2009 Ninth International Conference on Intelligent Systems Design and Applications

Mining Models for Failing Behaviors

Cláudia Antunes
Department of Computer Science and Engineering
Instituto Superior Técnico / Technical University of Lisbon
Lisbon, Portugal
claudia.antunes@ist.utl.pt

***Abstract*— Understanding the causes for failure is one of the bottlenecks in the educational process. Despite failure prediction has been pursued, models behind that prediction, most of the time, do not give a deep insight about failure causes. In this paper, we introduce a new method for mining fault trees automatically, and show that these models are a precious help on identifying direct and indirect causes for failure. An experimental study is presented in order to access the drawbacks of the proposed method.**

Keywords-mining behaviors, fault trees; pattern mining

I. INTRODUCTION

Failure models have been a reality for a long time, mostly in areas where failures are not tolerable, like in nuclear facilities or aviation. One of the most well-known is the Fault Tree Analysis [15], where a logic tree is designed from the scratch manually by domain experts. Fault trees are then a model that makes explicit all possible causes for failure.

In educational environments such a failure model might be of a great utility, since it could be used as a prevention tool. For this reason, finding the causes for students' failure is one of the major goals in educational data mining. Nevertheless, the research has been focused mainly on determining students' models (see for example [3] and [5]), and more recently on mining frequent behaviors ([2] and [10]). Exceptions to this general scenario are the works [12] and [14] that try to identify failure causes. In the first case, causes for failing in the first year at university are determined, and in the second one, causes for failing to learn object-oriented programming are investigated. Moreover, and despite the expertise in the area, the manual design of failure models is too hard, most of all due to the fact that all the actors in the educational context are humans.

Since failure is a constant in educational environments, we can think on identifying those models based on failure data. In this paper, we propose a new automatic method to design fault trees for students' failure, based on their evaluation results.

The rest of the paper is organized as follows: next, we formalize the notion of fault trees and state their advantages over traditional classifiers in the context of the educational process. In section 3, we designed an automatic method to construct those models and proposed an implementation of it, by using association rules. The paper concludes with the analysis of experimental results and some guidelines for future work.

II. FAULT TREE ANALYSIS

A Fault Tree (FT for short) is a recursive data structure, with the root corresponding to the failure in analysis. Its particularity is that instead of having simple nodes representing propositions, each internal node contains a proposition and a logic gate. In this manner, the proposition results from the combination of its children through the logic gate.

Formally, a *fault tree* is a proposition (i.e. a pair attribute / value) or a triple (proposition, gate, set of fault trees).

Usually, logic gates are restricted to AND and OR gates, but any others may be used. In this paper, we only consider these two types of gates. Note, that an AND gate represents a conjunction and an OR gate a disjunction. In this manner, the first gate implies that all descendants have to be satisfied in order to set the parent, while the OR gate only requires the satisfaction of one of the descendants to be activated.

Consider for example, the enrolment of a student at an under graduation subject: he is submitted to an exam (*Exam1*) and has to deliver a final project (*Proj*). Whenever student fails on the exam he has the opportunity to be evaluated again (*Exam2*). In this case, the student is approved if he has a grade different from *F* on the project and on one of the exams. Figure 1 illustrates a fault tree that describes the possible causes for student's failure in that subject: failing on the project or failing both on exams.

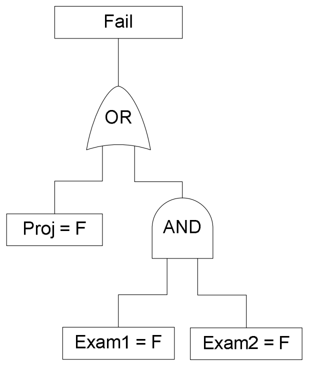

Figure 1. A simple fault tree for student failure.

A. *Fault Trees versus traditional classifiers*

Once discovered, the analysis of fault trees is simple, and the direct and indirect causes for failure are all identified. With such models, understanding the failure causes is easier,

978-0-7695-3872-3/09 $26.00 © 2009 IEEE
DOI 10.1109/ISDA.2009.122

since each occurrence may be explained, i.e., decomposed. For example, the FT above could be extended to include an explanation why students fail on the project.

