Indexing Rules in Rule Sets for Fast Classification

Tony Lindgren
Department of Computer and Systems Sciences
Stockholm University
Borgarfjordsgatan 12
164 40, Kista, Sweden
tony@dsv.su.se

ABSTRACT

Using sets of rules for classification of examples usually involves checking a number of conditions to see if they hold or not. If the rule set is large the time to make the classification can be lengthy. In this paper we propose an indexing algorithm to decrease the classification time when dealing with large rule sets. Unordered rule sets have a high time complexity when conducting classification; we hence conduct experiments comparing our novel indexing algorithm with the standard way of classifying ensembles of unordered rule sets. The result of the experiment shows decreased classification times for the novel method that are ranging from 0.6 to 0.8 of that of the standard approach averaged over all experimental datasets. This time gain is obtained while retaining an accuracy ranging from 0.84 to 0.99 with regard to the standard classification method. The index bit size used with the indexing algorithm influence both the classification accuracy and time needed for conducting the classification task.

CCS Concepts

•**Theory of computation** → *Approximation algorithms analysis;* •**Computing methodologies** → Rule learning; Ensemble methods; •**Information systems** → Expert systems;

1. INTRODUCTION

Unordered rule sets are a set R of rules were each rule has the form of r_i = if $c_1 ... c_n$ then $Class_{c \in C}$ where $r_i \in R$ denotes the i:th rule of the rule set with conditions 1 to n which predicts a class c from the set of classes C. When classifying with unordered rule sets all rules have to be checked and typically a (possibly weighted) voting scheme are eventually used for deciding the class.

When using ordered rule sets or decision lists which are read top-down and the first rule which have all conditions fulfilled are used for classification purposes, see [1]. The last rule of such an ordering often contains the default rule which predict a whole (default) class. This gives a more compact representation compared to unordered rule sets, and as a consequence ordered rule sets are significantly faster when used for classifying as the number of conditions to check is much less than when using unordered rule set.

To illustrate this consider a decision list for a binary decision problem with 5 rules that predict the positive class and 1 default rule, which predict the negative class. Consider also a similar unordered rule set with 5 positive rules and 5 negative rules. Let's say that in the dataset we want to classify both classes are equally common. Assume that when classifying the positive class on average there is a need to check the conditions on half of the rules, i.e. 2.5 rules for the positive class. For the negative class we always have to check 5 rules. If we were to classify 100 examples we then need to investigate for 2.5*50 + 5*50 = 375 rules.

Depending on the inducing algorithm the conditions of the unordered rule sets could involve more conditions than that of ordered rules, as the rules need to be able to *stand on their own*. When considering the above case with an unordered rule set we have to investigate all rules for all examples, giving 10*100 = 1000 rules, with possibly more conditions in each rule.

Given this extra complexity, what are the benefits of unordered rule set compared to using ordered rule set? The answer comes in two flavors one which is interpretability, with unordered rule sets each rule *stands on its own* meaning that no extra information is needed for interpreting it. Considering rules in decision lists which are dependent on their ordering most evidently is this true for the default rule which motivates its prediction by *all previous failed* which is not that informative, when trying to interpret/understand the reasons behind a classification.

The second flavor is more expressiveness and better classification power, see [2, 3]. The downside is that one has to handle possible clashes between unordered rules, this can be done in a verity of ways see for example [4, 5], but the standard way is to use voting. For a thorough in-depth book about rule learning [6] is recommended reading.

Recent interest [7–9] in working with data streams highlight different needs for methods to detect changes and quickly adapt the decision model, the speed of classification is also stressed as an important characteristic. The algorithm presented in this paper address this last issue and might hence prove useful when working with data streams.

In this paper we introduce an indexing method with the aim of improving the classification speed of rule sets while retaining the rules interpretability. One setting which suffers

ICAIR and CACRE '16, July 13 - 15, 2016, Kitakyushu, Japan

ISBN 978-1-4503-4235-3/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2952744.2952750

from long classification times is large ensembles of unordered rule sets. So using this setting we conduct experiments to compare standard way of doing classification with our proposed indexing method. We also look at theoretical properties of our method and analyze how they align with the empirical results.

