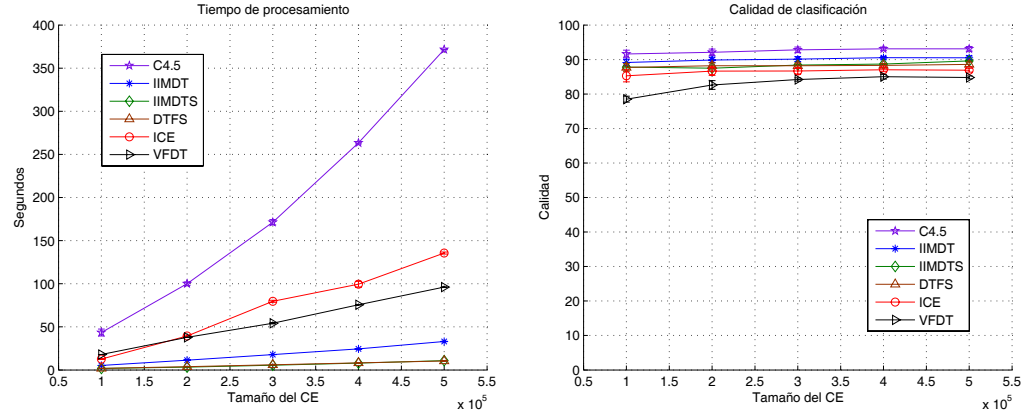


Figura B.2: Tiempo de procesamiento y calidad de clasificación para SpecObj.

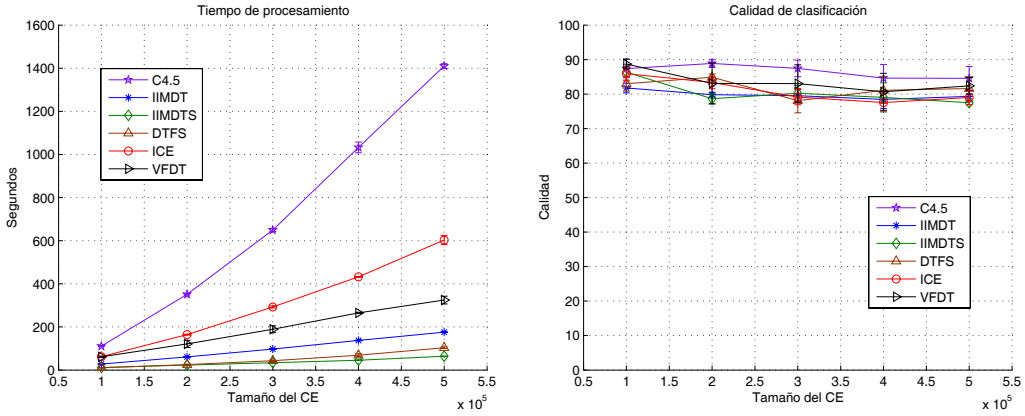


Figura B.3: Tiempo de procesamiento y calidad de clasificación para GalStar.

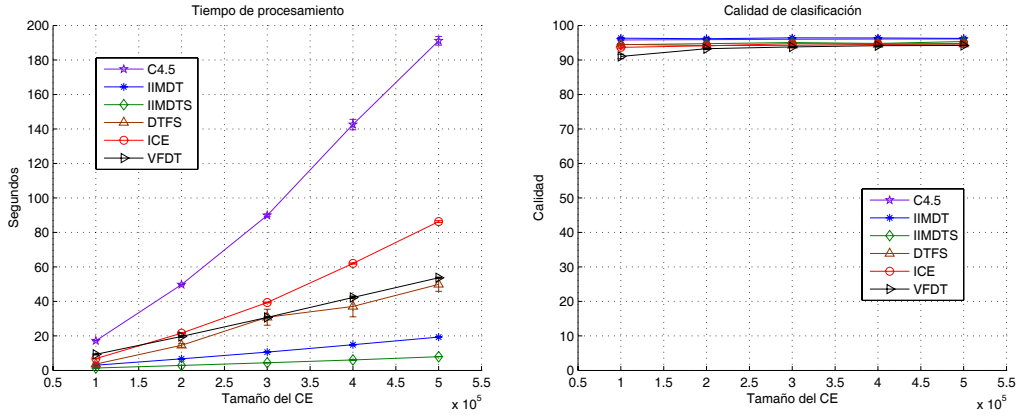


Figura B.4: Tiempo de procesamiento y calidad de clasificación para conjuntos de entrenamiento de 2 clases y 5 atributos.

