Electronic Letters on Computer Vision and Image Analysis 19(3):38-54, 2020

A Novel Method to Improve the Efficiency of Classification Phase of a Decision Tree

Janapati Naga Muneiah[1,*] Ch D V Subba Rao[2]

[1]*Department of Computer Science and Engineering, Jawaharlal Nehru Technological University, Kakinada-533005, Andhra Pradesh, India.*

[2]*Department of Computer Science and Engineering, Sri Venkateswara University College of Engineering, Tirupati- 517502, Andhra Pradesh, India.*

Received 25th of July 2020; accepted 25th of September 2020

Abstract

So far, most of the research on classification algorithms in machine learning has been focused only on improving the training speed and further improving the technical performance evaluation measures of the constructed models. There is no focus on improving the runtime efficiency of the classification phase which is much required in some critical applications. In this paper, we are considering the computation complexity of a decision tree's classification phase as the major criterion. A novel approach has been proposed to predict the class label of an unseen instance using the decision tree in less time than the regular tree traversal method. In the proposed method, the constructed decision tree is represented in the form of arrays. Then, the process of finding the class label is carried out by performing the bitwise operations between the elements of the arrays and test instance. Empirical results on various UCI data sets proved that the proposed method outperforms the standard method and five other benchmark classifiers and its classification is at least four times faster than the regular method.

Key Words: Data mining, Classification, Decision trees, Classification phase, Computational efficiency.

1 Introduction

Classification is the predominant concept in the fields of data mining and machine learning which is used to predict the class label of an unseen instance. Many sectors like medicine, telecom, banking, speech recognition, handwritten character recognition, fraud detection, and biology, etc. are using the concept of the classification to predict the class label of the unseen record [1, 2]. Though several classifiers like bayesian classifiers, neural networks, *k*-nearest neighbour, and support vector machines, and so on are available, decision trees (DT) are popular and received huge significance due to their various merits like less training time, good accuracy, dealing well with high dimensional data, and working for both numeric and categorical data. When the user needs a simple and interpretable classifier, DT is the most preferable one. For the automatic extraction of actionable knowledge, many researchers have considered the DT as the model [3, 4, 5]. In general, classification through DTs comprises three phases namely, training phase, testing phase, and classification phase. During the testing and classification phases, the class label has to be determined for an instance using the DT.

Correspondence to: <nagamuni513@gmail.com>
Recommended for acceptance by Angel D. Sappa
https://doi.org/10.5565/rev/elcvia.1267
ELCVIA ISSN: 1577-5097
Published by Computer Vision Center / Universitat Autonoma de Barcelona, Barcelona, Spain

Earlier, many researchers have focused on comparing various technical performance evaluation measures of a variety of classifiers [6, 7, 8, 9]. Few researchers also put their efforts to compare the performance of various classifiers in terms of their computational times. Reif et al. proposed an approach to predict the runtime of the training phase of different classifiers [10]. Doan et al. presented an approach to estimate the total runtime, i.e. training time and classification time, on various classification algorithms on an arbitrary data set by incorporating the idea of meta-learning [11].

Most of the past research has concentrated only on improving the technical evaluation measures like accuracy, AUC, sensitivity, specificity, precision, and F-score of the classifier [12, 13, 14]. The researchers of data mining have not focused on expediting the classification phase with a notion that *this phase takes less time*. In fact, if the classification phase of a DT is accelerated, then it helps speed up the training phase too. Either for the test records or any unseen records though the time taken for determining the class label by traversing the DT from root to an appropriate leaf node is low, still, this process can be improvised by adopting the efficient data structures.

Reducing the classification complexity need not be neglected since there are applications where the fast classification of an instance is much required. In many time-critical cases, actions have to be taken very quickly based on the decision otherwise the cost can be very high or consequences can be unbearable. Such criticality could be achieved through fast classification. Predicting the complications in critical care quickly, where electronic surveillance systems can detect the symptoms, needs an automated rapid action based on the decision [15]. Text to speech conversion in the case of CRM automation, hard real-time applications, aircraft systems, fraud detection during online transactions, dynamic systems, and other applications require fast classification to take timely necessary remedial actions. In big data scenarios, in the Map phase, if it is required to predict the class label for a huge number of instances, an efficient method is much useful to save the processing time [16].

Ahmad Ashari et al. have studied the classification speed of three standard classifiers [17] and opined that the decision tree's performance is relatively high. Weinberg et al. [16] have proposed a method to improve classification speed when the ensemble of decision trees is constructed on big data. Their method finds the best representative tree from the ensemble on which classification is performed. Hurtik et al. have introduced a method viz. PCA+FT which can enhance the training and classification speed [18]. However, this is a lossy dimensionality reduction based method where a classifier has to be fit on a PCA transformed data.

To predict the class label of a test record, the existing conventional decision tree based method performs comparison operations at each node for finding the attribute representing the node. Thereafter, to traverse down to reach the appropriate leaf, it performs comparison operations to find the value of the required attribute within the test record. This method employs linked lists in the process. Comparison operations are slower and the linked lists are not too fast for accessing the elements.

The research in this paper focuses on improvising the computational efficiency of the classification phase of a DT. A new approach that is based on bitwise operations has been introduced. In this approach, the constructed DT is represented as arrays and then, the bitwise AND operations are performed on the elements of the arrays for finding the class label. The step count required to find the class label of an unseen record using the proposed method is relatively less than the step count of the regular approach. Moreover, as the bitwise operations are faster [19, 20, 21], and the arrays are also faster for accessing the elements, the proposed method exhibits better computational performance than the classical method. Once the DT is represented as the arrays, it is discarded and further at no stage it is used. The applications which require fast classification can be benefitted by adopting the proposed method.

Rest of the paper is organised as: Section 2 reviews the conventional classification method of a DT. Section 3 presents the proposed novel method with an illustration. Section 4 provides the performance analysis of the proposed method and section 5 gives the concluding remarks.

2 Review of classification phase of a decision tree

All the existing DT construction algorithms follow the regular tree traversal method to predict the class label of an unseen instance [22]. An instance is classified by submitting it to a series of tests that determine its class label. To traverse down, at each of the non-terminal nodes, it is required to determine the attribute, $attr_i$, representing the node. Thereafter, in the test instance, the value of the attribute $attr_i$ has to be found. Then, based on its $attr_i$'s value the test instance traverses down the tree. This process is repeated until one of the leaf nodes is reached. The pseudocode for the regular method (hereafter we refer it as *R_Classify*), is presented in Algorithm 1.

In this approach, the DT is represented in the form of linked lists where each node contains the pointers to its children. At each node to find the attribute, *atr* number of comparisons are required to be performed where *atr* is the number of attributes in the data set. If the average number of outcomes of the attributes of the data set is *o*, then *o* number of comparisons is required to find that attribute's value in the test instance. Up to this stage, (*atr* + *o*) number of comparison operations are needed to be performed. If the average length of each path from the root to a leaf of the DT is *h*, then a total of (*h* * (*atr* + *o*)) comparisons are needed. Consequently, time complexity of the traditional method is O(*h* * (*atr* + *o*)). To our knowledge, no research has introduced an alternative efficient method for finding the class label of an instance on the DT classifier.

Algorithm 1: Pseudo code for predicting class label of an instance using the conventional ***R_Classify*** method

```
Inputs :  list  -   Reference of the root node of the decision tree
          X[ ]  -   Test instance
Output :  Class label for the test instance

Step 1 :     while list.next is not NULL   do
Step 2 :        for each $i \in atr$   do
Step 3 :           if list.atrname = name[$i$]  then
Step 4 :              $ind \leftarrow i$ ;
Step 5 :              Jump to step 8;
Step 6 :           end if
Step 7 :        end for
Step 8 :        for each $j \in$ attr[$ind$].fan_out do
Step 9 :           if X[$ind$] = attr[$ind$].val[$j$]  then
Step 10 :             Jump to step 13;
Step 11 :          end if
Step 12 :       end for
Step 13 :       list $\leftarrow$ list.next + $j$;
Step 14 :    end while
Step 15 :    return list.atrname;
```

3 Proposed scheme

To achieve better computational efficiency than the conventional approach a new method namely, *E_Classify* has been introduced. The new method employs the arrays and bitwise AND operations to achieve the objective of finding the class label of an instance. The working of the proposed method is briefly described in the following three steps.