Reasons to consider fault trees as better failure models than traditional classifiers, for example decision trees [8], are several. The first reason is that decision trees are disperse failure models, which means that for identifying failure causes is necessary to discard all branches that end with a non-failure classification, and analyzing the rest of them. Indeed, decision trees do not try to capture the causes for failure, but instead, they choose propositions that better differentiates the instances on resulting branches, which, in turn, makes usual the existence of several branches for the same class.

A second, but more important, reason is that each attribute (proposition in a node) is seemed as an observable event that does not requires an explanation. Definitely, a node in a decision tree is just an occurrence and its successors are just other requirements to achieve a certain classification value. On the other hand, in a fault tree each depth level corresponds to an explanation for the precedent level.

It is also important to note, that decision trees do not distinguish temporal attributes, this is, they treat all attributes equally, ignoring their instant of occurrence.

III. Fault Tree Construction

Discovering fault trees from data requires a clear formulation, where instances and attributes are defined precisely.

An *attribute* is a variable which represents a feature or an event. While *features* are static characteristics for entities, like name and age; an *event* corresponds to the occurrence of some important fact at a certain instant of time, like the grade obtained in a test. Let I be a set of instances, A a set of attributes, and C a set of possible classes; an instance x_i from I is described by a list of m attributes from A, ordered in accordance to the $\prec$ partial order, and is represented as $x_i=x_{i1}x_{i2}\ldots x_{im}c_i$, where $c_i\in C$ corresponds to the class of x_i.

The $\prec$ partial order is defined as follows: all feature attributes are less than every event attribute; event attributes are ordered according to their temporal order. Formally:

$$\begin{aligned}&\forall a, e \in A{:}\ a\ is\ feature \wedge e\ is\ event \Rightarrow a \prec e\\ &\forall a_1, a_2 \in A{:}\ a_1\ isfeature \wedge a_2\ isfeature \Rightarrow a_1 = a_2\\ &\forall e_1, e_2 \in A{:}\ e_1\ is\ event \wedge e_2\ is\ event \wedge\ e_1 before\ e_2\\ &\qquad\qquad \Rightarrow e_1 \prec e_2\end{aligned} \quad (1)$$

where *before* is defined among temporal instants as proposed in [7]. Note that the order among feature attributes is non-relevant.

Given a set of instances from I and the failure value for the class c_f, the problem of finding the fault tree for c_f consists on identifying all the combinations of direct causes for an instance to be classified as c_f and simultaneously, identifying all direct causes for those causes, recursively, considering only preceding attributes according to the partial order $\prec$.

Mining fault trees may be done as in Figure 2:

```
FT extendFT (FT ft, Instances data)
 FT[] children←process(ft.root,data)
 if (~∃c∈children) then
  return ft
 else
  ft←addChildren(ft,children)
  for all $t_i$∈children(ft) do
   if ($t_i$.isAND) then
    for all $t_{ij}$∈children($t_i$) do
     d2←reduceData($t_{ij}$.root,data)
     $t_{ij}$←extendFT($t_{ij}$,d2)
   else
    d2←reduceData($t_i$.root,data)
    $t_i$ ← extendFT($t_i$,d2)
  return ft
```

Figure 2. Pseudo code for algorithm extendFT.

Initially, the algorithm is called with a fault tree only constituted by a node, corresponding to the class failure value and the entire dataset, with instances described by all available attributes.

The first step consists on finding the direct causes for failure; this is, to achieve the class failure value. If some causes are identified, the fault tree is extended by adding the causes as children, the fault tree gate is set as an OR, whenever there is more than one cause. In the presence of only one cause, the gate is set to an AND only if that cause is a composed one, which means that it is required that multiple situations occur simultaneously to cause the failure.

After extending the tree, the algorithm mines each sub-tree recursively, only considering instances described by a subset of attributes that are less than the sub-tree root attribute, in accordance to the $\prec$ partial order.

The algorithm stops whenever it does not discover any cause for the root node or it corresponds to a feature attribute. Note that procedure *reduceData* creates a new dataset with instances described by fewer attributes as the depth of the tree increases; and it uses the new node to set the dataset class.

The simplicity of the proposed algorithm just opens one problem: to identify direct causes for achieving required failure values. Next, we propose an implementation of that algorithm using Class Association Rules for identifying direct causes for failure.

