Expert Systems with Applications 38 (2011) 14290–14300

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Decision tree induction using a fast splitting attribute selection for large datasets

A. Franco-Arcega [a,*], J.A. Carrasco-Ochoa [a], G. Sánchez-Díaz [b], J.Fco. Martínez-Trinidad [a]

[a] *Computer Science Department, National Institute of Astrophysics, Optics and Electronics, Luis Enrique Erro #1, Santa Maria Tonantzintla, Puebla, C.P. 72840, Mexico*
[b] *Computational Science and Technology Department, University of Guadalajara, CUValles, Carretera Guadalajara-Ameca Km. 45.5, Ameca, Jalisco, C.P. 46600, Mexico*

ARTICLE INFO

Keywords:
Decision trees
Large datasets
Gain-ratio criterion

ABSTRACT

Several algorithms have been proposed in the literature for building decision trees (DT) for large datasets, however almost all of them have memory restrictions because they need to keep in main memory the whole training set, or a big amount of it, and such algorithms that do not have memory restrictions, because they choose a subset of the training set, need extra time for doing this selection or have parameters that could be very difficult to determine. In this paper, we introduce a new algorithm that builds decision trees using a fast splitting attribute selection (DTFS) for large datasets. The proposed algorithm builds a DT without storing the whole training set in main memory and having only one parameter but being very stable regarding to it. Experimental results on both real and synthetic datasets show that our algorithm is faster than three of the most recent algorithms for building decision trees for large datasets, getting a competitive accuracy.

1. Introduction

Classification is an important task in data mining (Tan, Steinbach, & Kumar, 2006). Currently, there are many classification problems where large training datasets are available, therefore there is a big interest for developing classifiers that allow handling this kind of datasets in a reasonable time.

Decision trees (Quinlan, 1986, 1993) are commonly used for solving classification problems in Machine Learning and Pattern Recognition. A DT is formed by internal nodes, leaves, and edges, and it can be induced from a training set of instances, each one represented by a tuple of attribute values and a class label. Internal nodes have a splitting attribute and each node has one or more children (edges). Each one of these children has associated a value for the splitting attribute and these values determine the path to be followed during a tree traversal. Each leaf has associated a class label. In order to classify a new instance, the tree is traversed from the root to a leaf, when the new instance arrives to a leaf it is classified according to the class label associated to that leaf.

Several algorithms have been developed for building DTs from large datasets (Alsabti, Ranka, & Singh, 1998; Domingos & Hulten, 2000; Gehrke, Ramakrishnan, & Ganti, 1998, 2000, 1999; Mehta, Agrawal, & Rissanen, 1996; Shafer, Agrawal, & Mehta, 1996; Yang, Wang, Yang, & Chang, 2008; Yoon, Alsabti, & Ranka, 1999). However, almost all of them have spatial restrictions, because they have to keep the whole training set in main memory and some other use a representation of the attributes that requires more space than the whole training set. On the other hand, in those algorithms without spatial restrictions, the construction of the DT is based only on a small subset, but for obtaining this subset additional time is required, which could be too expensive for large training sets; or the algorithms uses several parameters, which could be very difficult to determine.

Having these drawbacks identified, this work introduces a new algorithm for building DTs that solves these problems. Our algorithm (DTFS) follows two main ideas for building DTs, it uses a fast splitting attribute selection for expanding nodes (deleting the instances stored in the expanded node after its expansion) and processes all the instances of the training set in an incremental way, therefore it is not necessary to store the whole training set in main memory.

In the literature some new techniques to select splitting attributes have been proposed (Berzal, Cubero, Marn, & Snchez, 2004; Chandra & Paul Varghese, 2009; Ouyang, Patel, & Sethi, 2009), however these techniques are not proposed for handling large datasets, because some of them have to evaluate a lot of candidate splits for choosing the best attribute, other use discretization methods to deal with numerical attributes, and some other use expensive techniques to expand nodes. On the other hand, several algorithms for building DTs in an incremental way have been proposed, such as ID5R (Utgoff, 1989), PT2 (Utgoff & Brodley, 1990), ITI (Utgoff, 1994), StreamTree (Jin & Agrawal, 2003) and UFFT (Gama & Medas, 2005), however these algorithms cannot handle large datasets either, because they need to keep the whole training set in main memory for building the DT.

In this paper, we propose an algorithm that processes the training instances one by one, thus each training instance traverses the

* Corresponding author. Tel.: +52 222 2663100x8311; fax: +52 222 2472580.
E-mail address: anifranco6@inaoep.mx (A. Franco-Arcega).

0957-4174/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2011.05.087

DT until a leaf is reached, where the training instance will be stored. In our algorithm, when a leaf has stored a predefined number of instances (a parameter of the algorithm), it will be expanded choosing a splitting attribute, using the instances in the leaf, and creating an edge for each class of instances in the leaf. After expanding a leaf, the instances stored in that leaf are deleted. Experimental results over several large datasets show that our algorithm is faster than three of the most recent algorithms for building DTs for large datasets, obtaining a competitive accuracy.

The rest of the paper is organized as follows. Section 2 gives an overview of the works related to DT induction for large datasets. Section 3 introduces the DTFS algorithm, which allows building DTs for large datasets. Section 4 provides experimental results and a comparison against other algorithms for DT induction for large datasets, on both real and synthetic datasets. Finally, Section 5 gives our conclusions and some directions for future work.

2. Related work

In this section, several algorithms that have been proposed to build DTs for large datasets are described.

Mehta et al. (1996) presented SLIQ (Supervised Learning In Quest), an algorithm for building DTs for large datasets. This algorithm uses a list structure for each attribute, these lists are used in order to avoid storing the whole training set in main memory, by storing them in disk. However, SLIQ uses an extra list that must be stored in main memory, this list contains the class of each instance and the number of the node where this instance is stored in the tree. This could be a problem for large datasets, because the size of this list depends on the number of instances in the training set. The process that SLIQ follows to build a DT is similar to C4.5, but the difference is that SLIQ uses the lists for splitting attribute selection, therefore, the lists must be read from disk each time a node is going to be expanded.

Shafer et al. (1996) presented an improvement of SLIQ, called SPRINT (scalable parallelizable induction of decision trees). The difference with respect to SLIQ lies in how SPRINT represents the lists for each attribute. SPRINT adds a column to each list for storing the class of each instance, hence SPRINT does not need to store in main memory any whole list. However, since SPRINT has to read from disk all the lists for expanding each node, just like SLIQ, the runtime is too large if the training set has a lot of instances.

Alsabti et al. (1998) proposed CLOUDS (Classification for Large or OUt-of-core DataSets), an algorithm that uses, as SLIQ and SPRINT, lists for representing the information of the attributes in the training set. However, these lists are simplified representing the numerical attributes by intervals. This modification substantially reduces the time required for choosing the attributes that will represent the internal nodes of the DT, because it is not needed to check all the values for each attribute. A drawback of SPRINT and CLOUDS is that for storing the lists they require at least the double of the space needed for storing the original training set.

Gehrke et al. (1998), Gehrke, Ramakrishnan, and Ganti (2000) introduced the Rainforest algorithm. It follows the idea of using lists for representing the attributes of a training set, but this algorithm only stores all the different values for each attribute. In this way, Rainforest tries to reduce the size of the lists, thus the list size will not be the number of training instances but the number of different values. However, these lists must be stored in main memory, therefore if the attributes have a lot of different values in the training set, the available space could be not enough. Besides, in order to generate the lists, in each level of the DT, Rainforest has to read the whole training set twice and write it once, which is very expensive for large datasets. Nguyen and Tae-Choong (2007) presents an improvement of Rainforest, the difference is that this improvement adds to the lists, used by Rainforest, the position of each instance in the training set, in order to use them in the expansion of each node, in this way this algorithm does not have to read twice and write once the whole training set in each level of the tree. This algorithm only scans once the whole training set, however if an attribute has too many values, its lists may not fit in main memory.