(i) Form the 2-dimensional attributes' bit-patterns array, AB[ ][ ], with the outcomes of all the attributes in the DT.

(ii) Represent the outcomes of the non-terminal nodes of the DT in the form of arrays, LT[ ][ ], AX[ ][ ]. Maintain the number of outcomes of each of the attributes of the DT in an array, fan_out[ ]. The entries of the LT[ ][ ] are used to find the subsequent attribute/node, i.e. *attr*, to be processed along the path to a leaf node for an input instance. Then, for that *attr*, to find the corresponding row number in LT[ ][ ], the elements of the matrix AX[ ][ ] are used. Every entry in AX[ ][ ] denotes one of the row numbers pertaining to LT[ ][ ]. These arrays are of type integers which contains either the attribute *Id*s or the leaf node *Id*s.

(iii) By accessing the elements of the arrays LT[ ][ ], AX[ ][ ], and fan_out[ ] perform the bitwise AND operations between the contents of AB[ ][ ] and the input instance X[ ] to find the class label.

To demonstrate the working of the proposed scheme, a sample 2-class data set as given in Appendix has been considered [35]. This data set is given as input to the C4.5 algorithm [22] and the DT as given in Fig. 1 has been obtained where each of the leaf nodes is also associated with an *Id* viz., L1, L2, L3, L4, and L5 in addition to a class label.

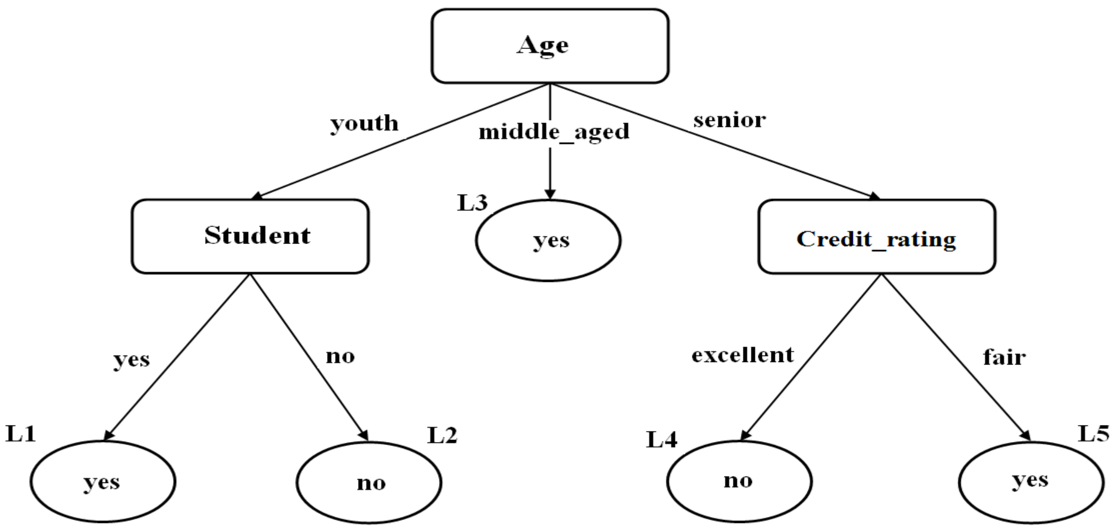

Fig. 1 Decision tree constructed using sample 2-class data set in Appendix.

An array, fan_out[ ], containing the fan-out of each of the attributes in the DT is maintained as given in Table 1. Each non-terminal attribute is also assigned a numeric identifier for future references. A 2-dimensional array, AB[ ][ ], which contains bit-patterns of the outcomes of the attributes is organized where each of its row contains o_{atr_i} patterns of the attribute atr_i. If the number of outcomes of an attribute is o then the length of each of the bit-patterns of that attribute is o-bits where one bit of the pattern is set to 1 and the remaining bits are set to 0s. This array is used to find the test record attributes' values that are further needed to find the next node to be visited in the next level of the tree. The pseudocode for the formation of the AB[ ][ ] is given below.

```
for each $i \in atr$ do
    for each $j \in o_{atr\_i}$ do
        Among $o$ bits/outcomes of attribute-$i$, set the $j$th bit to 1 and remaining bits to 0s
    end for
end for
```

The matrix, AB[ ][ ], for the DT given in Fig. 1 is shown in Table 2. For the test sample X, in the process of reaching the appropriate leaf node from the root, it is required to find the attribute representing each of the non-terminal nodes along the path. Thereafter, that attribute's value of input instance X has to be found to determine the appropriate path that has to be chosen to traverse down to the next level of the DT. To help attaining these tasks, two arrays i.e. LT[ ][ ], a lookup table, and AX[ ][ ], an auxiliary array, are maintained.

Table 1: fan_out [ ] array containing fan-out of all the attributes in the decision tree given in Fig. 1.

fan_out [ ]	fan_out [1]	fan_out [2]	fan_out [3]
Attribute Name	*Age*	*Student*	*Credit_rating*
Attribute *Id*	*1*	*2*	*3*
Attribute's fan-out	3	2	2

Table 2: Matrix AB[ ][ ] containing the bit-patterns of the attributes in the DT in Fig. 1.

Attribute	Row#	Column#		
		1	2	3
Age	1	AB[1][1] **100** *Youth*	AB[1][2] **010** *Middle_aged*	AB[1][3] **001** *Senior*
Student	2	AB[2][1] **10** *Yes*	AB[2][2] **01** *No*	**-**
Credit_rating	3	AB[3][1] **10** *Fair*	AB[3][2] **01** *Excellent*	**-**

If the number of non-terminal nodes in the DT is *n*, then the number of rows in the LT[ ][ ] is also *n*. Consequently, one row in the LT[ ][ ] corresponds to one of the non-terminal nodes represented by an attribute of the DT. However, the first row of the LT[ ][ ] corresponds to the attribute representing the root node. A variable, *row*, maintains the required row number in LT[ ][ ]. To track the attribute/node of the DT that has to be processed, another variable *attr* is maintained. This variable holds one of the attribute *Id*s as given in Table 1. As it is obvious that the traversal has to start from the root node, *attr* is initialized with the root node attribute's Id. Hence, for the DT in Fig. 1, *attr* = 1 which is the attribute *Id* of *Age* (Table 1). At the same time, *row* = 1. This implies that the node/attribute to be processed in the DT is *Age*, and the corresponding row for *Age* (at root node) in LT[ ][ ] is row-1. Each entry of the LT[ ][ ] is either an attribute's *Id* or a leaf node's *Id*. If a row-*r* of LT[ ][ ] corresponds to an attribute *attr* whose fan-out is *o* then, the number of non-null elements in the row-*r* of LT[ ][ ] is *o*. Thus, the number of outcomes of attribute *attr,* and the entries in the corresponding row of LT[ ][ ] are related. The *i*th outcome of *attr,* where *i* = 1, 2, ..., *o*, corresponds to LT[*row*][*i*].

For the test record X[ ], to find the next node/attribute along the appropriate path of DT, a bitwise AND operation is performed in between X[*attr*]'s bit-pattern and the bit-patterns of attribute *attr* maintained in the row-*attr* of AB[ ][ ]. Whenever, one of the bit-patterns in the row-*attr* of AB[ ][ ] match to the value of the test instance X[*attr*], the corresponding bitwise ANDing results in *True*. Thus, the value of the attribute *attr* in X[ ] is found. If X[*attr*] value is matched with AB[*attr*][*i*] then the *Id* of the next node/attribute along the path to be visited in the DT is found at LT[*row*][*i*]. For the next attribute along the path of the DT, the corresponding row number within LT[ ][ ] is accessed from the auxiliary array AX[ ][ ] at AX[*row*][*i*] to continue the process. This procedure is repeated until a leaf node is reached. For the DT in Fig. 1, LT[ ][ ] and AX[ ][ ] are formed and given in Table 3 and Table 4 respectively.