The paper is organized in the following way: the next section will discuss related work then we will in detail explain the indexing method and reason about its properties; the experimental set-up and the empirical results are then presented and discussed and finally conclusion and pointers to future work is given.

2. RELATED WORK

Spatial databases, i.e. databases that are concerned with objects and their geometrical interrelations, have developed efficient indexing methods for checking whether two object are inside an area of interest etc. One type of such an index is R-tree, in the work by Zhang et al. [8] which is aimed for fast prediction of ensemble models they create their own index method called E-tree that builds upon R-Tree. Another similar type of method is Redundant Bit Vectors (RBVs) presented in [10] which aims at solving the problem of search/matching in high dimensions, in their case they are interested in finding an item in a database that is similar to a query. In contrast to both of these methods our suggested method do not have a separate data structure that needs to be maintained and queried when used, like an index-tree. In our case each rule and example with its conditions and values are assigned a bit vector and are after that "stand-alone" just as in the standard unordered rule setting.

The usage of bit vectors for speeding up classification has been proposed for various machine learning methods. In the work of [11] the authors propose a method for discovery of association rules. The task they look at is mining association rules between items in a database of sales transactions. These association rules describe the relationship of items that are frequently sold together. They present two novel algorithms for calculating support of frequent item sets, which both use bit vectors and their finding is that one of the novel algorithms dominate the other algorithms in their evaluation (including a standard algorithm not using bit vectors) both when considering speed and memory needs. In their approach they use bit vectors for indicating whether or not an a item is present in an item set by setting the bit to 1 for that particular bit that represent that item. Their bit vector representation is similar to our representation on a general level but as they are address a whole other problem the approaches differs in the way the bit vectors are utilized.

There are two major approaches for learning classifier system (LCS) which consist of binary rules that are altered by genetic algorithms for finding the best rules for a particular dataset. In Michigan-style setting one rule set is maintained in contrast to the Pittsburgh-type which maintain a population of separate rule sets, in the former two types of fitness functions are used; either accuracy-based (XCS) or strength based (ZCS). In the paper of [12] which provide methods for "...efficient condition encoding and fast rule matching strategies using vector instructions", which can be seen as an aim that our method indeed also wants to fulfill, the authors experiment with the XCS matching process with and without the use of bit vector indexes. The former performs the same task beyond ninety times faster than the latter. In their paper they do experiments on two different types of processors (CPU:s) with different instructions sets, Altivec and SSE2 respectively and the focus of their work is to speed up the construction of the final rule set and the usage of it, in our setting we are only interested in the latter. In their setting they only consider categorical values in their examples while we on the other hand only consider examples which have numerical inputs. It should be mentioned that our presented algorithm easily can be modified to handle categorical values.

3. INDEXING

The indexing method we propose is not used for creating rules but as a post processing step before the usage of the rules. This transformation step must also be done for unknown examples which will be classified by the indexed rules.

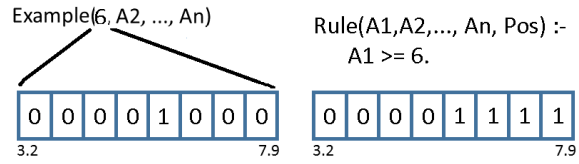
The following example will illustrate the steps of the indexing process; first all training examples that have been used to build the rule set are analyzed and for each attribute its minimum and maximum value are identified. These values then serve as boundaries where it is assumed, using closed world assumption, that the values for the attribute will vary in-between. Note however that these boundaries do not hinder the method to handle values that fall outside this region when for example doing classifying. The method takes as an input parameter the size of the index vector to use, if this is 8 bits and an example has four input variables then the total index vector size for an example in this setting is $8 * 4 = 24$ bits.