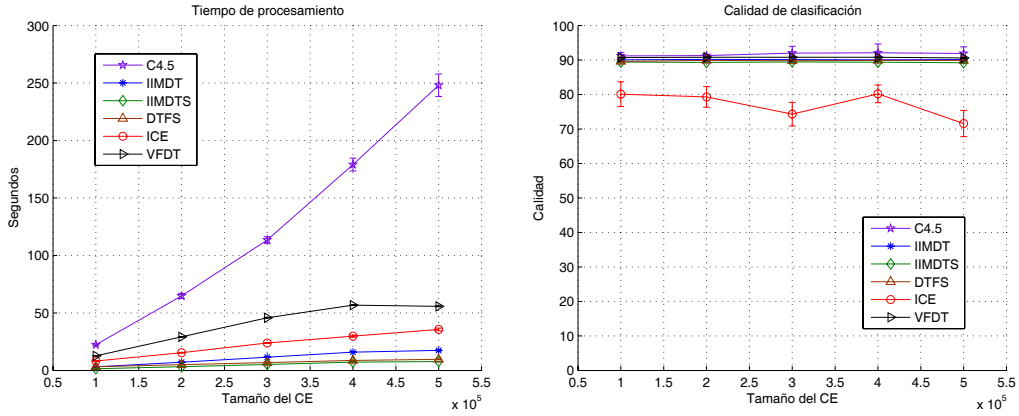


Figura B.5: Tiempo de procesamiento y calidad de clasificación para conjuntos de entrenamiento de 3 clases y 5 atributos.

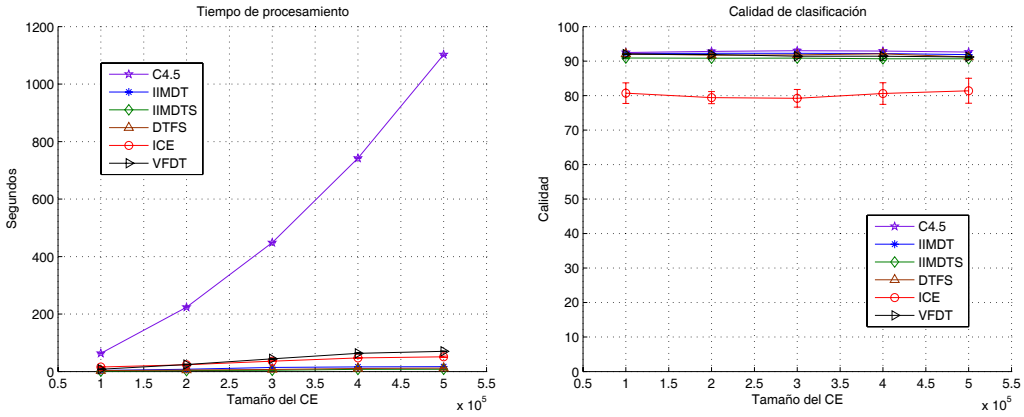


Figura B.6: Tiempo de procesamiento y calidad de clasificación para conjuntos de entrenamiento de 5 clases y 5 atributos.

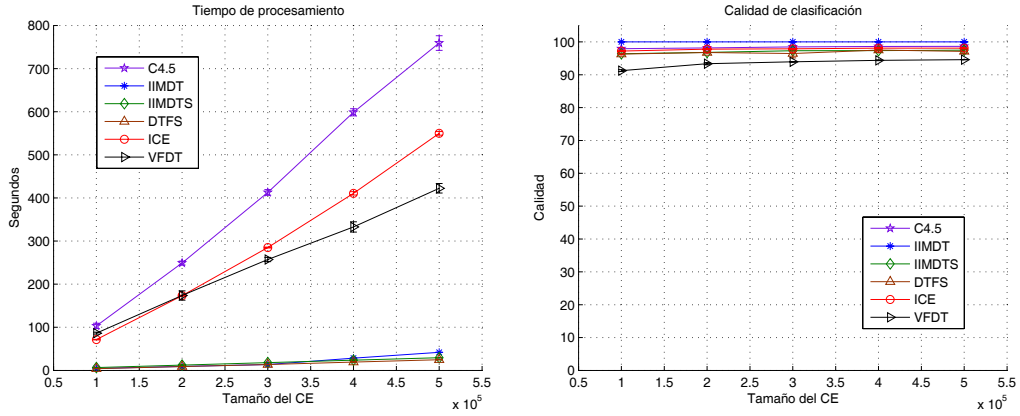


Figura B.7: Tiempo de procesamiento y calidad de clasificación para conjuntos de entrenamiento de 2 clases y 40 atributos.