If the value of *Age* of an input instance X is *youth*, then the non-terminal node represented by the attribute *Student*, whose attribute *Id* is 2, has to be visited. Hence, entry at LT[1][1] is filled with 2. If the *Age* attribute's value of X is *middle_aged*, then leaf node L3 has to be reached. Consequently, LT[1][2] = L3. If the *Age* value is *senio*r, the non-terminal node represented by *Credit_rating*, whose attribute *Id* is 3. Thus, LT[1][3] = 3 has to be reached. For example, when the *Age* = *youth*, the non-terminal node represented by *Student* has to be visited. During the time of processing of this node, the required row (number) in LT[ ][ ] for *Student*, which is the leftmost child of *Age*, is 2. Hence, AX[1][1] is filled with 2. When a leaf node is reached, no further processing is required. Hence, AX[1][2] is filled with *0*. The rows of LT[ ][ ] and AX[ ][ ] are formed by following the depth-first search order. According to the procedure discussed, the *E_Classify* algorithm has been designed and the pseudocode is presented in Algorithm 2.

Table 3: Lookup table, LT[ ][ ], with Attribute/Leaf *Id*s.

Row #	Column#		
	1	2	3
1	LT[1][1] **2**	LT[1][2] **L3**	LT[1][3] **3**
2	LT[2][1] **L1**	LT[2][2] **L2**	LT[2][3] **-**
3	LT[3][1] **L5**	LT[3][2] **L4**	LT[3][3] **-**

Table 4: Auxiliary array AX[ ][ ] for identifying corresponding row in LT[ ][ ].

Row #	Column#		
	1	2	3
1	AX[1][1] **2**	AX[1][2] **0**	AX[1][3] **3**
2	AX[2][1] **0**	AX[2][2] **0**	AX[2][3] **-**
3	AX[3][1] **0**	AX[3][2] **0**	AX[3][3] **-**

Algorithm 2: Pseudocode for predicting class label of an instance using ***E_Classify***

```
Inputs :  - X[ ] i.e. Test instance
          - Root_ Id
          - AB[ ][ ], LT[ ][ ], AX[ ][ ], fan_out[ ]
Output :  Class label for the test instance

Step 1  :  attr ← Root_ Id;
Step 2  :  row ← 1;
Step 3  :  while attr is not Leaf do
Step 4  :      for each i ∈ fan_out[attr] do
Step 5  :          if AB[attr][i] & X[attr] then
Step 6  :              attr ← LT[row][i] ;
Step 7  :              row ← AX[row][i] ;
Step 8  :              Jump to Step 10;
Step 9  :          end if
Step 10 :      end for
Step 11 :  end while
Step 12 :  return Class Label represented by Leaf;
```

Consider a customer's record X = (*Age* = *senior*, *Student* = *no, Credit_rating* = *fair*) whose class label has to be predicted using the DT in Fig. 1. Bit-patterns of the attributes of record-X are given in Table 5. The process begins with *attr* = 1, and *row* = 1. Then, initially, the bitwise ANDings are performed in between the elements of AB[1][ ] since the root node represents *Age* and X[1] as shown in Table 6.

Table 5: Bit-patterns of the test record X[ ].

Attribute Name	*Age*	*Student*	*Credit_ rating*
Attribute ***Id***	**1**	**2**	**3**
Bit-pattern	X[1] **001**	X[2] **01**	X[3] **10**

Table 6: Bitwise operations between X[*Age*] and outcomes of attribute *Age.*

i	*i* = 1	*i* = 2	*i* = 3
X[*attr*] = X[1]	X[1] **001**	X[1] **001**	X[1] **001**
AB[1][*i*]	AB[1][1] **100**	AB[1][2] **010**	AB[1][3] **001**
X[1] **& A**B[1][*i*]	*False*	*False*	***True***

When *i* = 3, the ANDing resulted in *True*. Consequently, *Age* value of X[ ] has been identified as *senior*. After finding the *Age* value in X, the attribute *Id* of the next attribute to be visited in DT is, *attr* = LT[*row*][*i*] = LT[1][3] = 3 i.e. *Credit_rating* (from Table 1), and the corresponding row number in LT[ ][ ] for *Credit_rating* is, *row* = AX[*row*][*i*] = AX[1][3] = 3. Then, bitwise ANDings are performed between X[*Credit_rating*] and AB[3][ ] as shown in Table 7 to find the next node in the DT. From the results in Table 7 it can be observed that when *i* = 1 the ANDing resulted in *True* and the entry at LT[3][1] is accessed and the class label *yes*, which is represented by L5, is returned.

Table 7: Bitwise AND operations between X[*Credit_rating*] and outcomes of attribute *Credit_rating.*

i	*i* = 1	*i* = 2
X[*attr*] = X[3]	X[3] **10**	X[3] **10**
AB[3][*i*]	AB[3][1] **10**	AB[3][2] **01**
X[3] **&** AB[3][*i*]	***True***	

4 Results and Discussion

The proposed method for predicting the class label of an instance purely depends on arrays and bitwise operations. Arrays are faster for random accessing of the required element with a constant time complexity O(1). Bitwise operations are also quicker when compared to equality conditional operations since they take only one clock cycle [19, 20, 21]. In the conventional tree traversal method, at each node, it is required to find the attribute and then the test instance's value of that attribute. For achieving both tasks, the traditional method performs comparison operations. Moreover, in the regular method, the tree is represented in the form of linked lists. For determining the attribute's value of the test instance, the proposed method employs bitwise AND operations. Subsequently, to determine the next attribute to be visited along the path, it employs the arrays. Comparison operations and linked lists are computationally costlier than the bitwise operations and arrays. It can be observed from the pseudocodes of the two methods (Algorithm 1 and Algorithm 2), that the proposed method is more efficient than the regular method in terms of step count. In the proposed method, instead of two *for*-loops, one *for*-loop is able to achieve the objective. The overhead of finding the attribute by applying comparison operations at a node is avoided in the proposed method.

For complexity analysis, we consider that the average length of the paths from the root to leaves of the DT is *h* and the average fan-out of each of the attributes of DT is *o*. Then, *E_Classify* requires ($h * o$) bitwise AND operations to predict the class label of a record. As a result, the time complexity of *E_Classify* is O($h * o$). On the other hand, the runtime of the regular method is O($h * (atr + o)$) where it has to perform ($h * (atr + o)$) number of comparison operations.

4.1 Experimental set up

To verify the computational performance, the proposed method and the comparing methods are implemented in Java programming language and the experiments are conducted on a dual core Intel i3, 3.80 GHz processor with 8GB RAM running on Windows7 Operating System. All the numeric attributes are discretized during the preprocessing stage. C4.5 algorithm [22] has been used for decision tree construction as it is a popular algorithm and also ranked as #1 algorithm in the field of data mining [23]. Experiments are conducted on 10 UCI ML data sets [24]. The constructed models are validated using 10-fold cross-validation.

4.2 Performance analysis with *German* data set

Initially, for the detailed analysis of the proposed method, *German* data set from the UCI repository has been used. This data set contains 20 input attributes and a binary class label, i.e. *good* or *bad*. We renamed the input attributes as A, B, ..., and, T, class labels as C_1 and C_2 and outcomes of the attributes as a, b, c, etc. for simplicity. By using 666 randomly chosen records from the *German* data set, a decision tree as shown in Fig. 2 has been constructed. Size of the tree is 40 with 29 leaf nodes and 11 non-terminal nodes. Each leaf is also associated with a leaf Id from L1 through L29. A data set with a moderate size and dimensionality can help for a detailed explanation of the methodology of our proposed approach. Hence, *German* data set has been considered for detailed experimental analysis in this section.