Gehrke, Ganti, Ramakrishnan, and Loh (1999) developed an incremental algorithm for building DTs, called BOAT (Bootstrapped Optimistic Algorithm for Tree construction). This algorithm avoids to store the whole training set in main memory by using only an instance subset as training for building the DT, however obtaining this subset requires additional time for building the DT, which could be expensive for large datasets. Starting from this subset, BOAT applies a bootstrapping technique for generating multiple DTs, using a traditional main memory DT induction algorithm (for example C4.5, CART, etc.). The constructed DTs are combined, and finally, BOAT refines the combined DT using the whole training set.

Yoon et al. (1999) proposed another incremental algorithm for building DTs, called ICE (Incrementally Classifying Ever-growing large datasets). This algorithm divides the training set in subsets, called epochs, and processes them separately, therefore ICE does not need to store the whole training set in main memory. ICE builds a DT for each epoch using a traditional main memory DT induction algorithm (as C4.5, CART, etc.) and from each DT, ICE obtains a subset of instances applying a sampling technique. Then ICE joins the subsets, obtained from each epoch, for building the final DT. A DT T_i is built from each subset D_i (epoch i) of the training set, and using a sampling technique a set S_i of samples is extracted from T_i. The union of S_i and the previous sets of samples are stored in U_i. Then the new set of samples $U_i = U_{i-1} \cup S_i$ is preserved for building the DT in the next epoch. For a training set divided in k epochs, ICE joins $S_1, S_2, \ldots, S_k$, the subsets of instances extracted from $T_1, T_2, \ldots, T_k$, and builds the final DT C_k with the last U_k, the algorithm preserves the subset U_k and the DT C_k. If a new epoch D_{k+1} must be processed, ICE builds T_{k+1} from D_{k+1}, extracts S_{k+1} from T_{k+1} and uses $U_{k+1} = U_k \cup S_{k+1}$ for building the new DT, C_{k+1}. A drawback of ICE is that when the algorithm processes large training sets, it spends a lot of time for obtaining the subset of instances for building the final DT.

Domingos and Hulten (2000) introduced an incremental algorithm called VFDT (very fast decision trees). This work proposed the Hoeffding trees, which can be learned in constant time per instance, and they are similar to the trees built by traditional main memory DT induction algorithms (for example C4.5, CART, etc.). For building the DT, VFDT needs the training instances in a random order, if this is not the case, they should be randomized in a preprocessing step. VFDT starts with a tree produced by a conventional DT induction algorithm, this tree is built from a small subset of instances, then VFDT processes each instance of the training set traversing the DT and updating the statistics needed to compute the information gain of each attribute in the leaf in which the instance arrives. VFDT uses a user-defined parameter n_{min}, which indicates the minimum number of instances that must be stored in a leaf before checking if the node has enough information to be expanded. When n instances have arrived to a leaf, VFDT obtains the information gain of each attribute using the statistics stored in the leaf, chooses the two attributes with highest information gain, $G(X_a)$ and $G(X_b)$, and obtains the Hoeffding bound ε using Eq. (1).

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

In Eq. (1) R is $log(c)$ (c is the number of classes) and δ is a user-defined parameter, where $1 - \delta$ indicates the probability of choosing the correct splitting attribute. If the difference between the information gain of the two best attributes ($\Delta G = |G(X_a) - G(X_b)|$) is greater than the Hoeffding bound ($\Delta G > \varepsilon$) then the leaf must be expanded, if $\Delta G \leqslant \varepsilon$, the leaf must receive more instances before it would be expanded. VFDT uses n_{min} to avoid checking if $\Delta G > \varepsilon$ each time an instance arrives to a leaf, which would lead a large runtime for building the DT. If $\Delta G = 0$, VFDT uses another user-defined parameter τ to decide if the current best attribute can be chosen as splitting attribute (if $\varepsilon < \tau$ then VFDT uses the best attribute as splitting attribute). VFDT finishes when all the instances have been processed. This algorithm uses three parameters that could be too difficult to be defined by the user. Besides, for expanding a node, VFDT has to evaluate for each numerical attribute all the possible splits, which could be too expensive if the attributes have a lot of different values.

Yang et al. (2008) developed a DT induction algorithm, called BOAI (BOttom-up evaluation for ADTree Induction), based on the ADT algorithm Freund and Mason, 1999, which is a DT induction algorithm for two-class problems. A DT built by ADT, called ADTree, contains two kinds of nodes, decision and prediction nodes. The ADTree consists of alternating levels of prediction and decision nodes, each prediction node is associated with a weight, which represents its contribution to the final prediction score, while each decision node has a splitting attribute. ADT associates a weight to each training instance Freund and Mason, 1999, which is used for computing the splitting attribute each time a prediction node is expanded as well as for computing the prediction values associated to new prediction nodes. ADT uses an iterative process to build the DT, it starts with a root prediction node and chooses the best splitting attribute for expanding this node, creating a decision node and two prediction nodes associated to that decision node. At each iteration, ADT expands a prediction node, traversing the DT using a top-down approach and evaluating all possible splits for all attributes in all prediction nodes, in order to find the best splitting test attribute. A new decision node can be attached to any previous prediction node, not only at the leaves. ADT continues expanding prediction nodes until a predefined number of iterations is reached. The classification of a new instance with an ADTree is the sign of the sum of the prediction values along all paths that this instance can traverse in the DT.

BOAI is an alternative way of building an ADTree for large datasets. For each attribute, BOAI creates a list which stores the weights associated to the training instances, these lists are stored in the prediction nodes and are used for computing the splitting attribute when a prediction node is going to be expanded. BOAI uses a bottom-up approach to choose the node to be expanded in each iteration, instead of a top-down approach. The bottom-up approach allows BOAI to maintain only the group of lists in the last prediction node of each path in the DT (leaves), since these lists allow creating the lists associated to previous prediction nodes in each path. Besides, BOAI sorts the values of numeric attributes as a preprocessing step, in order to avoid sorting these values each time BOAI evaluates all possible splits for expanding a prediction node. A drawback of BOAI is the size of the attributes' lists, because their size depends on the number of different attribute values in the training set, and therefore these lists could be too big for large datasets.

From the algorithms for building DT for large datasets above described, we can notice that SLIQ, SPRINT and CLOUDS use a representation of the attributes that requires at least the double of the space required for storing the whole training set; Rainforest and BOAI use structures for storing the training set, but these structures cannot be stored in main memory when a large training set is being processed; VFDT needs three parameters for building the DT, which could be very difficult to define by the user; and BOAT and ICE use only a subset of the training set for building the DT, but these algorithms require a lot of time for finding this subset of instances for large training sets. To overcome these shortcomings, we propose a new DT induction algorithm for large training sets, called DTFS, which requires only one parameter for building a DT, but DTFS is very stable respect to this parameter (as we will show in the experimental results). Besides, our algorithm uses a fast splitting attribute selection and it processes the whole training set, without storing it in main memory.

3. Proposed algorithm

In this work, we propose a new algorithm for building DTs for large datasets (DTFS) that overcomes the shortcomings of the related algorithms. In order to avoid storing the whole training set in main memory, DTFS builds DTs in an incremental way. Thus the training instances will be processed one by one, traversing the DT with each one, until it reaches a leaf, where the instance will be stored. Besides, to avoid storing all the training instances into the tree, a leaf stores at most *s* instances (*s* is a parameter of DTFS), and when the number of instances stored in a leaf reaches this limit, DTFS expands or updates the leaf according to the instances stored in it, and in both cases those instances are discarded. If the leaf contains instances from two or more classes, DTFS will expand the leaf choosing a splitting attribute and creating one edge for each class of the stored instances (each edge will have a value that represents the instances of the corresponding class). But if the leaf contains instances from only one class, our algorithm only will update the value of the input edge of the leaf.