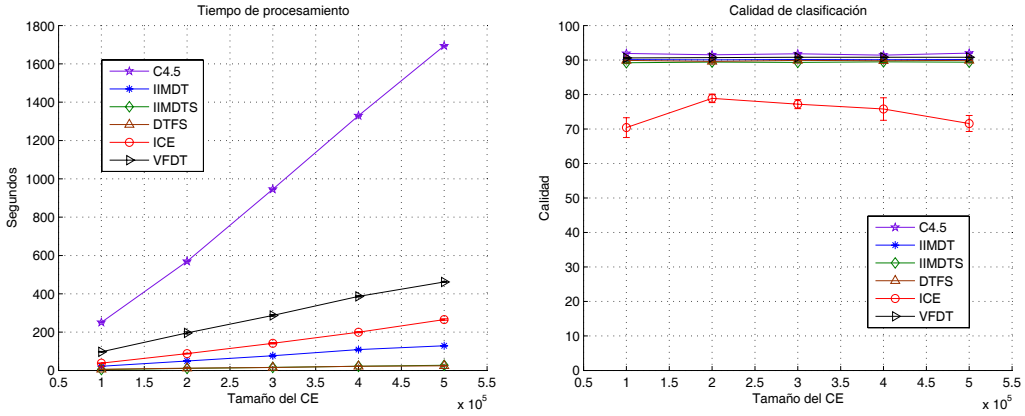


Figura B.8: Tiempo de procesamiento y calidad de clasificación para conjuntos de entrenamiento de 3 clases y 40 atributos.

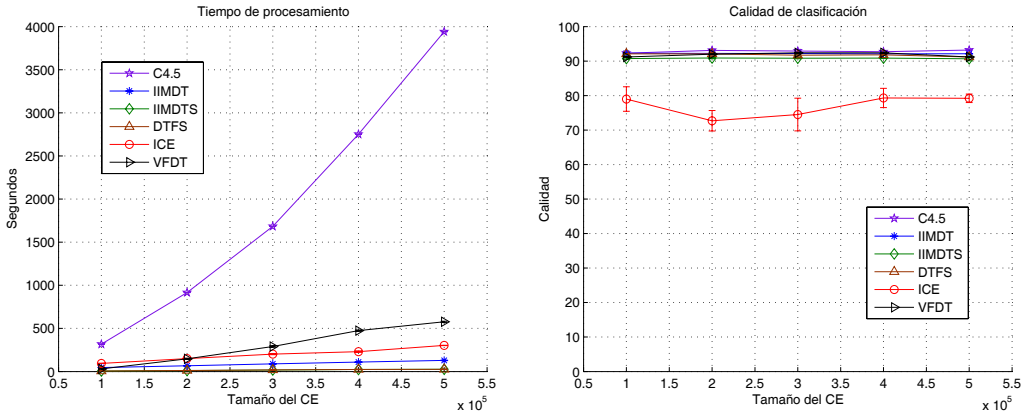


Figura B.9: Tiempo de procesamiento y calidad de clasificación para conjuntos de entrenamiento de 5 clases y 40 atributos.