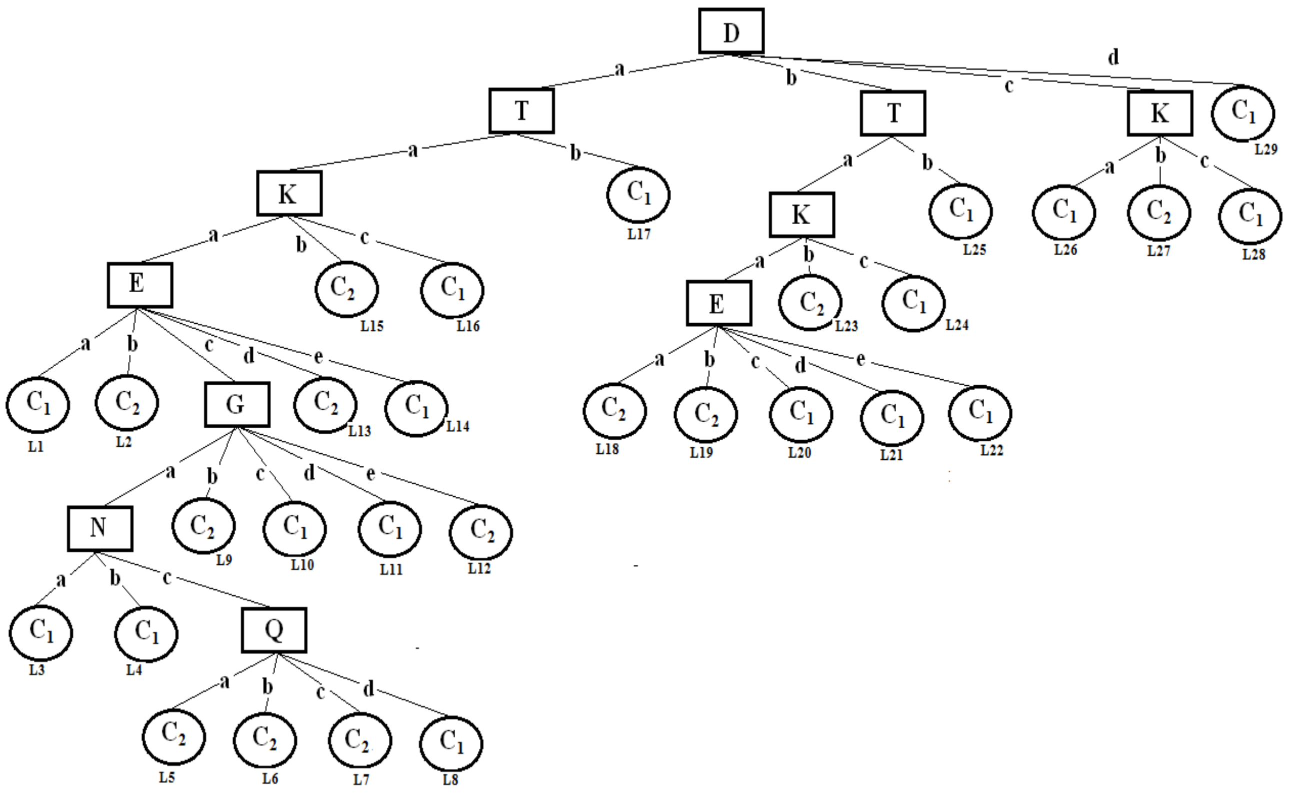

Fig. 2 Decision tree constructed using *German* data set

The data structures required during the process of predicting the class label of a test instance are furnished below. Table 8 contains the attributes' *Id*s, fan-out values, and the outcomes of the 7 distinct attributes present in the decision tree in Fig. 2.

Table 8: Fan-out and *Id*'s of each attribute in the decision tree in Fig. 2.

fan_out [ ]	fan_out [1]	fan_out [2]	fan_out [3]	fan_out [4]	fan_out [5]	fan_out [6]	fan_out [7]
Attribute Name	*D*	*E*	*G*	*K*	*N*	*Q*	*T*
Attribute *Id*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Attribute's fan-out	4	5	5	3	3	4	2
Outcomes of attribute	*a, b, c, d*	*a, b, c, d, e*	*a, b, c, d, e*	*a, b, c*	*a, b, c*	*a, b, c, d*	*a, b*

The 2-D array AB[ ][ ] containing the bit-patterns of outcomes of all the 7 attributes in the decision tree is given below.

AB[7][5] = {{1000, 0100, 0010, 0001, -}, {10000, 01000, 00100, 00010, 00001}, {10000, 01000, 00100, 00010, 00001}, {100, 010, 001, -,-}, {100, 010, 001,-,-}, {1000, 0100, 0010, 0001, -}, {10, 01, -, -, -}}.

The elements of the lookup table LT[ ][ ] which are used to follow the outcomes of a node/attribute in the tree are furnished below. As the tree contains 11 non-terminal nodes, LT[ ][ ] also contains 11 corresponding rows.

LT[11][5] = {{7, 7, 4, L29, -}, {4, L17, -, -, -}, {2, L15, L16, -, -}, {L1, L2, 3, L13, L14}, {5, L9, L10, L11, L12}, {L3, L4, 6, -, -}, {L5, L6, L7, L8, -}, {4, L25, -, -, -}, {2, L23, L24, -, -}, {L18, L19, L20, L21, L22}, {L26, L27, L28, -, -}}.

The elements of the auxiliary array, AX[ ][ ], to find the corresponding row in the LT[ ][ ] are as shown below.

AX[11][5] = {{1, 7, 10, 0, -}, {2, 0, -, -, -}, {3, 0, 0, -, -}, {0, 0, 4, 0, 0}, {5, 0, 0, 0, 0}, {0, 0, 6, -, -}, {0, 0, 0, 0, -}, {8, 0, -, -, -}, {9, 0, 0, -, -}, {0, 0, 0, 0, 0}, {0, 0, 0, -, -}}.

At one time one record has been given as input to the two approaches and the classification times are noted. To obtain notable runtimes, the process of predicting class label for one sample has been repeated for 10000 times. For each test instance, while recording the runtime with *R_Classify* method (T1), the total number of attribute name comparisons (O1), test instance attributes' values comparisons (O2), assignment operations (O3), and the total step count (O4 = O1 + O2 + O3) required for predicting class label are also recorded. For the same instance when *E_Classify* method is used for predicting the class label, the total number of bitwise AND operations (O5), assignment operations (O6), and the total step count (O7 = O5 + O6) have been noted along with the execution time (T2). P1, P2, and P3 values represent the percentage of bitwise operations among all the operations done by *E_Classify*, percentage of step count of *E_Classify* in comparison to *R_Classify*, and the percentage of runtime of *E_Classify* in comparison to *R_Classify* respectively. The results on 10 test instances are furnished in Table 9. For each test record in Table 9, the values of the input attributes A, B, ..., and T are given without separating them by a comma. The runtimes and the total step counts of the two methods on each record are plotted as bar charts and given in Fig. 3 and Fig. 4 respectively.

Table 9: Classification times comparison of ***R_Classify*** and ***E_Classify*** on *German* data set.

T_ Id	Test Record	*R_Classify*					*E_Classify*					**(O7/ O4) %**	(**T2/ T1**) **%**
	(Input attributes order) ABCDEFGHIJKLMNOPQRST	Step count				Runtime (*ms*)	Step count			(O5/ O7) %	Runtime (*ms*)		
		O1	O2	O3	O4	(**T1**)	O5	O6	O7	**(P1)**	(**T2**)	**(P2)**	**(P3)**
1	abaacbabbbabbcbbdbba	78	14	7	99	12.76	14	14	28	50.0	2.53	28.3	19.8
2	bbabbaaaaaaaaaaaaaaa	40	6	4	50	5.96	6	8	14	42.8	1.23	28.0	20.6
3	babcaaaaaabaaaaaaaaa	15	5	2	22	2.98	5	4	9	55.5	1.08	40.9	36.2
4	abbacabaaaaaaaaaaaaa	47	8	5	60	7.67	8	10	18	44.4	2.17	30.0	28.3
5	bbbaaaaaaaaaaaaaaaaa	40	4	4	**48**	**5.62**	4	8	**12**	**33.3**	**1.08**	**25.0**	**19.2**
6	aaabaaaaaacaaaaaaaaa	35	6	3	44	4.99	6	6	12	50.0	1.14	27.3	22.8
7	babaaaaaaaaaaaaaaaab	24	3	2	**29**	**3.71**	3	4	**7**	42.8	**0.66**	**24.1**	**17.8**
8	abaacaaaaaaaaaaaaaaa	61	8	6	75	9.21	8	12	20	40.0	2.32	26.7	25.2
9	abbaeaaaaaaaaaaaaaaa	40	8	4	52	6.86	8	8	16	50.0	1.77	30.8	25.8
10	bbadaaaaaaaaaaaaaaaa	4	4	1	**9**	**1.04**	4	2	**6**	**66.7**	**0.55**	**66.7**	**52.8**