This strategy helps DTFS to maintain only a small number of instances in main memory for building a DT. Additionally, since the expansion of a leaf is done attending only the *s* instances stored in it, this strategy also allows a fast splitting attribute selection, in comparison to the conventional splitting in DTs.

Since DTFS processes the training instances in an incremental way, processing a long sequence of instances from only one class would lead to represent them into a single node of the DT, because DTFS only will update the input value of the node. In order to avoid this situation, DTFS (function *ReorganizeTS*, called in the DTFS algorithm in Fig. 2) reorganizes the training set alternating instances from each class, i.e., the first instance from the first class, the second instance from the second class and so on, if there are *r* classes, the instance in the position $r + 1$ will be from class 1 and so on. In the experiments the time required for this reorganization will be included as part of the DTFS runtime.

The structure of the DT built by DTFS is similar to a traditional DT, it will contain in each internal node a splitting attribute, each edge will correspond to a possible outcome of the splitting attribute and each leaf will have associated a class label.

3.1. Splitting attribute selection

One of the most commonly used criteria for splitting attribute selection in DTs is Gain Ratio Quinlan, 1993. However, in traditional main memory DT induction algorithms this criterion becomes very expensive for large datasets. This fact is because for choosing the best splitting attribute, these algorithms have to select the best splitting attribute for each non homogeneous node, based on the instances stored in the node (usually a lot) and using the selected splitting attribute they create several edges, one for each different value of each attribute, and the instances stored in the node (which as we mentioned usually are a lot) are classified according to these edge values.

In order to do a fast selection of the splitting attribute, DTFS uses the Gain Ratio Criterion in an different way. For choosing the best splitting attribute when a leaf has to be expanded, only the *s* instances stored in that leaf are taken into account. The idea is to choose, as splitting attribute, the attribute and a set of splitting values for it (one for each class) that best reconstruct the partition defined by the classes of instances in the leaf to be expanded. Hence, for each attribute, DTFS computes for each class, as splitting value, the mean of the attribute values appearing in the instances of the leaf belonging to that class. After, for each attribute, our algorithm classifies the instances in the leaf using the computed splitting values (finding the closest mean for each instance, according to the current attribute), and measures the Gain Ratio as in Quinlan (1993) but over these new partitions. Finally the attribute jointly with its associated splitting values, which produce the best partition (with the maximum Gain Ratio), is selected. If the maximum is reached by more than one attribute the first one is selected. Measuring the Gain Ratio in this way, for each attribute only one set of splitting values is evaluated using only the *s* instances in the node, which allows a fast selection.

3.2. Building the DT

DTFS starts with an empty root node (a leaf) and processes one by one the instances of the training set, traversing the DT with each one until it reaches a leaf, where the instance will be stored.

When a leaf reaches the maximum number of instances allowed by DTFS (*s* instances), DTFS follows one of the following cases. If a leaf has instances from two or more classes, DTFS replaces the leaf by an internal node. To expand a node, DTFS obtains the splitting attribute that best divides the instances (i.e. the one which produces the most homogeneous partition) based on the criterion described in Section 3.1, and then DTFS creates an edge connected to a new empty leaf for each class of instances in the node. After, DTFS assigns to each edge the mean computed for the selection process. The splitting attribute and the values of the edges will be used by DTFS for traversing the DT. Finally, DTFS deletes the instances that were used for expanding the node.

On the other hand, when a leaf has instances from a single class, our algorithm does not expand the node, it keeps the node as a leaf, and DTFS only updates the value of the splitting attribute associated to the input edge. For this updating, DTFS computes the mean of the values for the splitting attribute from the instances stored in the leaf, and the new value of the splitting attribute will be the average between this value and the value of the splitting attribute associated to the input edge. Finally, DTFS deletes the instances stored in it.

The DTFS's pseudocode for expanding a node is shown in Fig. 1.

The induction of a DT finishes when all the instances of the training set have been processed. Finally, our algorithm assigns to each leaf the label of the majority class of the instances stored in it or the label of the class from which the input edge was created, if the leaf is empty.

The DTFS's pseudocode for building the DT is shown in Fig. 2.

```
ExpandNode(NODE)
   NODE is the node in the tree to be expanded

   If NODE.classes > 1, then
      For each attribute Xi ∈ TS, do
         For each class Cj ∈ NODE, do
            M[j] = ObtainMean(Cj, NODE.Ins, NODE.Attr)
         End For
         GR[i] = gainratio(M, Xi, NODE.Ins)
      End For
      NODE.Attr = chooseBestAttr(GR)
      For each class Ci ∈ NODE, do
         Ri = CreateEdge()
         Ri = AssignMean(M, NODE.Attr)
      End For
      For each edge Ri ∈ NODE, do
         leafi = CreateNode()
      End For
      Delete(NODE.Ins)
   else
      V = ObtainValue(C, NODE.Ins, NODE.Input.Attr)
      Combine(V, NODE.Input.Ve, NODE.Input.Attr)
      Delete(NODE.Ins)
      NODE.numIns = 0
   End If
```

Fig. 1. ExpandNode function.

```
DTFS (TS, s)
   TS is the training set
   s is the maximum number of instances in the nodes

   ReorganizeTS(TS)
   ROOT = CreateNode()
   For each I ∈ TS, do
      UpdateDT(I,ROOT)
   End For
   AssingClassToLeaves()

UpdateDT (I, NODE)
   I is the instance to be processed
   NODE is the node in the tree to be traversed

   If NODE.numIns < s, then
      AddInsNode(NODE, I)
      NODE.numIns = NODE.numIns + 1
      If NODE.numIns = s, then
         ExpandNode(NODE)
         NODE.numIns = NODE.numIns + 1
      End If
   else /* NODE.numIns > s */
      For each edge Rj ∈ NODE, do
         simi[j] = Similarity(I, NODE.ORj)
      End For
      Edge = max(simi[i])
      UpdateDT(I, NODE.Edge)
   End If
```

Fig. 2. DTFS algorithm.

3.3. Traversing the DT

For traversing the DT, an instance starts at the root and descends through the internal nodes until the instance reaches a leaf. To descend to a node, DTFS follows the path of the splitting attribute that best matches with the corresponding value of the attribute of the instance that is traversing the DT. It is done computing the smallest absolute difference between the instance's value and the edge's value.

3.4. Classifying instances

The classification process in DTFS, as in all DT algorithms, consists in traversing the DT with an unseen instance until a leaf is reached, and assigning to the new instance the class label associated to that leaf.

3.5. Time complexity analysis of DTFS algorithm

For building a DT from a large dataset, DTFS has to traverse the DT with each training instance until it reaches a leaf, in which the instance is stored. Then, when a leaf has *s* instances stored in it, DTFS expands the node, choosing the best splitting attribute and creating an edge for each class of instances in the node.

For a training set of *m* instances, described by *d* attributes and divided in *k* classes, traversing the DT for each instance is, in the worst case, $O(k*log_k(m))$, since an instance has to choose among at most *k* edges per internal node to descend at most $log_k(m)$ levels of the tree, then traversing the DT with all *m* instances is $O(m*k*log_k(m))$, however, since for large datasets $k \ll m$

(commonly there are only a few classes) the complexity for traversing the DT with the m instances of the training set is $O(m*log_k(m))$.

In the expansion process, for a single node, DTFS chooses the best splitting attribute applying to each attribute the Gain Ratio Criterion (as we explained in Section 3.1) but using only the s instances stored in the leaf to be expanded, thus, DTFS computes for each attribute the mean of each class of instances stored in the leaf, then selecting the splitting attribute is $O(d*s)$. Once the splitting attribute has been chosen, DTFS creates at most k edges and assigns to each edge the mean computed from the instances belonging to the corresponding class, this process is $O(k)$. Therefore, the expansion process for a single node is $O((d*s) + k) = O(d*s)$ and the maximum number of expansions that DTFS does for building a DT is $O(m/s)$, then the whole expansion process in DTFS is $O((d*s)*(m/s)) = O(d*m)$, however, since for large datasets $d \ll m$ the complexity of all the expansions for building a DT is $O(m)$.