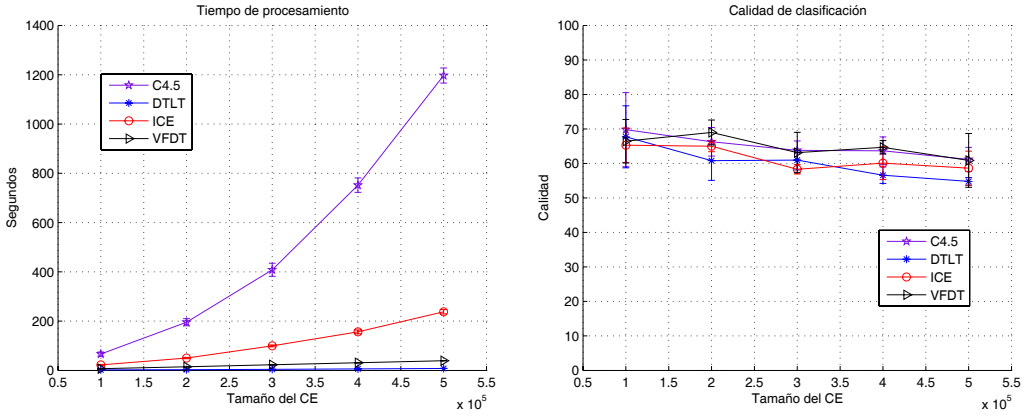


Figura B.10: Tiempo de procesamiento y calidad de clasificación para Forest CoverType.

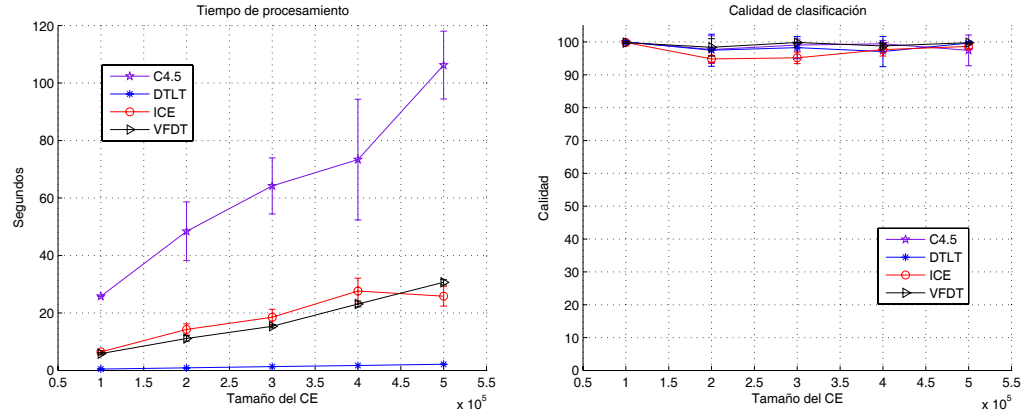


Figura B.11: Tiempo de procesamiento y calidad de clasificación para KDD.

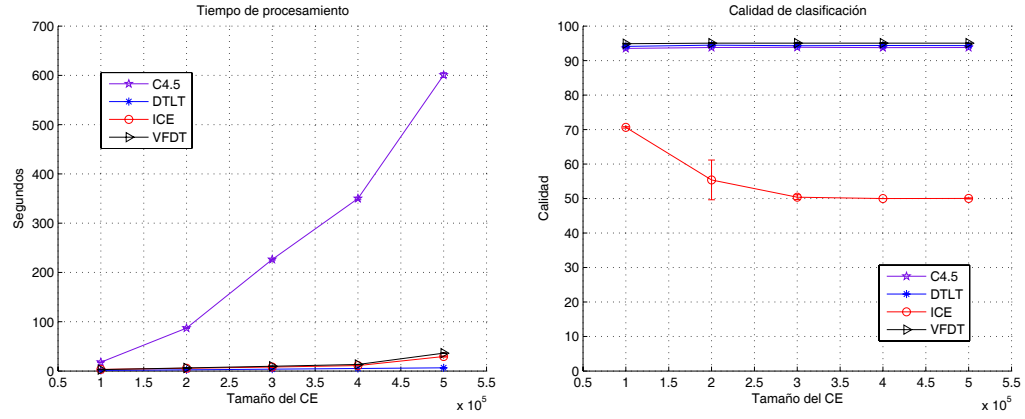


Figura B.12: Tiempo de procesamiento y calidad de clasificación para Agrawal F1.