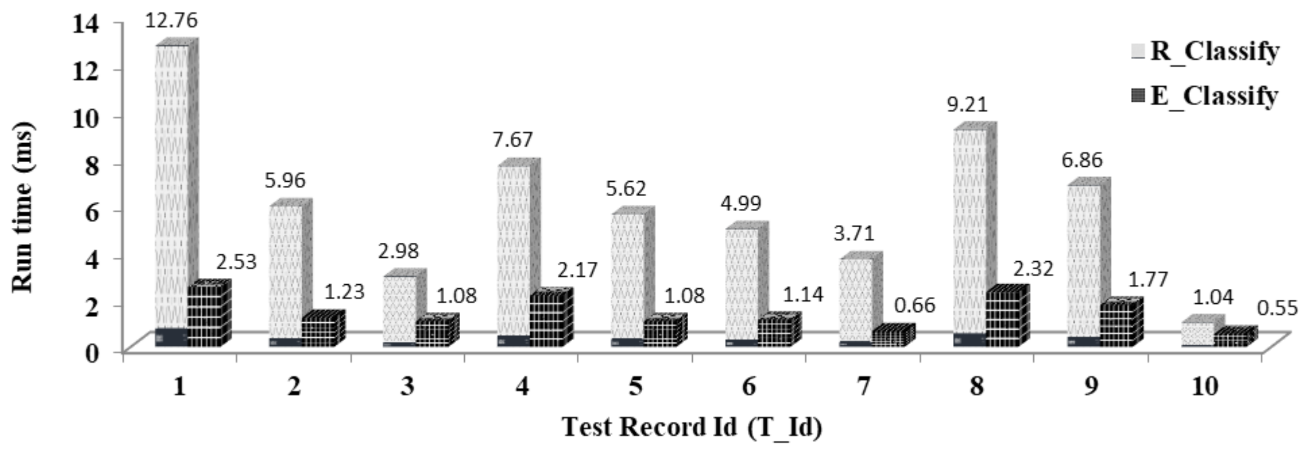

Fig. 3 Runtime comparison of *R_Classify* and *E_Classify* on *German* data set.

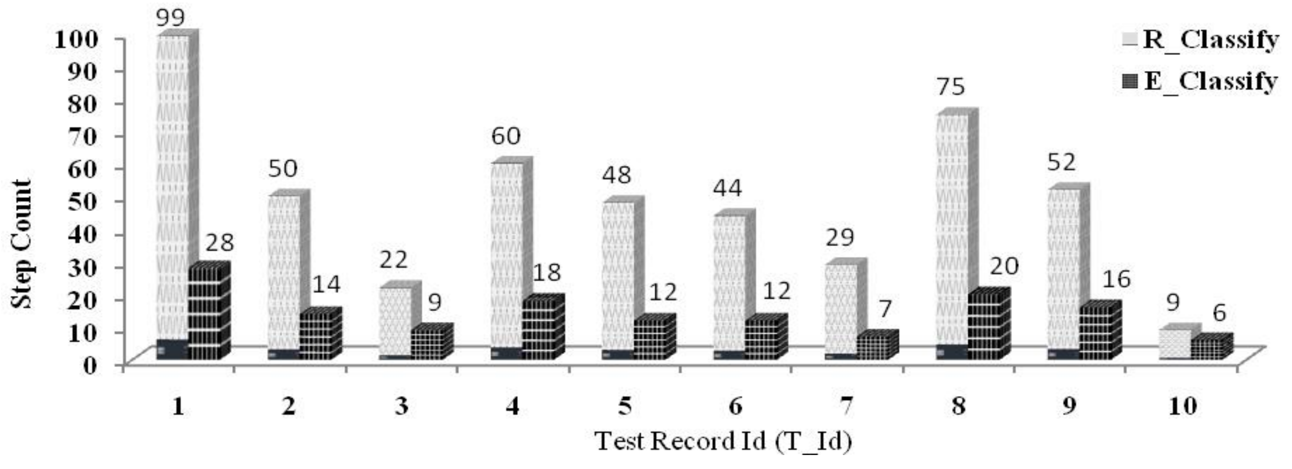

Fig. 4 Step count comparison of *R_Classify* and *E_Classify* on *German* data set.

The total runtimes of *R_Classify* and *E_Classify* are 60.8 *ms* and 14.52 *ms* respectively where the latter method is four times faster than the former one. On the other hand, the average runtime of *E_*Classify is nearly one-fourth of the former method. From the results shown in Table 9, it can be observed that the runtimes are directly proportionate to the step count. The step count of *E_Classify* is less than that of the *R_Classify* method for all instances and its total step count is 29.64% of the regular method. The results depict that the step count and runtimes of *E_*Classify are relatively less than that of the *R_Classify* for all cases. The bitwise AND operations of *E_Classify* (O5) and the comparison operations for finding the attribute's value at a node using *R_Classify* are equal since the bitwise AND operations are used for finding the test instance attributes' values.

The best performance of the proposed method has been observed on the 7th test record. For this record, the classification times of *R_Classify* and *E_Classify* are 3.71 *ms* and 0.66 *ms* respectively, where the latter method's runtime is only 17.8% of the former one. The reason for this significant difference is, to find the class label for this instance, the step count required by *E_Classify* is 24.1% of *R_Classify* which is minimum among all the cases shown in Table 9.

The 10th record takes the rightmost path from the root and reaches the leaf node L29. For this record, *E_Classify* performs 4 bitwise AND's since the attribute *D*'s value for this instance is its fourth outcome i.e. *d*. The difference in the runtimes for the 10th test sample is least since the step count of *E_Classify is 66.67%* of *R_Classify* which is highest among all the 10 cases. In proportion to the step count, the increase in the runtime of *E_Classify* is not too high since the contribution of bitwise operations of *E_Classify* is high i.e. 66.7%. Thus, in some of the cases, though there is no much difference in the step count of two methods (since that path contains very less number of non-terminal attributes), the *E_Classify* still exhibits better performance than the *R_Classify* as it employs the bitwise operations and accesses the elements from the arrays. *E_Classify* is not explicitly performing any operations to identify the attribute representing a non-terminal node.

4.3 Experiments on UCI data

For experimental analysis, we have chosen 10 data sets from UCI ML repository [24], the most extensively used database in the classification literature and the classification times of *R_Classify* and *E_Classify* methods are compared. Reasons for choosing these data sets are: Classifier behaviour also depends on the data set properties like dimensionality, size, and the number of classes. Each of these data sets has a sufficient number of records and is composed of different dimensionalities and sizes that help better observation and differentiation of the performances of the comparing methods. If the decision tree is constructed using the data sets with different dimensionalities and size, then the trees with different sizes are obtained. The proposed method's runtimes are influenced by the tree size and dimensionality. These datasets are also the mix of binary class and multi-class and they cover various application domains. Hence, for better observation of runtimes in different scenarios, these datasets are chosen.

Each data set is given as input to the C4.5 decision tree construction algorithm. 10-fold cross-validation is applied for validation of the model and the technical evaluation measures [25] are also recorded as shown in Table 10.

From a data set, each time, one instance is input to two methods i.e. *R_Classify and E_Classify* to predict the class label and the classification times are noted. Thusly, the total time taken for finding class labels for all the instances in a data set w.r.t. each of the methods has been recorded and shown in Table 10.