Let's assume that the min and max value for $attribute_1$ is 3.2 and 7.9 respectively. These values are arranged so that the left-most bit of the bit vector stands for the value 3.2 and the right-most for 7.9. The difference between the boundaries is 4.7, $7.9 - 3.2 = 4.7$, this value is divided over the 8 bits in a linear fashion, leaving a coverage of $4.7/8 = 0.5875$, for each bit.

The values for each bit in this particular vector size and attribute are (from left to right): $3.2 - 3.7875 - 4.375 - 4.9625 - 5.55 - 6.1375 - 6.725 - 7.3125 - 7.9$. Values for examples are encoded by its corresponding bit vector by inferencing which bit of the bit vector in question that should be set to 1 and the rest of the bits are set to 0. In the left figure of Figure 1 an example and its corresponding bit vector is shown, the first value of the example is 6. This would then result in the following bit vector [0,0,0,0,1,0,0,0], as 6 is between 5.55 and 6.1375.

Finally rules are encoded as bit vectors in similar fashion to the examples, but the difference is that rules do not have point values but intervals. In a setting with numerical attributes the conditions of a rule is either *less than* $<$ or *greater than or equal* $>=$. If a rule has a condition that *if attribute A1 is above or equal* $>=$ *to* 6 *then the class Pos is predicted*. The encoding of such a rule will correspond to the rule and bit vector shown in the right most part of the Figure 1.

The algorithm for indexing examples is described in more detail in Algorithm 1, for indexing rules the algorithm is similar but with small differences in the inner for-loop, when indexing examples only one bit is set to 1 and the rest to 0, when creating an index for rules more than one bit can be assigned the value 1. Input to the algorithm is the example



(a) Corresponding bit vector for the first attribute of an example

(b) Corresponding bit vector for a rule with one condition

Figure 1: Example of bit vectors

that is to be indexed, the size of the bit vector and a list of the minimum- and maximum-value of each attribute. Returned from the algorithm is a bit vector corresponding to the example.

The algorithm iterates over all attributes, and for each attribute it uses the function **getPrec** which uses the bit size and the attributes minimum and maximum values list to initialize the precision and set the minimum and maximum values for the particular attribute. A left and right bound are set, these act as a sliding window if the examples value fall in-between of these values, then that particular bit should be set to 1. An inner-loop which iterates as many times as the bit size uses the left- and right-bound to add bits, 1 or 0, to the bit vector. When the inner-loop is finished the final bit vector is added to the indexed example, the outer-loop is continued until all attributes has been covered, and finally the indexed example is returned.

Algorithm 1 Indexing example

Inputs: $Example, BitS, AttsMinMax$
Outputs: $IndexedExample$

$Atts \leftarrow$ all attributes in Ex
for $a_i \in Atts$ **do**
  $Prec_i, Min_i, Max_i \leftarrow \text{getPrec}(BitS, AttMinMax, a_i)$
  $LeftBound = Min_i$
  $RightBound = Min_i + Prec_i$
  $BitVector = 0$
  for each *bit* to $BitS$ **do**
    if $LeftBound =< a_i \bigwedge RightBound > a_i$ **then**
      $BitVector \leftarrow BitVector + 1$
    else
      $BitVector \leftarrow BitVector + 0$
    end if
    $BitVector << 1$ ▷ Shift one position
    $LeftBound \leftarrow RightBound$
    $RightBound \leftarrow RightBound + Prec_i$
  end for
  $IndexedExample \leftarrow IndexedExample \cup BitVector$
end for
return $IndexedExample$

Before using a rule set for prediction purposes all its rules must be converted to index format. When predicting an unknown example it also must converted to an index format as described above. Then each rule is tested if its conditions are fulfilled by using the rules bit vector and the example bit vector, $BitVecRule \wedge BitVecEx$ followed by the *popcount* function which returns the number bits that are set to one. This number is then checked with the number of conditions that the rule has, if they are equal all conditions of the rule has been fulfilled and the rule are used for predicting the class of the unknown example.