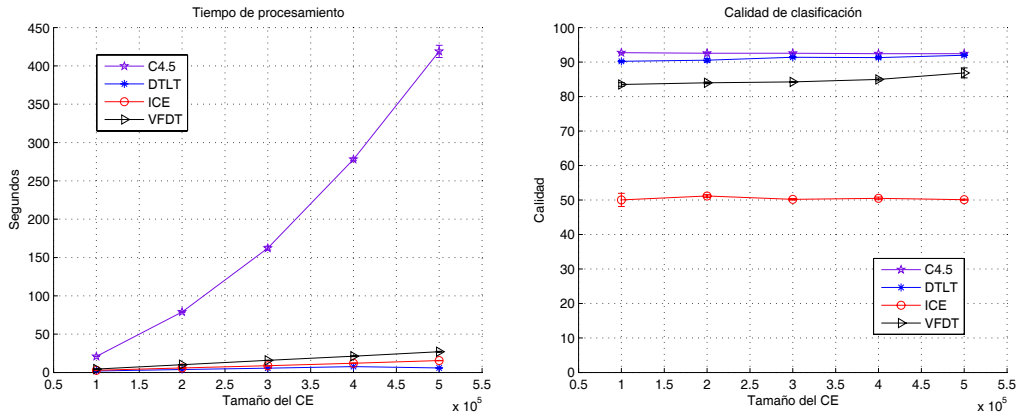


Figura B.13: Tiempo de procesamiento y calidad de clasificación para Agrawal F2.

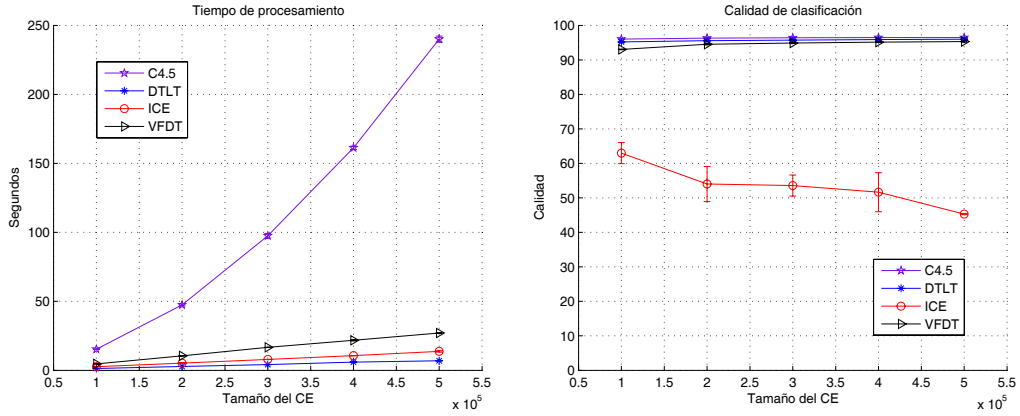


Figura B.14: Tiempo de procesamiento y calidad de clasificación para Agrawal F7.

Apéndice C

Glosario

Acrónimo	Definición
AD	Árbol de Decisión
AVC	Conjunto de listas formadas por Atributo-Valor, Clase
BOAI	BOttom-up evaluation for ADTtree Induction
BOAT	Bootstrapped Optimistic Algorithm for Tree construction
CART	Classification And Regression Tree
CE	Conjunto de Entrenamiento
CLOUDS	Classification for Large or OUt-of-core DataSets
CTC	Consolidated Tree Construction algorithm
C-DT	C-Fuzzy Decision Trees
DTFS	Decision Trees using a Fast Splitting attribute selection
DTLT	Decision Trees for Large Training sets

Acrónimo	Definición
FACT	FAst Classification Tree algorithm
FDT	Fuzzy Decision Trees
GALE	Genetic and Artificial Life Environment
ICE	Incrementally Classifying Ever-growing large datasets
IIMDT	Incremental Induction of Multivariate Decision Trees
IIMDTS	Incremental Induction of Multivariate Decision Trees with attribute splitting Subsets
ITI	Incremental Tree Induction
LMDT	Linear Machine Decision Trees
LTU	Linear Threshold Unit
PC	Punto de Corte
QUEST	Quick, Unbiased, Efficient, Statistical Tree
SLIQ	Supervised Learning In Quest
SPRINT	Scalable PaRallelizable INduction of decision Trees
UFFT	Ultra Fast Forest Tree system
VFDT	Very Fast Decision Trees

Referencias

[AN07] A. Asuncion and D.J. Newman. Uci - machine learning repository. *http://www.ics.uci.edu/∼mlearn/MLRepository.html*, University of California, Irvine, School of Information and Computer Sciences, 2007.