A. Class Association Rules

Association analysis is an unsupervised task, which tries to capture existing dependencies among attributes and its values, described as association rules. The problem was first introduced in 1993 [1], and is defined as the discovery of “all association rules that have support and confidence greater than the user-specified minimum support and minimum confidence, respectively”.

An association rule corresponds to an implication of the form $A\Rightarrow B$, where A and B are propositions (sets of pairs *attribute / value*), that expresses that when A occurs, B also occurs with a certain probability (the *rule’s confidence*). The *support* of the rule is given by the relative frequency of instances that include A and B, simultaneously. In this case, *A* and *B* are named the *antecedent* and the *consequent* of the rule, respectively.

A *Class Association Rule* (CAR for short) is an association rule, which consequent is a single proposition related to the value of the class attribute. In this manner, a set of CARs can be seen as a classifier [11].

The main drawback on mining CARs and association rules is the explosive number of discovered rules and the human inability to deal with that explosion.

Several efforts have been made to minimize this effect, namely on defining interestingness measures to filter off non-relevant rules (see [13] for an overview) or to reduce the number of discovered ones [4].

B. Finding the Fault Tree with CARs

Association rules are privileged tools for representing direct causes, since they relate propositions to one another, in a quantified manner. Indeed, both confidence and support measure the certainty and the coverage for the cause, allowing for assessing the rule contribution to the occurrence of the consequent.

By making use of CARs, the implementation of the proposed algorithm is trivial (Figure 3). Note the need of imposing the minimum thresholds for support and confidence (*sp* and *cf*, respectively).

```
FT CARExtendFT(FT ft, Instances data,
                     float sp, float cf)
  CAR[] rul←process(ft.root,data, sp,cf)
  if (~∃r∈rul) then
    return ft
  else
    ft←addChildren(ft,rul)
    for all $t_i$∈children(ft) do
      if ($t_i$.isAND) then
        for all $t_{ij}$∈children($t_i$) do
          d2←reduceData($t_{ij}$.root,data)
          $t_{ij}$←CARExtendFT($t_{ij}$,d2,sp,cf)
      else
        d2←reduceData($t_i$.root,data)
        $t_i$←CARExtendFaultTree($t_i$,d2,sp,cf)
    return ft

CAR[] process(Attribute root,
     Instances data, float sp, float cf)
  Apriori ap ← new Apriori()
  ap.setLowerBoundMinSupport(minSp)
  ap.setMinMetric(minCf)
  ap.setClassIndex(data.classIndex())
  CAR[] rules ← ap.mineCARs(data)
  for all r∈rules do
    if (r.support<sp
       && r.confidence<cf) then
      rules ←rules\{r}
  return rules
```

Figure 3. Pseudo code for algorithm `CARExtendFT` and function `process`.

It is also important to note that the procedure for processing each tree node (process in Figure 3) just mines all CARs with root as the consequence, and filters out any that do not satisfy the interestingness thresholds specified. (Note that Apriori is a class in WEKA software [16], for implementing the apriori algorithm [1]).

IV. Experiments

In this chapter we study how interestingness thresholds, for support and confidence, influence on the size and accuracy of the fault tree discovered. To support our study, the dataset used stores the results of students enrolled in the last five years, in the subject of *Foundations of Programming* of an undergraduate program at *Instituto Superior Técnico*. The dataset has 2050 instances, each one with 16 event attributes describing evaluation results. From these, 11 corresponds to weekly exercises (E_i), and the other five to an intermediate test (*T1*), a final project (*P*), a second test (*T2*), another optional test (*T3*) and an optional *Oral* . All have a classification from *A* to *F* (and *NA* – meaning not evaluated). The last attribute corresponds to the classification obtained at the end of the semester (*App* – if *A*, *B*, *C*, *D* or *E*, and *Fail* for *F*). Table 1 exemplifies three cases: the first student is approved, missing the last test (*T3*) and *Oral*; the second one fails on *T2* and is approved on *T3*, and the third one fails, since he is not evaluated on *T2* and fails both on *T3* and on *Oral*.