Finally, the complexity for building a DT using the DTFS algorithm is the sum of traversing and expansion components:

$$O(m*log_k(m) + m) = O(m*log_k(m)).$$

Yoon et al. (1999) analyzed the complexity of ICE algorithm, which is $O(m*log(m))$. The complexity of BOAI is also $O(m*log(m))$ Yang et al., 2008 and the complexity of VFDT is $O(m^2)$ (Li, Wang, Wang, Yan, & Chen, 2007). As it can be observed, the complexity of DTFS, is lower than VFDT's and is the same than ICE's and BOAI's. The difference among DTFS, VFDT, ICE and BOAI is in the effective cost of building the DT. Therefore, in Section 4 we show an experimental analysis of the runtime spent by each algorithm for building DTs for large datasets.

3.6. Spatial complexity analysis

For building the DT, DTFS has to keep in main memory only the instance that is being processed at each moment and the DT built with the previous instances. The maximum number of expansions that DTFS can do is m/s, then the space required for our algorithm is $O(m/s) = O(m)$.

The space that ICE needs for building the DT depends on the number of epochs (e) that is established by the user and the proportion of instances ($0 < p \leqslant 1$) that must be extracted from each epoch. Each epoch contains m/e instances, thus the size of each subsample is $p*(m/e)$, therefore the space required for storing all the subsamples is $p*(m/e)*e = p*m$. Finally, since only one epoch and all the subsamples need to be stored at each moment, the number of instances that ICE needs to store is $O((p*m) + (m/e)) = O(p*m) = O(m)$.

VFDT needs to store the instance that is being processed at each moment and the DT built previously. Therefore, the space required for this algorithm depends on the maximum number of expansions that VFDT can make, which is m/n, where n is the number of instances used for verifying if a node must be expanded, then the space required by VFDT is $O(m/n) = O(m)$.

BOAI is an algorithm that needs to store the whole training set in main memory for building the DT. The instances are stored in the leaves of the DT as a set of lists, then the space that BOAI requires is $O(m)$.

As it can be noticed, the space required by BOAI, VFDT, ICE and DTFS for building a DT is, in the worst case, $O(m)$.

4. Experimental results

The experiments were conducted in four directions. First in Section 4.1 we analyze the behavior of our algorithm when the parameter s (the maximum number of instances in a leaf) varies. In Section 4.2 we analyze the behavior of DTFS when the number of attributes in the training set varies, since we want to show that our algorithm does a fast selection of the splitting attribute no matter the number of attributes in the dataset. Additionally, in Section 4.3 we present a comparison among DTFS and three of the most recent algorithms for building DTs for large datasets, ICE (Yoon et al., 1999), VFDT (Domingos & Hulten, 2000) and BOAI (Yang et al., 2008), using both synthetic and real datasets. Finally, Section 4.4 shows a significance test, in order to study the difference in runtime and accuracy rate between DTFS' results and those results obtained by ICE, VFDT and BOAI.

In all the experiments we evaluated the runtime (including induction and classification time, and for DTFS also the time for the preprocessing step) and the accuracy rate over a testing set. For all experiments, we used 10-fold cross validation and all figures include the 95% confidence interval. However, in some cases these confidence intervals are not visible because they are very small. All our experiments were performed on a PC computer with a Pentium 4 at 3.06 GHz, with 2 GB of RAM, running Linux Kubuntu 7.10.

The datasets used for the experiments are described in Table 1. The first three are real datasets and the remaining are synthetic datasets.

Table 1
Datasets used in the experiments.

Dataset	# Classes	# Instances	# Attributes
Poker (UCI Machine Learning Repository, 2008)	2	923,707	10
SpecObj* (SDSS-Adelman-McCarthy et al., 2008)	6	884,054	5
GalStar* (SDSS-Adelman-McCarthy et al., 2008)	2	4,000,000	30
Synthetic_1	2	4,000,000	5
Synthetic_2	3	4,000,000	5
Synthetic_3	5	4,000,000	5
Synthetic_4	2	4,000,000	40
Synthetic_5	3	4,000,000	40
Synthetic_6	5	4,000,000	40

4.1. DTFS' parameter

For the three real datasets in Table 1, we applied DTFS over a training set and a test set containing 100,000 instances each one, taking the first instances of each class and maintaining the class proportion. For each dataset $s = 50$ and s from 100 to 600 with increments of 100 were evaluated.

Fig. 3 shows the runtime and the accuracy rate obtained by DTFS when the value of the parameter s varies. As we can observe, when the value of s increases the runtime slightly decreases, but the accuracy rate also decreases. We chose $s = 100$, which is a value that maintains a good compromise between runtime and accuracy.

4.2. Increasing the number of attributes

In order to study the performance of DTFS when the number of attributes increases, we generated several synthetic datasets with different number of attributes, from 5 to 40 attributes with increments of 5. These datasets were randomly generated following the normal distribution with mean M_{ij} and standard deviation SD_{ij} for each class C_i and each attribute j. We generated instances for two, three and five classes, with values of M_i and SD_i as appear in Table 2 (the values in Table 2 correspond to the 5-attribute synthetic datasets, for the synthetic datasets with a number of attributes greater than five, we repeated the first two means to get a number of means equal to the number of attributes; we used the same standard deviation for all the attributes).

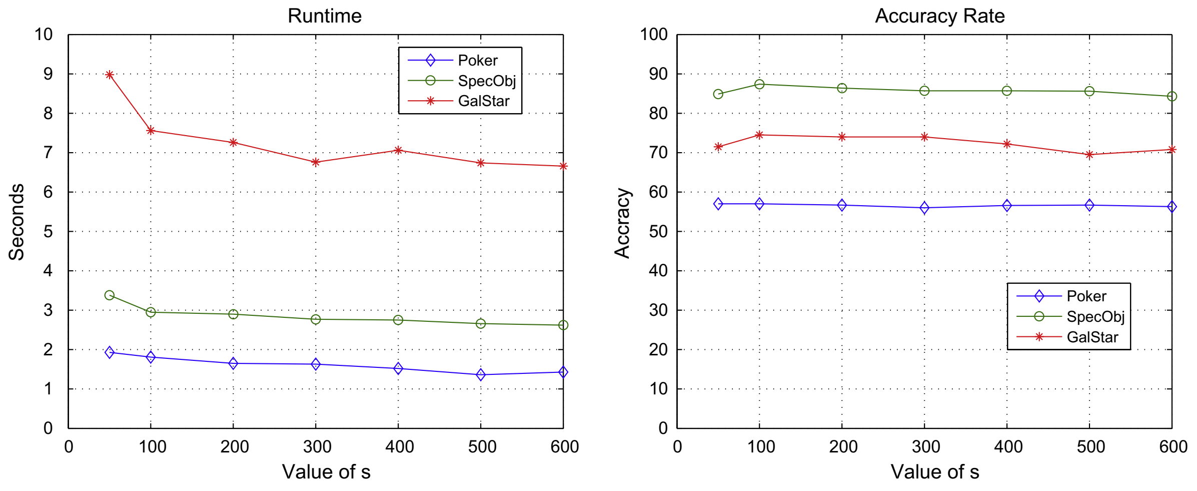

Fig. 3. Runtime and accuracy rate varying the value of *s*.

Table 2
Mean and standard deviation for synthetic datasets.