[Arc99] The UCI KDD Archive. Kdd cup 1999 data. *http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html*, Information and Computer Science. University of California, Irvine, 1999.

[ARS98] K. Alsabti, S. Ranka, and V. Singh. CLOUDS: A decision tree classifier for large datasets,. In *Proc. Conference Knowledge Discovery and Data Mining (KDD'98)*, pages 2 – 8, 1998.

[BFO84] L. Breiman, J. Friedman, and R. Olshen. *Classification and Regression Trees*. Wadsworth International Group, 1984.

[BU91] C.E. Brodley and P.E. Utgoff. Linear machine decision trees. Technical Report TR-91-10, University of Massachusetts, Department of Computer and Information Science, Amherst, MA, 1991.

[CKJ05] C. Shou Chih, P. Hsing Kuo, and L. Yuh Jye. Model trees for classification of hybrid data types. In *Intelligent Data Engineering and Automated Learning - IDEAL: 6th International Conference*, pages 32 – 39, 2005.

[CV09] B. Chandra and P. Paul Varghese. Moving towards efficient decision tree construction. *Information Sciences*, 179(8):1059 – 1069, 2009.

[DH00] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. of Six Int. Conference on Knowledge Discovery and Data Mining, ACM Press*, pages 71 – 80, 2000.

[Die98] T.G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895 – 1923, 1998.

[FBS04] N. Marín F. Berzal, J.C. Cubero and D. Sánchez. Building multi-way decision trees with numerical attributes. *Information Sciences*, 165(1-2):73 – 90, 2004.

[FM99] Y. Freund and L. Mason. The alternating decision tree learning algorithm. In *16th International Conference on Machine Learning*, pages 124 – 133, 1999.

[Gal86] S.I. Gallant. Optimal linear discriminants. In *Proceedings of the International Conference on Pattern Recognition*, pages 849 – 852, 1986.

[Gar09] G. García. Measure of time and memory of a program. In *http://dis.um.es/ ginesgm/medidas.html. Faculty of Informatic, University of Murcia*, 2009.

[GGRL99] J. Gehrke, V. Ganti, R. Ramakrishnan, and W. Loh. BOAT - optimistic decision tree construction. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 169 – 180, 1999.

[GM05] J. Gama and P. Medas. Learning decision trees from dynamic data streams. *Journal of Universal Computer Science*, 11(8):1353 – 1366, 2005.

[GMR04] J. Gama, P. Medas, and R. Rocha. Forest trees for on-line data. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 632 – 636, 2004.

[GRG00] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. *Data Mining Knowledge Discovery*, 4(2/3):127 – 162, 2000.

[HK01] J. Han and M. Kamber. *Data Mining, Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[JA03] R. Jin and G. Agrawal. Efficient decision tree construction on streaming data. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 571 – 576, 2003.

[JAM08] S.S Allam J. Adelman-McCarthy, M.A. Agueros. Sloan digital sky survey. *Data Realease 6, ApJS*, 175(297), 2008.

[Jan98] C.Z. Janikow. Fuzzy decision trees: Issues and methods. *IEEE Transactions on Systems, Man and Cybernetics - Part B: Cybernetics*, 28(1):1 – 14, 1998.

[JMTAM02] J.R. García-Serrano J.F. Martínez-Trinidad and I.O Ayaquica-Martínez. C-means algorithm with similarity functions. *Computación y Sistemas*, 5(4):241 – 246, 2002.

[JOS09] N. Patel J. Ouyang and I. Sethi. Induction of multiclass multifeature split decision trees from distributed data. *Pattern Recognition*, 42(9):1786 – 1794, 2009.

[JZ06] S. Jiang and H. Zhang. A fast decision tree learning algorithm. In *American Association for Artificial Intelligence*, pages 500 – 505, 2006.

[KQ02] R. Kohavi and J.R. Quinlan. *Decision-tree discovery.* Will Klosgen and Jan M. Zytkow, editors. Oxford University Press, 2002.

[LW04] X. Llora and S. Wilson. Mixed decision trees: Minimizing knowledge representation bias in lcs. *Genetic and Evolutionary Computation. GECCO*, 3103:797 – 809, 2004.

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. 5th International Conference Extending Database Technology (EDBT), Avignon, France*, pages 18 – 32, 1996.