TABLE I. Example of three students' records

E1	**…**	***E6***	***T1***	***E7***	**…**	***E11***	***P***	***T2***	***T3***	***Oral***	***App***
A	…	A	B	A	…	A	A	B	NA	NA	**App**
A	…	C	C	A	…	B	C	F	D	NA	**App**
B	…	E	D	D	…	D	D	NA	F	F	**Fail**

The accuracy of the model (equation 2) will be assessed as usual, by consider its global precision, its sensibility and its specificity (see equation 3). While the first measure gives an idea about how correct are the predictions made by the model, sensibility captures the ability to identify positive cases for a class and specificity the ability to exclude negative cases for a class. The three measures are computed using equations 2 and 3, where *TP* and *TN* correspond to *true positives* (positive cases predicted as positive) and *true negatives* (negative cases predicted as negative), and *FP* and *FN* to *false positives* (negative cases predicted as positive) and *false negatives* (positive cases predicted as negative), respectively).

$$accuracy = \frac{TP+TN}{TP+FN+FP+TN} \qquad (2)$$

$$sensibility = \frac{TP}{TP+FN} \quad specificity = \frac{TN}{TN+FP} \qquad (3)$$

It is clear, from our experiments, that our model, the fault tree, achieves interesting results, reaching 94% of accuracy, which compares with the 96% achieved with decision trees trained in the same data with algorithm C4.5 [9].

A deep analysis of the results (Figure 4- top) shows that the accuracy rises while confidence decrease from 100% to 95%; and then, it begins to decrease rapidly when confidence continues to decrease. This scenario is repeated with different levels of support. Best results are then achieved with 5% of support and 95% of confidence, reaching 94% of accuracy.

An important issue is that for small values of support (≥5%) and confidence≥85%, the model classifies all instances as failure, like it happens when the algorithm does not find any rule to extend the model (support≥20%). Similarly, for higher levels of support (≥10%), with the decrease of confidence (≥80%) the phenomenon is repeated.