Two-class	$M_1 = [0, 0, 0, 0, 0]$	$SD_1 = [0.6, 0.6, 0.6, 0.6, 0.6]$
	$M_2 = [1, 1, 1, 1, 1]$	$SD_2 = [0.6, 0.6, 0.6, 0.6, 0.6]$
Three-class	$M_1 = [0.2, 0.5, 0.2, 0.5, 0.2]$	$SD_1 = [0.15, 0.15, 0.15, 0.15, 0.15]$
	$M_2 = [0.3, 0.9, 0.3, 0.9, 0.3]$	$SD_2 = [0.15, 0.15, 0.15, 0.15, 0.15]$
	$M_3 = [0.7, 0.9, 0.7, 0.9, 0.7]$	$SD_3 = [0.10, 0.10, 0.10, 0.10, 0.10]$
Five-class	$M_1 = [0.2, 0.5, 0.2, 0.5, 0.2]$	$SD_1 = [0.15, 0.15, 0.15, 0.15, 0.15]$
	$M_2 = [0.3, 0.9, 0.3, 0.9, 0.3]$	$SD_2 = [0.15, 0.15, 0.15, 0.15, 0.15]$
	$M_3 = [0.7, 0.9, 0.7, 0.9, 0.7]$	$SD_3 = [0.10, 0.10, 0.10, 0.10, 0.10]$
	$M_4 = [0.9, 0.4, 0.9, 0.4, 0.9]$	$SD_4 = [0.15, 0.15, 0.15, 0.15, 0.15]$
	$M_5 = [0.5, 0.0, 0.5, 0.0, 0.5]$	$SD_5 = [0.10, 0.10, 0.10, 0.10, 0.10]$

To show the distribution of the synthetic datasets, some datasets with two attributes were generated, these datasets are shown in Fig. 4.

Fig. 5 shows the runtime and the accuracy rate obtained with DTFS, ICE and VFDT when the number of attributes varies in the synthetic datasets with two, three and five classes.

We can notice from Fig. 5 that the runtime of DTFS only has a small variation when the number of attributes in the training set is increased, while the runtimes of ICE and VFDT increase a lot. In a DT induction algorithm, the number of attributes in a dataset is mainly related to the selection of the splitting attributes, since for each expansion of a node each attribute must be evaluated to choose the best one. For these datasets the accuracy rate of each algorithm was very similar no matter the number of attributes in the training set.

4.3. Comparison against other DT induction algorithms for large datasets

This section presents the comparison of DTFS against three of the most recent algorithms for building DTs for large datasets. The algorithms used to compare our algorithm were ICE, VFDT and BOAI. For our experiments we implemented ICE based on Yoon et al. (1999), extracting 10% of the instances, as the subsample for building the DT (this percentage of instances was chosen according to the experiments presented by Yoon et al. (1999)), for BOAI we got the authors' version, establishing 30 iterations for building the DT (this value was chosen based on Yang et al. (2008)) and for VFDT we also got the authors' version (for this algorithm we did the experiments with the default values of the program).

In order to show the performance of DTFS, first we show a comparison on the real datasets (Poker, SpecObj and GalStar) and then we present a comparison using the synthetic datasets. In these experiments the performance of the algorithms was evaluated when the size of the training set increases.

4.3.1. Real datasets

For Poker we created different-size training sets (from 50,000 to 500,000 with increments of 50,000). Fig. 6 shows the runtime and the accuracy obtained for this dataset with the four algorithms. Based on these results it can be seen that DTFS, ICE, VFDT and BOAI obtained similar accuracy for this dataset, and DTFS is one of the fastest algorithms.

For SpecObj, we created different-size training sets (from 50,000 to 500,000 with increments of 50,000). With this dataset

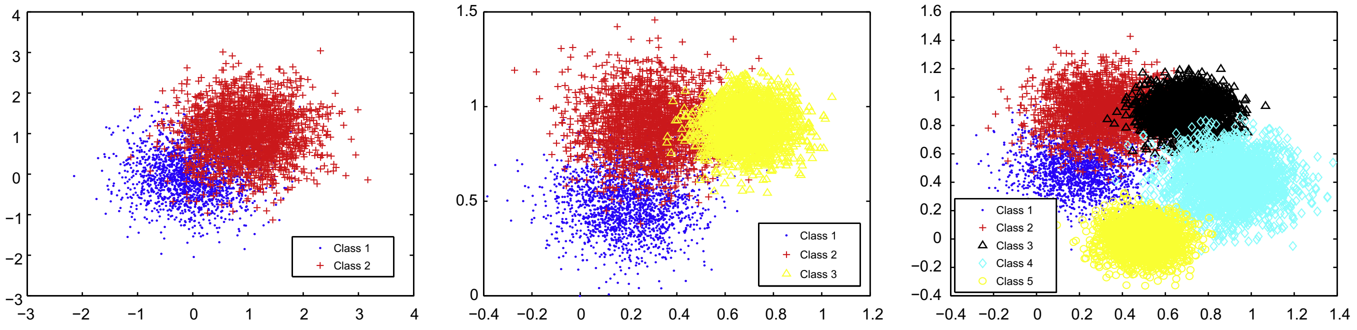

Fig. 4. Synthetic datasets with two, three and five classes.

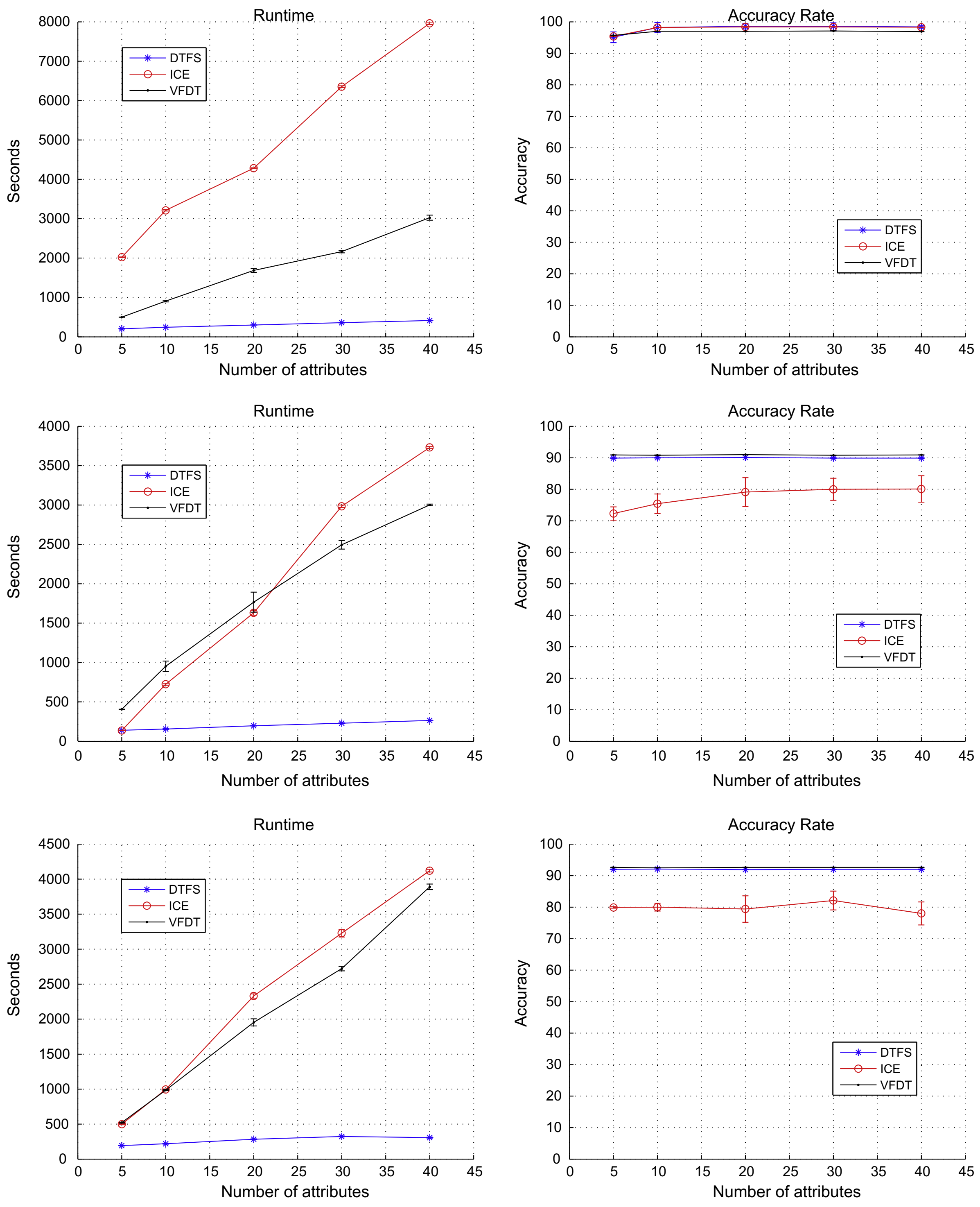

Fig. 5. Runtime and accuracy rate for DTFS, ICE and VFDT with 2-class, 3-class and 5-class synthetic dataset, respectively, when the number of attributes increases.