The classification times presented in Table 10 depicts that the *E_Classify* outperforms the *R_Classify*. The total classification times of *R_Classify* and *E_Classify* on 10 data sets are 596.65 *ms* and 101.9 *ms* respectively. On the other hand, the average runtimes of *R_Classify* and *E_Classify* are 59.67 *ms* and 10.19 *ms* respectively. In Table 10, *P* values denote the percentage of runtime of *E_Classify* over *R_Classify.* The *P* values also describe that the runtimes of *E_Classify* are around 25% of *R_Classify.*

If the data set is large, composed of more attributes and the average fan-out of the attributes is high then, both the methods exhibit higher computational times. However, in all cases, the computational times of *E_Classify* are relatively less than those of *R_Classify*. If the dimensionality is high, then the proposed method exhibits better performance than the traditional method since attribute finding is done straight away using arrays without performing any comparisons. This fact has been observed on the data sets Anneal, Autos, Connect-4, and Hypothyroid where the runtimes of *E_Classify* are 19.6%, 20.32%, 16.36%, and 19.55% of *R_Classify* respectively. In such a way, the experimental results demonstrate that on average, *E_Classify* is at least four times faster than *R_Classify*.

Due to low complexity than the regular method, and usage of efficient data structures, the proposed method is more efficient than the standard decision tree based method. Step count of the proposed method is indeed less than the standard method. The detailed performance analysis using the *German* data set has also proved this fact (Table 9). Employment of arrays in the process and performing bit-level operations on the elements of arrays also helped improve the efficiency. For *E_Classify*, it is not necessary to perform the attribute name identifying task at a node of the tree. These all helped the reduction in the runtime of the proposed method. If the dimensionality of a data set is high, then the standard method has to spend more time on attribute name identifying task at each node of a specific path. In such scenarios, the performance of the proposed method is more significant than the standard method. Due to these reasons, the proposed method outperformed the standard method. For the conventional method if the attribute compared is the first one in the list and the outcome is the first value among its outcomes then, low runtimes take place.

Table 10: Classification time comparisons of *R_Classify* and *E_Classify* on UCI data sets
(Tr. Inst - No. of Training Instances, No. Atr - No. of Attributes, No. Cls - No. of Classes,
Prec. - Precision, Acc- Accuracy, T1 - Runtime of *R_Classify*, T2 - Runtime of *E_ Classify*).

Data set	Tr. Inst	No. Atr	No. Cls	Prec.	Recall	F-score	Acc (%)	AUC	Tree Size	No. of Leaves	No. Test Inst	Classification Time (*ms*)		**(T2/ T1)** % **(*P*)**
												(T1)	**(T2)**	
Anneal	898	39	6	0.941	0.942	0.941	94.20	0.965	346	306	898	0.877	0.172	19.6
Autos	205	26	7	0.854	0.854	0.853	85.37	0.913	215	194	205	0.122	0.024	20.3
Balance Scale	625	4	3	0.642	0.693	0.666	69.28	0.755	221	199	625	0.126	0.039	31.4
Connect-4	67557	42	3	0.795	0.795	0.795	79.45	0.907	15952	10635	67557	551.19	90.19	16.4
German	666	20	2	0.706	0.721	0.710	72.10	0.697	40	29	1000	0.591	0.143	24.2
Glass	214	10	7	0.556	0.579	0.562	57.94	0.775	221	199	214	0.044	0.011	26.6
Heart-c	303	14	5	0.770	0.769	0.767	76.89	0.835	200	171	303	0.084	0.026	31.7
Hypo thyroid	3772	30	4	0.891	0.923	0.906	92.33	0.818	570	467	3772	7.513	1.469	19.5
Nursery	12960	8	5	0.988	0.988	0.988	98.78	0.999	944	680	12960	35.562	9.673	27.2
Solar	1066	12	6	0.727	0.738	0.729	73.82	0.924	192	145	1066	0.542	0.155	28.6

4.4 Performance comparison with other classifiers

For performance comparison, five standard classifiers namely Naive Bayes (NB) [26], *k*-Nearest Neighbor (*k*-NN) [27], Random Forest (RF) [28], Support Vector Machines (SVM) [29], and AdaBoost (AB) [30] are considered. Reasons for considering these classifiers for comparison are: *Accuracy* is the key predictive performance evaluation metric of a classifier and in general, RF and SVM are the benchmark algorithms and occupy the top positions with respect to accuracy [6]. However, no classifier can be always the best. SVM and RF are also identified as popular, reliable, most robust, and benchmark classifiers thus normally used as the default classifiers in various Machine Learning applications. Boosting is treated to be the significant family of ensemble methods and also become a buzzword in computer vision and many other domains. AB is relied to be robust and successfully works for wide applications like text classification and spam filtering [23]. NB is an eager learner, a highly scalable probabilistic classifier and works well for large and high dimensional data [23]. *k*-NN is the simplest and lazy learner whose training time is zero and it has to spend all its time on classification. *k*-NN fits well for multi-modal classes for the domains in which the objects can have many class labels. Moreover, to compare with standard and versatile algorithms from different families, we have chosen these classifiers. RF is an ensemble tree based classifier, SVM is of support vector machines, AB belongs to the boosting category. However, NB and *k*-NN belong to Bayes (probabilistic) and nearest neighbours respectively [6, 7]. The study by Xindong et al. also reported that SVM, C4.5, *k*-NN, AdaBoost, and Naïve Bayes are the most influential classifiers [31, 32, 33, 34] of data mining placed in the top 10 positions [23].

Before using them for classification, the classifiers are validated using 10-fold cross validation. Training and validation sets are generated randomly. 10% of the instances are used as validation data set required to tune the parameters of the classifier. If parameter tuning is not required for a classification model, then the validation data is combined with training data. Thereafter, by using the selected values for the tunable parameters, the classification accuracy, and AUC [25] of each classifier is obtained by the average of 10 runs of 10-fold cross-validation. This methodology has been used because it helps to reduce the computational cost of the experimental work. Moreover, every instance is used in testing exactly once and is used in the training phase for *k*-1 times, and reduces the bias.

In our experiments, for RF the number of trees is set to 100. For *k*-NN, *k* value is considered as 5 to avoid expensive classification times and the Euclidean metric is used for finding distance matrix. The whole time taken by *k*-NN to predict the class label of an input instance is considered as the classification time. While testing the SVM, precomputed kernel is used in the experiments. Precomputed kernel achieves better accuracy and does not need parameter tuning which helps to save the computational time. AdaBoost and Naïve Bayesian are run using the standard parameters. Accuracy and AUC of various classifiers on the 10 UCI data sets are shown in Table 11 and Table 12 respectively.

Table 11: Accuracy results for different classifiers on 10 UCI data sets

Data set	Accuracy					
	k-NN	NB	RF	SVM	AB	E_Classify
Anneal	75.00	72.00	**96.00**	95.50	86.40	94.20
Autos	31.25	37.50	81.25	62.50	31.25	**85.37**
Balance Scale	95.24	**96.83**	95.24	92.06	80.95	69.28
Connect-4	75.20	76.46	81.60	**82.20**	80.50	79.45
German	72.00	**76.00**	74.00	72.00	71.00	72.10
Glass	76.19	38.10	**80.95**	**80.95**	42.86	57.94
Heart-c	75.56	80.80	**84.60**	83.76	76.30	76.89
Hypo thyroid	90.28	90.28	**100.00**	90.28	93.06	92.33
Nursery	82.30	82.88	96.42	95.78	92.42	**98.78**
Solar	76.36	74.00	80.66	**81.26**	76.58	73.82

Table 12: AUC results for different classifiers on 10 UCI data sets

Data set	AUC					
	k-NN	NB	RF	SVM	AB	E_Classify
Anneal	0.746	0.718	0.962	0.952	0.843	**0.965**
Autos	0.745	0.799	0.847	0.844	0.800	**0.913**
Balance Scale	0.984	0.833	0.833	**0.867**	0.806	0.755
Connect-4	0.758	0.783	**0.904**	0.868	0.814	0.907
German	0.694	0.741	**0.939**	0.645	0.613	0.697
Glass	**0.963**	0.815	0.748	0.790	0.806	0.775
Heart-c	0.792	0.827	**0.866**	0.821	0.784	0.835
Hypo thyroid	0.500	0.667	**1.000**	0.500	0.833	0.818
Nursery	0.848	0.803	0.966	0.968	0.944	**0.999**
Solar	0.801	0.887	0.849	0.804	0.782	0.924

The technical performance evaluation measures furnished in Table 11 and Table 12 describe that in most of the cases, RF and SVM exhibit relatively higher accuracy and AUC. At least for 5 data sets, either the RF or the SVM are showing maximum accuracy and AUC values. The accuracy measures of *E_Classify* are also fair enough and its average AUC (0.859) is higher than the SVM (0.806) and slightly lesser than that of the RF (0.891).