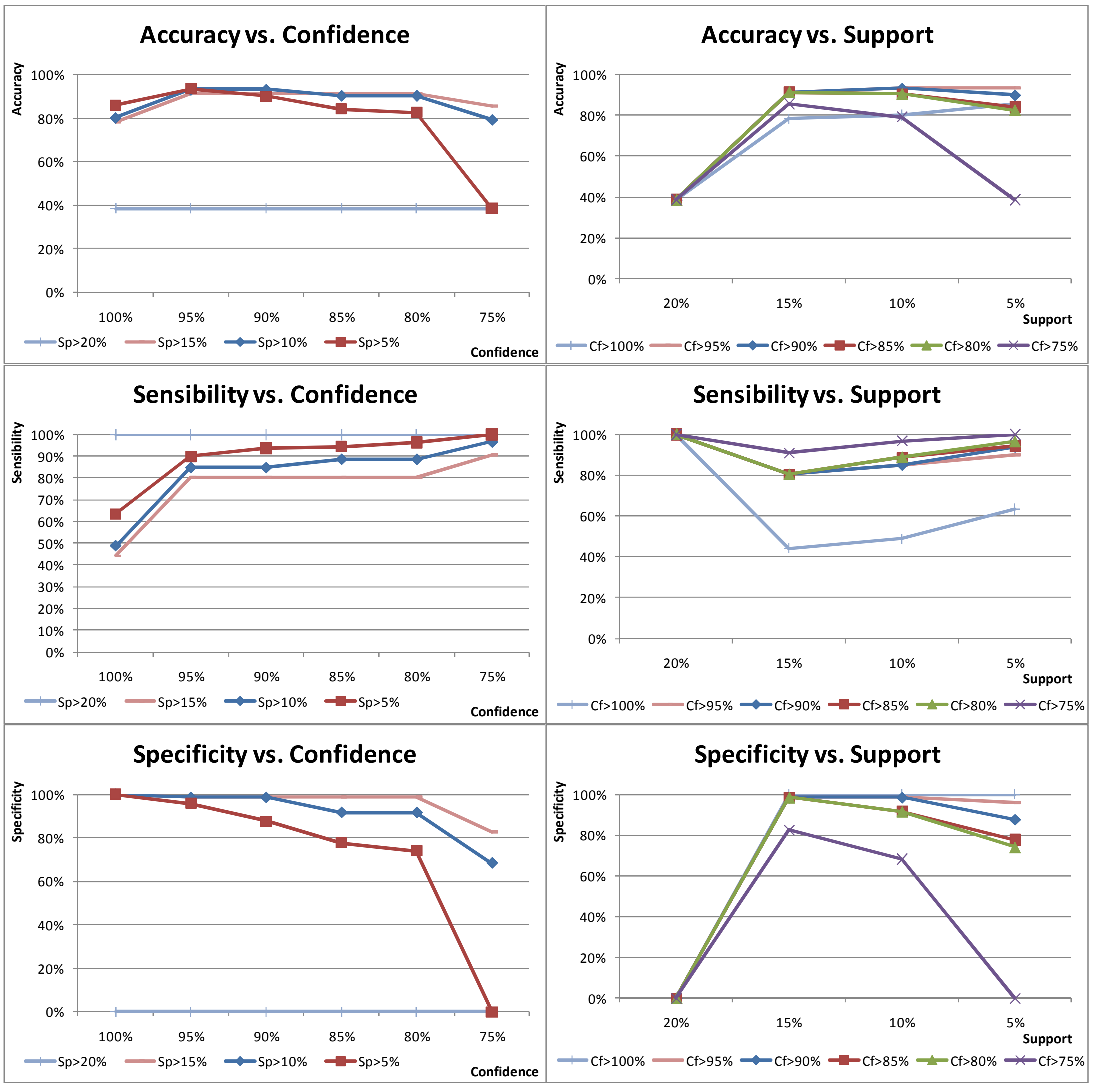

Figure 4. Accuracy, sensibility and specificity for different thresholds of confidence and support

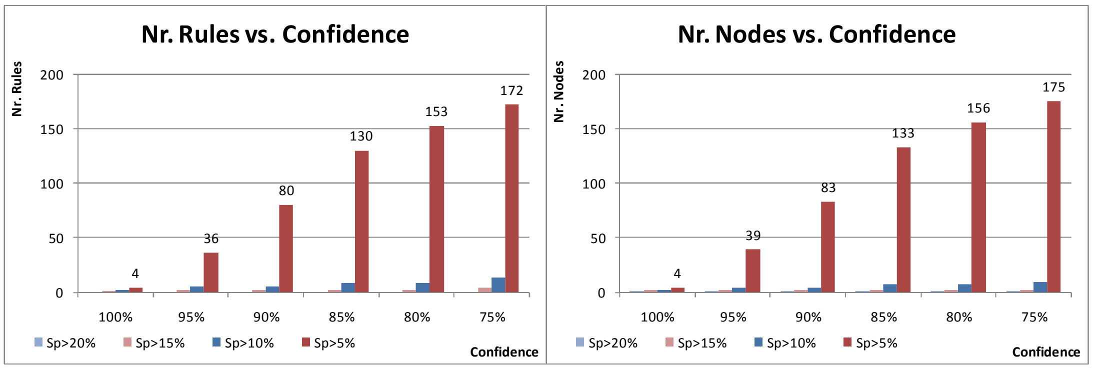

Figure 5. Number of rules discovered and tree nodes for different values of confidence

The values for sensibility (Figure 4-middle) and specificity (Figure 4-bottom) also show interesting results. For support≥5% and confidence≥95% it reaches 90% and 96% for sensibility and specificity, respectively. These results show that the fault tree discovered is able to recognize 90% of failures, only making 4% of mistakes (given by the complement of specificity – 100%-specificity).

Naturally, these results derive from the number of discovered rules, by using class association rules. With the small values of support and confidence, the number of discovered rules explodes, and consequently the number of nodes per tree (Figure 5). It is interesting to note that the number of nodes exceeds the number of rules discovered in these limit situations. This is due to the fact that the explosion results from the existence of long patterns. In our algorithm, long patterns represent conjunctions – AND nodes, that result on more nodes in the tree. As a result of long trees, the accuracy and sensibility falls, as stated in the Occam's razor [6].

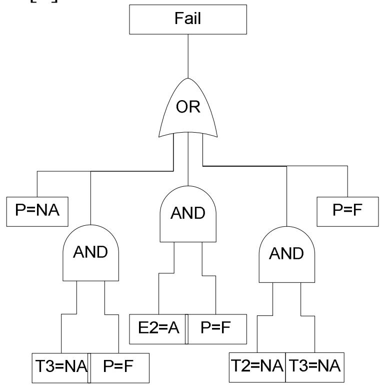

Figure 6. Fault tree for 10% of support and 95% of confidence.