we only compared our algorithm against ICE and VFDT, because BOAI is only for two-class problems. Fig. 7 shows the runtime and the accuracy for SpecObj. As it can be seen, our algorithm was similar than ICE and VFDT in accuracy but DTFS was about 13 and 9 times faster than ICE and VFDT, respectively.

For GalStar we created different-size training sets (from 500,000 to 4,000,000 with increments of 500,000). Fig. 8 presents the comparison between DTFS, ICE and VFDT. BOAI does not appear in this figure because for this dataset it could not process training sets with more than 300,000 instances. As it can be noticed DTFS, ICE and VFDT obtained a similar accuracy rate, but DTFS was about 56 and 8 times faster than ICE and VFDT, respectively.

4.3.2. Synthetic datasets

For these experiments we used the 5-attribute and 40-attribute synthetic datasets created for the experiment of Section 4.2. BOAI is only included in the results for the 5-attribute synthetic dataset with two classes, because for the 40-attribute synthetic dataset

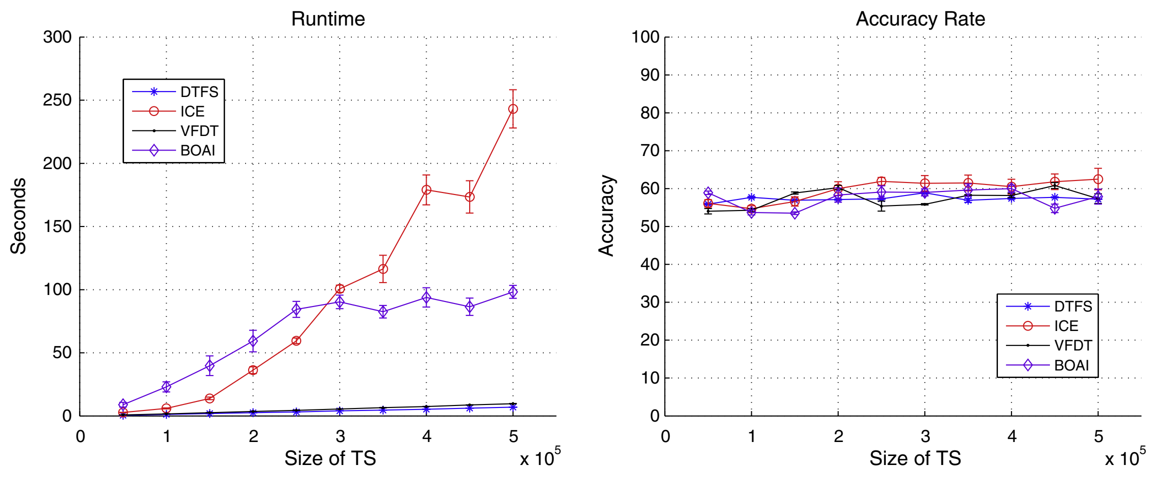

Fig. 6. Runtime and accuracy rate for DTFS, ICE, VFDT and BOAI algorithms for Poker.

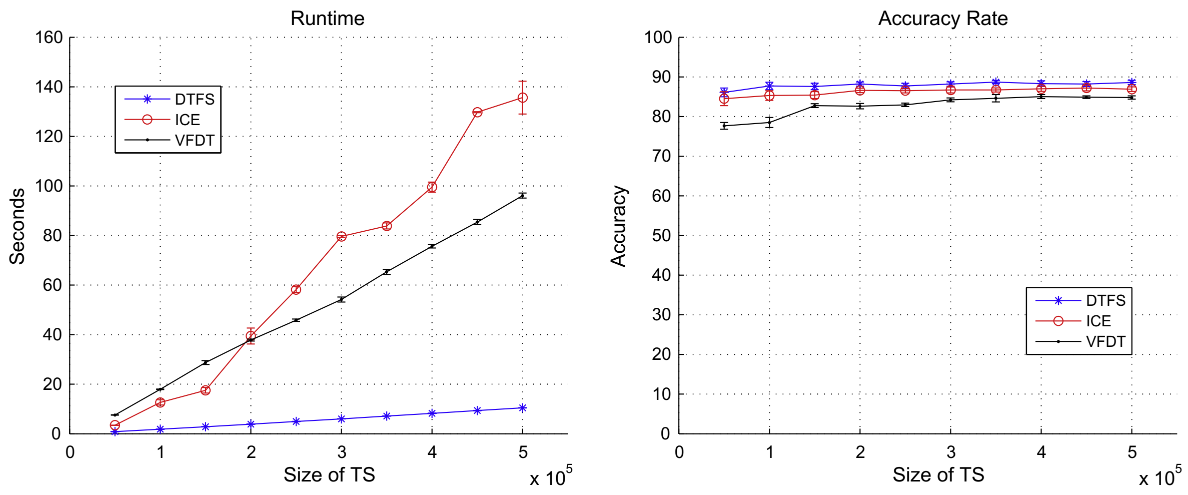

Fig. 7. Runtime and accuracy rate for DTFS, ICE and VFDT algorithms for SpecObj.

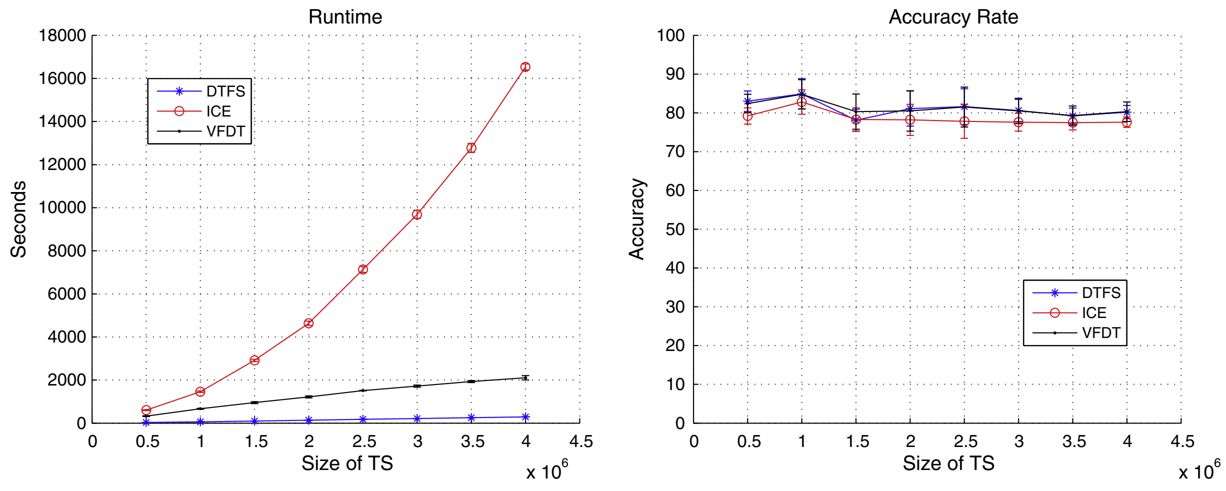

Fig. 8. Runtime and accuracy rate for DTFS, ICE and VFDT algorithms for GalStar.

with two classes the program failed, and the remaining synthetic datasets are not two-class problems.