For performance comparison, experiments are conducted on UCI data sets and the results are furnished in Table 13. The runtimes are also plotted using the bar chart as shown in Fig. 5. The experimental results depict that the proposed method outperforms the other classifiers. After *E_Classify*, SVM is the fastest in classifying a test record. However, *k*-NN is the slowest among all. Random Forest is the next slowest one but it is the best performer with respect to accuracy. The classification times of Random Forest also depend on the number of trees generated. Among all the 10 UCI datasets, on Connect-4, the proposed method exhibited the best performance. Classification time of *E_Classify* on Connect-4 is 90.19 *ms*. Connect-4 is relatively a large data set (67557 records) and its dimensionality, i.e. 42, is also the maximum among all the data sets. More the dimensionality of a data set then higher the performance of the proposed method. This is because other classifiers have to identify the attribute in the process of predicting the class label. On the other hand, explicit identification of an attribute is not performed by the proposed method that helps save its processing time. In the case of Connect-4, at each node, the proposed method avoids 42 attribute name comparisons.

Table 13: Classification time results for six classifiers on 10 UCI data sets

Data set	Classification time (*ms*)					
	k-NN	NB	RF	SVM	AB	*E_Classify*
Anneal	349.03	237.91	56.20	0.284	21.351	0.172
Autos	3.422	0.339	2.554	0.067	0.373	0.024
Balance Scale	5.601	0.460	4.342	0.092	0.396	0.039
Connect-4	95773.95	8911.21	47172.83	303.91	7417.11	90.19
German	31.99	2.590	23.60	0.205	2.031	0.143
Glass	3.205	0.210	2.503	0.028	0.206	0.011
Heart-c	3.781	0.290	2.816	0.066	0.239	0.026
Hypo thyroid	281.07	18.440	455.20	2.930	15.97	1.469
Nursery	2199.52	125.04	1690.46	17.93	96.08	9.673
Solar	31.005	2.063	25.281	0.211	1.681	0.155

For each data sample, *k*-NN needs to compute the distance between the test sample and the other instances in the training data, and then it has to determine *k* nearest neighbours using a metric like Euclidean distance. This process increases the processing time of *k*-NN. Consequently, when the data set size is large, *k*-NN exhibits even worse results. The results on Connect-4 depict this case. On the other hand, to compute the posterior probability w.r.t each class, the computations to be performed by Naive Bayes is more in number and those operations are costlier than the operations adopted by *E_Classify*. Hence, the computation times of Naive Bayes are also higher than the *E_Classify*. Classification times of Random Forest depend on the number of trees generated. In our experiments, we have considered the number of trees to be 100. Hence, to determine the prediction result for an instance, it is required to predict class labels using 100 trees, and the class label that obtains the maximum votes stands as the prediction result. This process requires more time than the methods that use a single tree. However, there is no substantial difference in the classification times of SVM and the proposed method. SVM uses the maximum margin hyperplane to predict the class label of a test sample where the computations are not much expensive. AdaBoost is an ensemble based classifier and it is also slower than the proposed method. It predicts the class label by calculating the weighted average of the weak classifiers which is computationally costlier than the *E_Classify*.

From the experimental results, it can be concluded that the proposed method's classification times are better than those of various other standard classifiers belonging to different families and also the regular tree based method.

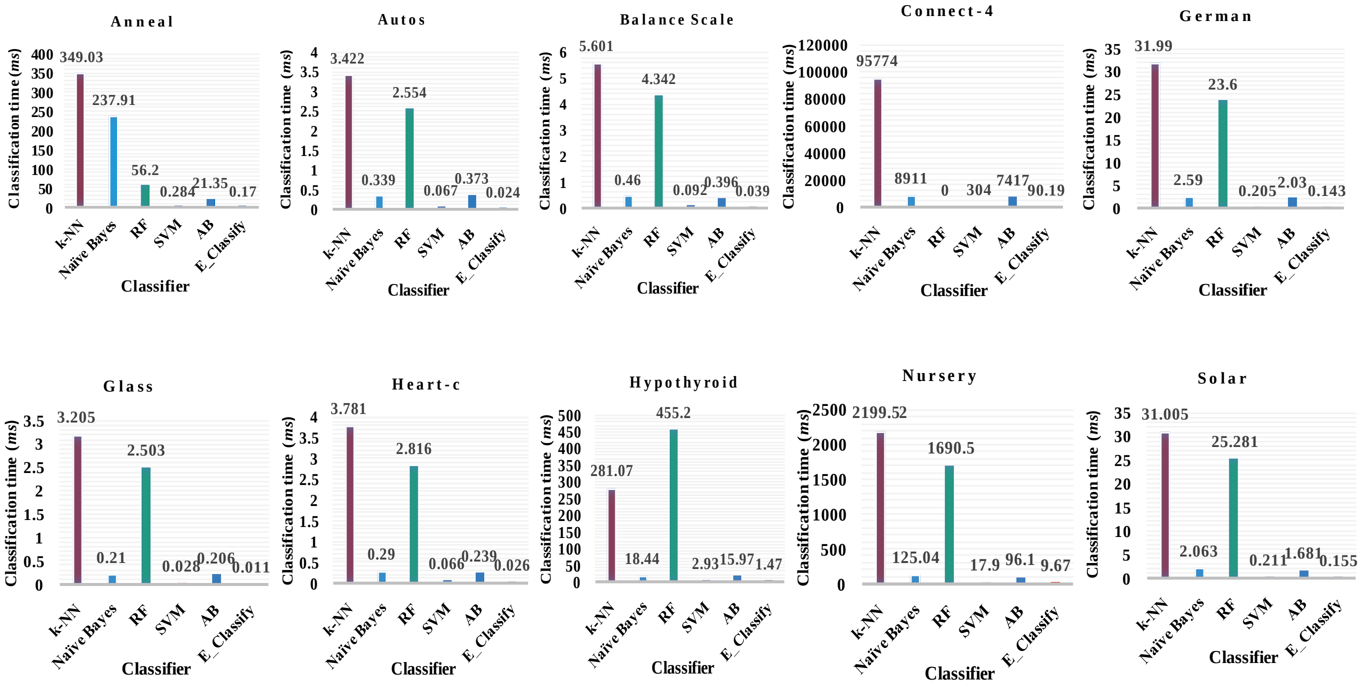

Fig. 5 Classification times comparison of various classifiers on 10 UCI data sets.

5 Conclusions

Classification is a vital concept of data mining and machine learning which is predominantly used in various sectors for the prediction of the class label of an unseen instance. Among various classification models, decision trees have got high significance due to their merits. In the research area of classification through decision trees, improving the computational performance of the classification phase has not been given much attention.

In this paper, an efficient method that is applicable to decision trees has been introduced to predict the class label of the test instance quicker than the regular tree traversal method. The experiments conducted on UCI data sets proved that the proposed method is at least four times faster than the regular method. The experimental results also verified that in comparison to other standard classifiers *k*-NN, Naive Bayes, Random Forest, SVM, and AdaBoost which belong to different families, the proposed method's performance is significant. The proposed method's step count is low and also due to the employment of various efficient data structures like arrays and bit-level operations, it achieved significant computational efficiency.

The work described in this paper can be considered as the first step in the development of fast classifying algorithms and it can serve as a basic stone that can be further improved. The proposed method is designed to work on a single decision tree and fits well for the data sets with discrete attributes. Extending the research for more optimization for an ensemble of trees and other classifiers can be taken up as the future work.