The best fault tree found reaches 94% of accuracy with 35 nodes as noted, but for a support of 10% and the same level of confidence (95%) the fault tree found has only 3 nodes and achieves 93% of accuracy (Figure 6). However, both these trees have only one level of depth, which means that they do not present any indirect cause for failure. In order to achieve that, we have to decrease the confidence threshold to 90%, which results in a tree with 151 nodes in two levels, but identifying several indirect causes.

V. CONCLUSIONS

The existence of automatic means to create failure models, from failure data, brings a new breath in the understanding of failure behaviors. Indeed, these means can reveal unknown events as causes for failure, and consequently, contribute to improve the design and organization of curricular units.

In this paper, we stated the problem of developing fault trees using failure data, and proposed a simple approach to deal with it. Experimental results show that this approach is sufficient to create accurate models, which can be used effectively. However, it is clear that the fault trees discovered are too large, and some post-processing applied to the discovered rules should reduce their size enormously. This post-processing, however, has to compact the identified rules combining them in order to reduce the number of branches in the tree. One possibility is to combine rules that share propositions, for example using Boolean algebra properties, like the distributive property.

Naturally, other approaches resulting from the work on classification are expected to overcome the proposed here, which in turn would create more comprehensive models, i.e. smaller models. Indeed, measures like the information gain based on the entropy, can reach more accurate models than class association rules. Measures like that should be explored in order to implement the proposed algorithm in a more efficient.

From this work, is now clear that fault trees can be trained based on past data, and that those trees present failure causes more clearly than other classifiers, even decision trees.

REFERENCES

[1] R. Agrawal., T. Imielinski, A. Swami, Mining association rules between sets of items in large databases. ACM SIGMOD Conf. on Management Data, pp. 207-216. 1993

[2] C. Antunes, Acquiring background knowledge for intelligent tutoring systems". Baker, Barnes and Beck (eds.), Educational Data Mining 2008, pp. 18-27. 2008

[3] R. Baker, A. Carvalho, Labeling student behavior faster and more precisely with text replays. Baker, Barnes and Beck (eds.), Educational Data Mining 2008, pp. 38-47. 2008

[4] R. J. Bayardo, R. Agrawal, D. Gunopulos, Constraint-based rule mining in large, dense databases. Proc Int'l Conf Data Engineering, pp. 188-197, IEEE Press 1999

[5] J.E Beck, Difficulties in inferring student knowledge from observations (and why should we care). Workshop of Educational Data Mining – Int'l Conf of Artificial Intelligence in Education, pp. 21-30. 2007

[6] P. Domingos, The role of Occam's razor in knowledge discovery. Data Minining and Knowledege Discovery, 3(4): 409-425. 1999

[7] P. Hayes, A datalog of temporal theories, Tech report UIUC-BI-AI-96-01, University of Illinois, 1995

[8] J. Quinlan, Induction of decision trees. Machine Learning , 81-106. 1986

[9] J. Quinlan, C4.5: programs for machine learning. Morgan Kaufmann. 1993

[10] C. Romero, S. Ventura, P. Espejo, C. Hervás, Data mining algorithms to classify students, Baker, Barnes and Beck (eds.), Educational Data Mining 2008 (pg. 8-17). 2008

[11] T. Scheffer, Finding association rules that trade support optimally against confidence. European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD'01) (pp. 424-435). Springer-Verlag. 2001

[12] J.F. Superby, J.-P. Vandamme, N. Meskens, Determining of factors influencing the achievement of first-year university students using data mining methods. . Intelligent Tutoring System (ITS): Educational Data Mining Workshop. pp. 37-44, 2006

[13] S. Ventura, C. Romero, C. Hervas, Analyzing rule evaluation measures with educational datasets: a framework to help the teacher. Baker, Barnes and Beck (eds.), Educational Data Mining 2008 (pg. 177-181). 2008

[14] M.H. Vee, B. Meyer, K.L. Mannock, Understanding novice errors and error paths in object-oriented programming through log analysis.

Intelligent Tutoring System (ITS): Educational Data Mining Workshop. pp. 13-20, 2006

[15] W.E. Vesely, F. Goldberg, N.H. Roberts, D.F. Haasl, Fault tree handbook. U.S. Nuclear Regulatory Commission, 1981.

[16] I. Witten, E. Frank, Data mining: practical machine learning tools and techniques with java implementations. Morgan Kaufmann. 2000.