Fig. 9 shows the runtime and the accuracy rate obtained with the 5-attribute synthetic datasets with two, three and five classes.

As it can be noticed in Fig. 9, DTFS is better than ICE and VFTD, since DTFS is the fastest algorithm for the three datasets and obtains a higher accuracy rate than ICE's and a similar accuracy rate with VFDT.

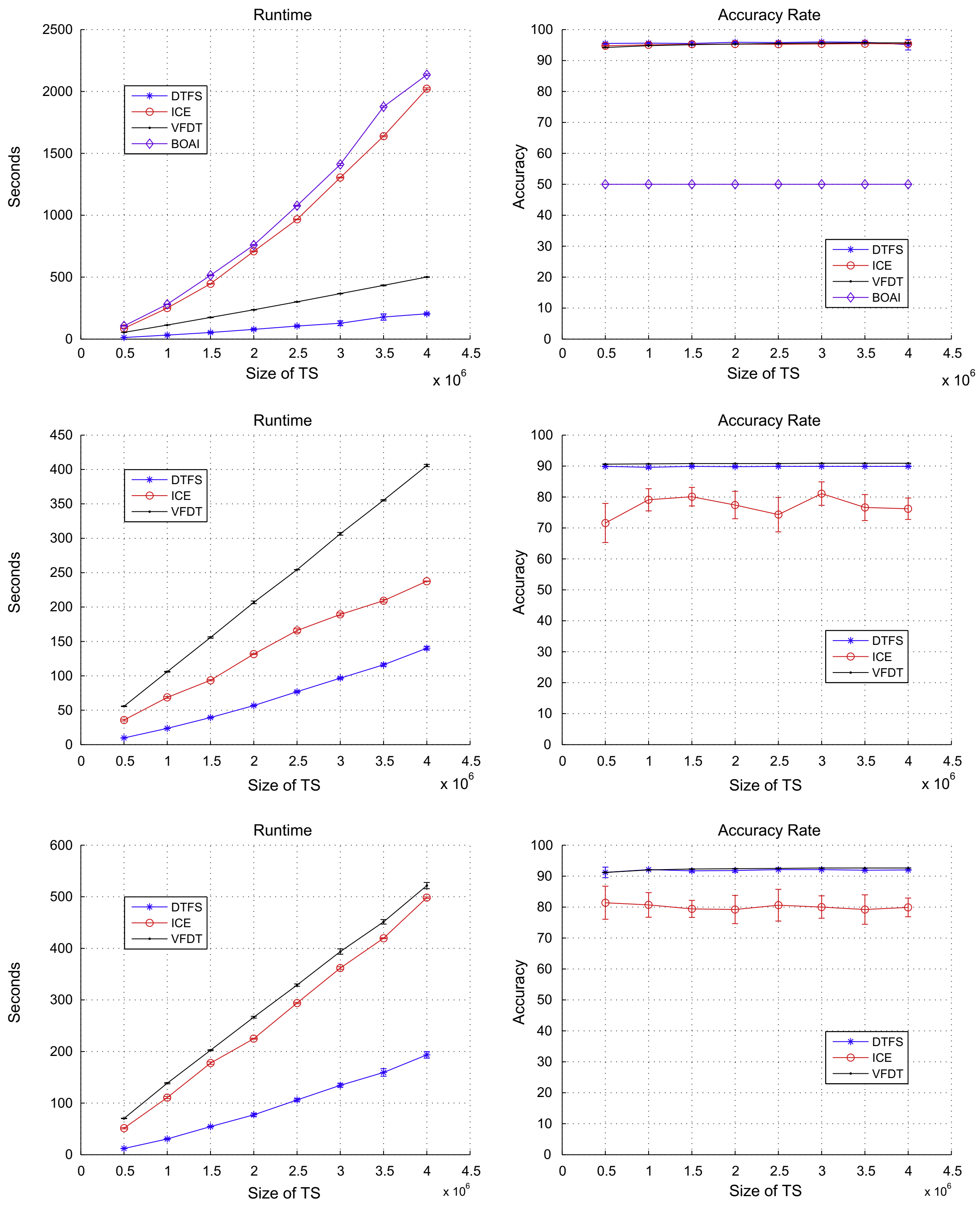

Fig. 9. Runtime and accuracy rate for DTFS, ICE, VFDT and BOAI with 2-class, 3-class and 5-class for the 5-attribute synthetic datasets, respectively.

In Fig. 10 we can observe the runtime and the accuracy rate obtained with the 40-attribute synthetic datasets with two, three and five classes.

DTFS and VFDT obtained similar accuracy rate for these three synthetic datasets, and their accuracies were better than ICE's accuracies for the 3-class and 5-class datasets. However, DTFS was faster than ICE and VFDT for all the datasets, for the 2-class datasets our algorithm was 19 and 7 times faster, for the 3-class dataset it was 14 and 11 times faster and for the 5-class dataset it was 13 and 12 times faster, than ICE and VFDT, respectively.

4.4. Significance test

In order to evaluate if the difference among DTFS, ICE, VFDT and BOAI is statistically significant, we applied over the 10-fold cross