3.1 Expected benefits from using the indexing method

How much faster will a prediction utilizing the indexing method be compared to using ordinary classification? In the next section we will present the experimental setup and the empirical results, but before this we will reason about the expected performance gains. Using indexing each rule is classified by two functions and one comparison, $\oplus$, *popcount* and $=$. So the time for classification is constant no matter the number of conditions in a rule, the only parameter that affects the speed of classification in the index case is the size of the bit vector used, as this affect the time of the two functions to calculate a result. The size of the bit vector will also influence of how well the method can represent the original rules and examples. If the size is low, the coarser the index view, which would probably result in a representation which might not capture the rules and examples adequately. Having made these observations one can make the following predictions about the effectiveness of indexing compared to the standard method:

- A high number of conditions per rule is probably advantageous for our indexing method compared to the standard method
- A small bit vector size will probably result in low prediction times, but not necessarily correct predictions

In the experimental section that follows we will adjust the latter as a parameter and measure the former for each dataset, to see if our intuition holds empirically.

4. EXPERIMENT

The two prediction methods, indexing and the standard method, was evaluated in a 10-fold cross validation scheme. The ensemble size was set to 1000, i.e. we induce 1000 rule sets. The ensemble strategy was that of bagging [13] thus creating a new training set for each of the 1000 rule sets by sampling examples with replacement.

Two datasets contained missing values in the case of breast cancer Wisconsin 16 missing values was replaced by the value 1, in the case of Cleveland heart disease all missing values was replace by 0.0. Different sizes of bit vectors was used for the indexing method, ranging from: 4, 8, 16, 32, 64, 128 to 256 bits.

From the experiments the average accuracy and average classification time (in milliseconds) over the 10 folds was

Table 1: Average accuracy

Dataset	No. rules	No. cond	Ensemble	i-4 bits	i-16 bits	i-64 bits	i-128 bits	i-256 bits
Br. c. wisc.	8682.0	10686.9	95.9, 17066	95.1, 9703	96.0, 10313	95.9, 10486	95.9, 11029	93.4, 11684
Bupa	22127.5	59770.8	73.3, 17175	58.3, 10764	65.2, 9936	73.0, 9688	74.8, 9922	72.5, 10547
Clev. h. dis.	29868.1	83939.4	56.3, 23431	57.3, 11560	53.3, 12356	58.7, 11653	59.3, 12448	59.7, 13868
Glass	4464.5	3578.3	92.7, 1701	93.6, 906	92.7, 1017	91.8, 1091	91.8, 1186	92.7, 1293
Haberman	29564.3	75417.4	69.3, 18284	72.9, 10653	72.2, 9782	68.6, 10734	68.6, 10813	68.3, 11029
Image seg.	10840.8	18543.7	81.3, 9594	37.0, 6192	62.7, 5475	77.5, 5445	80.9, 5852	80.4, 6660
Ionosphere	7916.7	11875.4	90.1, 8190	86.9, 4603	88.9, 4929	91.4, 5400	90.6, 6316	90.6, 8625
Iris	4497.5	4374.1	93.3, 2417	55.3, 1374	91.3, 1467	94.0, 1543	92.7, 1576	92.7, 1622
Pen digits	58224.0	205478.2	96.2, 1205513	82.6, 230180	96.0, 237948	83.0, 332405	83.2, 337727	83.2, 365323
Pima ind.	32456.8	101371.5	74.6, 44540	74.5, 3805	75.9, 6367	76.7, 8751	75.9, 9268	75.4, 9344
Sonar	7423.5	11396.9	82.7, 4461	74.5, 3057	80.8, 2978	84.1, 3635	83.7, 4851	84.1, 6926
Spec. flare	10179.6	18911.4	85.7, 9874	51.9, 5758	82.5, 6067	84.2, 7005	86.0, 8720	86.2, 12106
Thyroid	5634.0	6593.3	94.4, 4414	70.2, 2731	80.5, 2776	91.2, 2636	92.56, 2649	94.4, 2839
Wine	4760.8	5051.8	96.1, 3167	93.3, 1734	95.5, 1903	94.9, 2027	95.5, 2215	96.1, 2511
Fr. rank acc.	-	-	2.79	5.07	3.93	3.25	2.96	3.0
P value	0.01015							
Fr. rank time	-	-	1.29	5.21	4.86	4.50	3.29	1.86
P value	2.74E-10							