References

[1] A.T. Azar and S.M. El-Metwally, "Decision tree classifiers for automated medical diagnosis", Neural Comput & Applic, 23(7), pp. 2387-2403, 2013. https://doi.org/10.1007/s00521-012-1196-7

[2] G. Nie, W.L. Rowe, Y. Zhang, Tian, Y. Shi, "Credit card churn forecasting by logistic regression and decision tree", Expert Syst Appl., 38(12), pp. 15273–15285, 2011. doi: 10.1016/j.eswa.2011.06.028

[3] J. Naga Muneiah and C. Subba Rao, "An Efficient Probability Estimation Decision Tree Postprocessing Method for Mining Optimal Profitable Knowledge for Enterprises with Multi-Class Customers", Inteligencia Artificial, 22(64), pp. 63-84, 2019. https://doi.org/10.4114/intartif.vol22iss64pp63-84

[4] Naga Muneiah Janapati and Ch. D. V. Subba Rao, "Customer's class transformation for profit maximization in multi-class setting of Telecom industry using probability estimation decision trees", Journal of Intelligent & Fuzzy Systems, 37(6), pp. 8167-8197, 2019. DOI: 10.3233/JIFS-190628

[5] Quiang Yang, Jie Yin, Charles Ling, and Rong Pan, "Extracting Actionable knowledge using decision Trees", IEEE Transactions on Knowledge and Data Engineering, 17(1), pp. 43-56, 2007. doi: 10.1109/TKDE.2007.9

[6] Manuel Fern´andez-Delgado, Eva Cernadas, Sen´en Barro, and Dinani Amorim, "Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?", Journal of Machine Learning Research, vol. 15, pp. 3133-3181, 2014.

[7] Stefan Lessmann, Bart Baesens, Hsin-Vonn Seow, and L. C. Thomas, "Benchmarking state-of-the-art classification algorithms for credit scoring: An update of research", European Journal of Operational Research, 247(1), pp. 124-136, 2015. DOI: 10.1016/j.ejor.2015.05.030

[8] Marina Sokolova and Guy Lapalme, "A systematic analysis of performance measures for classification tasks", Information Processing and Management, 45, pp. 427–437, 2009. DOI: 10.1016/j.ipm.2009.03.002

[9] Tjen-Sien Lim and Wei-Yin Loh, "A Comparison of Prediction Accuracy, Complexity, and Training Time of Thirty-Three Old and New Classification Algorithms", Machine Learning, 40, 203–228, 2000. https://doi.org/10.1023/A:1007608224229

[10] M. Reif, F. Shafait, A. Dengel, "Prediction of Classifier Training Time Including Parameter Optimization", In: Bach J., Edelkamp S. (eds) KI 2011: Advances in Artificial Intelligence. KI 2011, Springer, Berlin, Heidelberg, 2011. https://doi.org/10.1007/978-3-642-24455-1_25

[11] T. Doan, J. Kalita, "Predicting run time of classification algorithms using meta-learning", Int. J. Mach. Learn. Cybern., 8, pp.1929–1943, 2017.

[12] Long Bing Cao, D. Luo, C. Zhang, "Knowledge actionability: Satisfying technical and business interestingness", International Journal of Business Intelligence and Data Mining, 2(4), pp. 496-514, 2007. DOI: 10.1504/IJBIDM.2007.016385

[13] M. Panda and A. Abraham, "Hybrid evolutionary algorithms for classification data mining", Neural Comput & Applic, 26(3), pp. 507-523, 2015. https://doi.org/10.1007/s00521-014-1673-2

[14] L. E. Hamid and S.A.R. Al-Haddad, "Automated Leaf Alignment and Partial Shape Feature Extraction for Plant Leaf Classification", Electronic Letters on Computer Vision and Image Analysis, 18(1), pp.37-51, 2019. DOI: https://doi.org/10.5565/rev/elcvia.1143

[15] V. Huddar, B. K. Desiraju, V. Rajan, S. Bhattacharya, S. Roy, and C. K. Reddy, "Predicting Complications in Critical Care Using Heterogeneous Clinical Data", IEEE Access, vol. 4, pp. 7988-8001, 2016. doi: 10.1109/ACCESS.2016.2618775

[16] A. I. Weinberg and M. Last, "Selecting a representative decision tree from an ensemble of decision-tree models for fast big data classification", J. Big Data, 6(23), 2019. https://doi.org/10.1186/s40537-019-0186-3

[17] A. Ashari, I. Paryudi, and A. Min, "Performance Comparison between Naïve Bayes, Decision Tree and k-Nearest Neighbor in Searching Alternative Design in an Energy Simulation Tool", Int. J. Adv. Comput. Sci. Appl., 49(11), pp. 33–39, 2013. DOI: 10.14569/IJACSA.2013.041105

[18] P. Hurtik and I. Perfilieva, "Fast Training and Real-Time Classification Algorithm Based on Principal Component Analysis and F-Transform", Proc. of 2018 Joint 10th International Conference on Soft Computing and Intelligent Systems (SCIS) and 19th International Symposium on Advanced Intelligent Systems (ISIS), IEEE, 275-280, 2018. doi: 10.1109/SCIS-ISIS.2018.00056.

[19] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, M. A. Kozuch, Onur Mutlu, P. B. Gibbons, T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM", IEEE Computer Architecture Letters, 14(2), 127-131, 2015. doi: 10.1109/LCA.2015.2434872

[20] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2, September 2016.
[21] Randall Hyde, Understanding the Machine, vol. 1, 2nd Ed., No Starch Press, 2020.
[22] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers, 1993.
[23] W. Xindong, J. Vipin Kumar, R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, N. Angus, Bing Liu, S. Philip, Y.Z. Zhou, S. Michael, D.J. Hand, and D. Steinberg, "Top 10 algorithms in data mining. Knowledge and Information Systems", 14(1), pp. 1-37, 2008. DOI 10.1007/s10115-007-0114-2.
[24] D. Dua, C. Graff, UCI Machine Learning Repository, http://archive.ics.uci.edu/ml. 2019.
[25] Huang J. and Ling C. X., "Using AUC and Accuracy in Evaluating Learning Algorithms," IEEE Trans. on Knowledge and Data Engineering, 17(3): 299-310, 2005. DOI: 10.1109/TKDE.2005.50
[26] R.O. Duda, D.G. Starc & P.E. Hart, Pattern classification, Wiley, 2000.
[27] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression", The American Statistician, 46(3), pp.175-185, 1992.
[28] L. Breiman, Random Forests, Machine Learning, 45 (1), pp. 5-32, 2001. https://doi.org/10.1023/A:1010933404324
[29] C. Cortes and V. Vapnik, "Support-vector networks", Machine Learning, 20, pp. 273–297, 1995. https://doi.org/10.1007/BF00994018
[30] Y. Freund and R. E. Schapire, "Experiments with a New Boosting Algorithm", Machine Learning: Proceedings of the Thirteenth International Conference, 1996.
[31] Duleep Rathgamage Don, I. E. Iacob, "DCSVM: fast multi-class classification using support vector machines", International Journal of Machine Learning and Cybernetics, 11, pp. (433–447), 2020. https://doi.org/10.1007/s13042-019-00984-9
[32] R. R. Yager, "An extension of the naive Bayesian classifier", Information Sciences, 176, pp. 577–588, 2006.
[33] Filip Kadlček and Otto Fučík, "Fast and Energy Efficient AdaBoost Classifier", Proceedings of the 10th FPGAworld Conference, 2, pp.1-5, 2013.
[34] S. Ruggieri, "Efficient C4.5", IEEE Transactions on Knowledge and Data Engineering, 14(2), pp. 438-444, 2002. doi: 10.1109/69.991727.
[35] J. Han, J. Pei, and M. Kamber, Data Mining: Concepts and Techniques. 3rd edition, Elsevier, 2011.

Appendix

Sample 2-class data set.

Age	Income	Student	Credit_rating	Class: Buys_computer
youth	high	no	fair	no
youth	high	no	excellent	no
middle_aged	high	no	fair	yes
senior	medium	no	excellent	no
senior	low	yes	excellent	no
senior	low	yes	fair	yes
middle_aged	low	yes	excellent	yes
youth	medium	no	fair	no
youth	low	yes	fair	yes
senior	medium	yes	excellent	no
youth	medium	yes	excellent	yes
middle_aged	medium	no	excellent	yes
middle_aged	high	yes	fair	yes
senior	medium	no	fair	yes