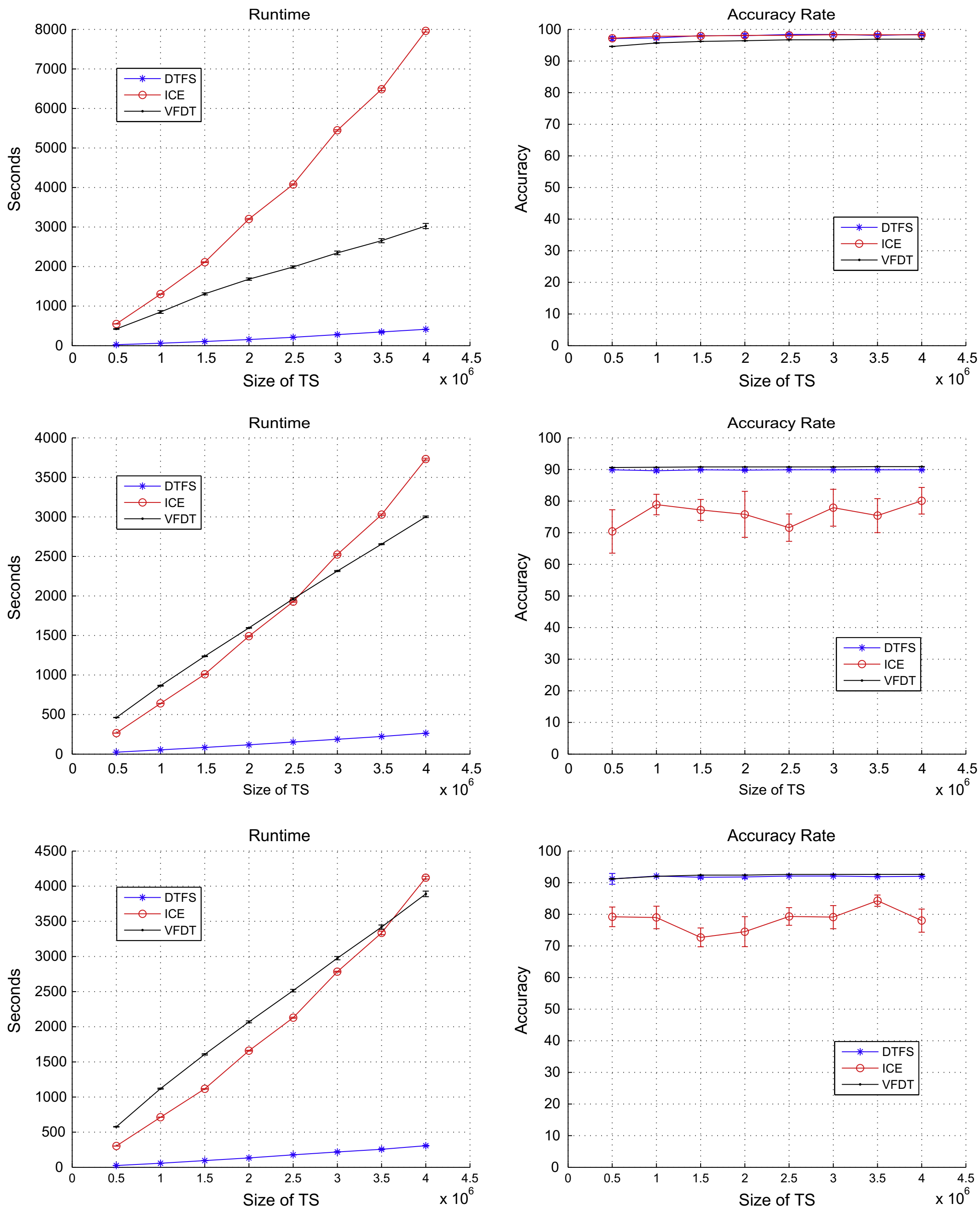

Fig. 10. Runtime and accuracy rate for DTFS and VFDT with 2-class, 3-class and 5-class for the 40-attribute synthetic datasets, respectively.

validation the paired *t* test (Demsar, 2006) with a confidence level of 99.9%.

With respect to the accuracy rate, there is no statistically significant difference among DTFS, ICE, VFDT and BOAI for most of the datasets. But, with respect to the runtime, DTFS is statistically significant faster than ICE, VFDT and BOAI in all datasets.

5. Conclusions and future work

In this work, we have introduced a decision tree induction algorithm, called DTFS, which uses a fast splitting attribute selection for expanding nodes. Our algorithm does not require to store the whole training set in memory and processes all the instances in the training set. The key insight is to process one by one the instances for updating the DT with each one (processing the data in an incremental way) and to use a small number of instances for expanding a leaf (discarding them after the node expansion). Besides, using only a predefined number of instances for expanding a leaf allows a fast splitting attribute selection, therefore DTFS is able to process large datasets. This fast selection of the splitting attributes makes DTFS stable in runtime when the number of attributes increases. This fact was shown in the experimental results.

DTFS is an algorithm that has a user-defined parameter (s), however, our experiments showed that the behavior of our algorithm with respect to this parameter is very stable, since the processing time just varies a little and the accuracy rate is very similar for the different values of s.

Since pruning is an expensive process, especially for large datasets, DTFS does not include a pruning step. However, our algorithm does not always expand a node. The option of updating a node instead of doing an expansion (case two in Section 3.2), allows DTFS to avoid expanding homogeneous nodes which do not provide useful information for separating classes, and it also avoids to build big DTs.

In the experiments over both real and synthetic datasets our algorithm was statistically significant faster than three of the most recent algorithms for building DTs, ICE, VFDT and BOAI algorithms, while DTFS maintains competitive accuracy.

As future work, we will try to speed up DTFS even more using parallel computing or hardware implementation techniques.

Acknowledgement

Funding for the SDSS and SDSS-II has been provided by the Alfred P. Sloan Foundation, the Participating Institutions, the National Science Foundation, the US Department of Energy, the National Aeronautics and Space Administration, the Japanese Monbukagakusho, the Max Planck Society, and the Higher Education Funding Council for England. The SDSS Web Site is http://www.sdss.org/.

The SDSS is managed by the Astrophysical Research Consortium for the Participating Institutions. The Participating Institutions are the American Museum of Natural History, Astrophysical Institute Potsdam, University of Basel, University of Cambridge, Case Western Reserve University, University of Chicago, Drexel University, Fermilab, the Institute for Advanced Study, the Japan Participation Group, Johns Hopkins University, the Joint Institute for Nuclear Astrophysics, the Kavli Institute for Particle Astrophysics and Cosmology, the Korean Scientist Group, the Chinese Academy of Sciences (LAMOST), Los Alamos National Laboratory, the Max-Planck-Institute for Astronomy (MPIA), the Max-Planck-Institute for Astrophysics (MPA), New Mexico State University, Ohio State University, University of Pittsburgh, University of Portsmouth, Princeton University, the United States Naval Observatory, and the University of Washington.

References

Alsabti, K., Ranka, S., & Singh, V. (1998). CLOUDS: A decision tree classifier for large datasets. In *Proceedings of conference knowledge discovery and data mining (KDD'98)* (pp. 2–8).

Berzal, F., Cubero, J. C., Marn, N., & Snchez, D. (2004). Building multi-way decision trees with numerical attributes. *Information Sciences, 165*(1–2), 73–90.

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

Demsar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research, 7*, 1–30.

Domingos, P., & Hulten, G. (2000). Mining high-speed data streams. In *Proceedings of of sixth international conference on knowledge discovery and data mining* (pp. 71–80). ACM Press.

Freund, Y., & Mason, L. (1999). The alternating decision tree learning algorithm. In *16th international conference on machine learning* (pp. 124–133).

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

Gehrke, J., Ramakrishnan, R., Ganti, V. (1998). Rainforest-A framework for fast decision tree classification of large datasets. In *Proceedings of VLDB conference* (pp. 416–427).

Gehrke, J., Ganti, V., Ramakrishnan, R., & Loh, W. (1999). BOAT – Optimistic decision tree construction. *ACM SIGMOD Record, 28*(2), 169–180.

Gehrke, J., Ramakrishnan, R., & Ganti, V. (2000). Rainforest – A framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery, 4*, 127–162.

Jin R., & Agrawal G. (2003). Efficient decision tree construction on streaming data. In *Proceedings of ninth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 571–576).

Li, Z., Wang, T., Wang, R., Yan, Y., & Chen, H. (2007). A new fuzzy decision tree classification method for mining high-speed data streams based on binary search trees. In *Proceedings of of FAW conference* (pp. 216–227).

Mehta, M., Agrawal, R., Rissanen, J. (1996). SLIQ: A fast scalable classifier for data mining. In *Proceedings of fifth international conference extending database technology (EDBT), Avignon, France* (pp. 18–32).

Nguyen U., & Tae-Choong, C. (2007). An efficient decision tree construction for large datasets. In *Proceedings of fourth international conference on innovations in information technology* (pp. 21–25).

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

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. San Francisco, CA: Morgan Kaufmann.

SDSS-Adelman-McCarthy, J., Agueros, M. A., & Allam, S. S. (2008). Data release 6. *ApJS, 175*, 297.

Shafer, J.C., Agrawal, R., & Mehta, M. (1996). SPRINT: A scalable parallel classifier for data mining. In *Proceedings of 22nd international conference very large databases* (pp. 544–555).

Tan, P. N., Steinbach, M., & Kumar, V. (2006). *Introduction to data mining*. Addison Wesley.

UCI Machine Learning Repository, University of California (2008). <http://archive.ics.uci.edu/ml>.

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine Learning, 4*, 161–186.

Utgoff P. E., & Brodley C. E. (1990). An incremental method for finding multivariate splits for decision trees. In *Proceedings of seventh international conference on machine learning* (pp. 58–65).

Utgoff P. E. (1994). An improved algorithm for incremental induction of decision trees. In *Proceedings of 11th international conference on machine learning* (pp. 318–325).

Yang, B., Wang, T., Yang, D., & Chang, L. (2008). BOAI: Fast alternating decision tree induction based on bottom-up evaluation. In *Proceedings of PAKDD conference* (pp. 405–416).

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