[Mit97] T. Mitchell. *Machine Learning.* McGraw Hill, 1997.

[NTC07] U. *N*guyen and C. Tae-Choong. An efficient decision tree construction for large datasets. In *Proc. 4th International Conference on Innovations in Information Technology*, pages 21 – 25, 2007.

[PMA+07] J. Pérez, J. Muguerza, O. Arbelaitz, I. Gurrutxaga, and J. Martín. Combining multiple class distribution modified subsamples in a single tree. *Pattern Recognition Letters*, 28(4):414 – 422, 2007.

[PS05a] W. Pedrycz and Sosnowski. C-fuzzy decision trees. *IEEE Transactions on Systems, Man and Cybernetics - Part C: Applications and reviews*, 35(4):498 – 511, 2005.

[PS05b] W. Pedrycz and Sosnowski. Genetically optimized fuzzy decision trees. *IEEE Transactions on Systems, Man and Cybernetics - Part B: Cybernetics*, 35(3):633 – 641, 2005.

[Qui93] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[RAS93] T. Imielinski R. Agrawal and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914 – 925, 1993.

[RD01] D.G. Stork R.O. Duda, P.E. Hart. *Pattern classification*. Wiley, New York, 2001.

[RM05] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers - a survey. *IEEE Transactions on Systems, Man and Cybernetics*, 35(4):476 – 487, 2005.

[RSyJMT99] J. Ruiz-Shulcloper and A. Guzmán-Arenas y J.F. Martínez-Trinidad. *Enfoque Lógico Combinatorio al Reconocimiento de Patrones*. Instituto Politécnico Nacional, 1999.

[SAM96] J.C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *roc. 22nd International Conference Very Large Databases*, pages 544 – 555, 1996.

[Sha48] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379 – 423 and 623 – 656, 1948.

[SL91] S.R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man and Cybernetics*, 21(3):660 – 674, 1991.

[SL97] Yu-Shan Shih and Wei-Yin Loh. Split selection methods for classification trees. *Statistica Sinica*, 7(4):815 – 840, 1997.

[TD07] O. Taner and O. Dikmen. Parallel univariate decision trees. *Pattern Recognition Letters*, 28(7):825 – 832, 2007.

[UB90] P.E. Utgoff and C.E. Brodley. An incremental method for finding multivariate splits for decision trees. In *Proc.7th International Conference on Machine Learning*, pages 58 – 65, 1990.

[UB95] P.E. Utgoff and C.E. Brodley. Multivariate decision trees. *Machine Learning*, 19(1):45 – 77, 1995.

[UBC97] P.E. Utgoff, N.C. Berkman, and J.A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(5):5 – 44, 1997.

[Utg89] P.E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161 – 186, 1989.

[Utg94] P.E. Utgoff. An improved algorithm for incremental induction of decision trees. In *Proc. 11th International Conference on Machine Learning*, pages 318 – 325, 1994.

[VL88] N. Vanichsetakul and Wei-Yin Loh. Tree-structured classification via generalized discriminant analysis. *Journal of the American Statistical Association*, 83(403):715 – 728, 1988.

[WM00] D.R. Wilson and T.R. Martínez. Reduction techniques for instance-based learning algorithms. *Machine Learning*, 38(3):257 – 286, 2000.

[WNS07] X. Wang, D. *N*auck, and M. Spott. Intelligent data analysis with fuzzy decision trees. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 11:439 – 457, 2007.

[WSJ06] C. Hsin Wei, O. Chen Sen, and L. Shie Jue. Improved c-fuzzy decision trees. In *IEEE International Conference on Fuzzy Systems*, pages 1763 – 1768, 2006.

[YAR99] H. Yoon, K. Alsabti, and S. Ranka. Tree-based incremental classification for large datasets. Technical Report TR-99-013, CISE Department, University of Florida, Gainesville, FL 32611, 1999.

[YWYC08] B. Yang, T. Wang, D. Yang, and L. Chang. BOAI: Fast alternating decision tree induction based on bottom-up evaluation. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining - PAKDD*, pages 405 – 416, 2008.

[ZLC07] R. Wang-Y. Yan Z. Li, T. Wang and H. Chen. A new fuzzy decision tree classification method for mining high-speed data streams based on binary search trees. In *Proceedings of FAW Conference*, pages 216 – 227, 2007.