measured. The total number of conditions and total number of rules generated was also collected for each dataset. In total 15 datasets was used in the experiment, all taken from the University of California Irvine (UCI) Machine Learning Repository [14]. The 15 datasets ranged from a size of 150 to 7494 instances and the number of attributes ranged from 3 to 34 attributes excluding the class label. The algorithms was implemented with SWI-Prolog, and are available to downloaded from http://dsv.su.se/∼tony/programs.html together with the data sets.

4.1 Experimental results

Overall the results from the experiment follow the expected pattern. Typically the accuracy and time do increase with larger index sizes, as expected. In Table 1 the measured values are presented, the first column contains the names of the datasets, the second column contains the averaged total number of rules over the ten folds, the thirds column present the averaged total number of conditions over the ten folds, the fourth column displays the average accuracy and classification time (in milliseconds) for the original ensemble in the following columns the accuracies and classification times for index sizes 4 to 256 is shown. Note that index size 8 and 32 has been omitted in the table due to lack of space. Also note that the average total number of conditions is lower than the averaged total number of rules in the Glass and Iris dataset. This odd observation has a perfectly good explanation, if the rule set contains only a few rules and each rule only utilizes a few conditions then this causes the averaged total number of conditions to become lower than the average total number of rules, as when inducing each rule set we usually end up with a rule with no conditions, which is the last induced rule which in turn covers the last examples (5 examples in our case).

Following the significance test procedure of [15][1], we first apply the Friedman test, the ranking is shown in Table 1 in the row with *Fr. rank acc* in the first column. Here we see how the accuracy is ranked, not surprisingly the ensemble is ranked best then comes index-128, index-256, index-64, index-16 and index-4. The p-value of the test is: 0.001015, which indicate that we indeed have statistical significant differences between the methods.

We then rank the methods with regard to time, in the row below, here the lower value the faster classification, the ranking is as follows, index-4, index-16, index-64, index-128, index-256 and the ensemble. This is as expected as the time needed for classifying with the ensemble and index-256 is followed by shorter index values in the expected order. This ranking has a p-value of: 2.74E-10 which shows that we have a statistical significant difference between the methods. Using Bergmann's procedure we can for the accuracies reject the null hypotheses in the following cases:

- Ensemble vs. index-4
- index-4 vs. index-128
- index-4 vs. index-256

Which indicate that the accuracy for the index-4 method performs statistically worse than the ensemble and index-128 and index-256.

Using the same test but focusing on classifying time we can reject the null hypothesis in the following cases:

- Ensemble vs. index-4
- Ensemble vs. index-16
- Ensemble vs. index-64
- Ensemble vs. index-128
- index-4 vs. index-128
- index-4 vs. index-256
- index-16 vs. index-256
- index-64 vs. index-256

Hence all indexing methods except index-256 are statistically significantly faster than the ensemble method. Using this knowledge together with the statistical significance test of accuracies we can conclude that the best choice of method would be any indexing method with bit sizes from

[1]The statistical test are performed with the software that can be found at: http://sci2s.ugr.es/keel/multipleTest.zip.

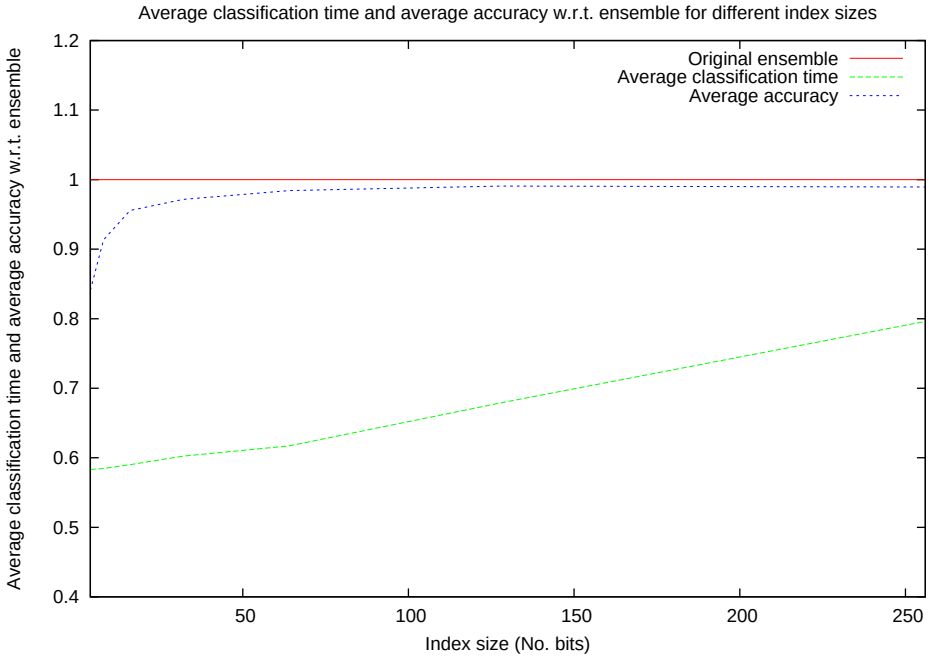


Figure 2: Average classification time and average accuracy w.r.t. ensemble for different index sizes

16 to 128, then you get a classifier that is statistically significantly faster while they cannot be separated statistically w.r.t. classification performance measured by accuracy in this case. Using the Friedman rank of accuracy, index-128 is the best choice.

In Figure 2 we have used the ensemble to normalize the index values against both in terms of accuracy and time, i.e. a value below 1 would mean lower performance than the ensemble for accuracy, for classification time the same value would mean a faster classification. For each index size the average normalized values of accuracies and classification times w.r.t. the ensemble is plotted. From the figure it is clear that all index methods perform below the ensemble in terms of accuracy. Indexes of size 4 to 64 have a much lower accuracy than the ensemble, with larger index sizes than 64 bits the accuracies stabilizes around 0.99 (in comparison to the ensemble accuracy), until maximum index size of 256. The indexing classification time varies from just below 0.6 to 0.8 compared to the ensemble classification time and there is strong correlation with larger index size and increased classification time. The relationship looks almost linear, which might indicate that the functions $\oplus$ and *popcount* has a linear complexity.

To investigate if there is a correlation between number of conditions per rule and gain in indexing classification time w.r.t. ensemble classification time (hence the classifications times normalized by the ensemble), we plot the average classification time (for all indexing methods) on the Y-axis and the number conditions divided by the number rules on the X-axis in Figure 3.

We expect to see a negative slope in the plot. From the figure we see that the overall trend of the plotted data do support this hypothesis, but there are quite a few "bumps" in the figure. So from this figure the we can indeed draw the conclusion that our hypothesis is still valid.

5. DISCUSSION, CONCLUSIONS AND FUTURE WORK

Unordered rule sets have traits that are to their advantage comparing with ordered rule sets, as better interpretability and higher prediction accuracy, but one downside is that unordered rule sets have a higher time complexity when classifying examples. This is becuse each rule in a rule set must be tested when classifying one example. In this paper we suggest an indexing method for speeding up classification when using unordered rule sets.

The classifying accuracies for our indexing method were on average slightly lower than that of the original ensemble, 0.991 accuracy compared to that of the ensemble, using the indexing method with 128 bits which was the best performing indexing method size. This method has classifying time which is 0.679 than that of the ensemble, so if time is more important than the final accuracy our proposed method is right for the job.

As the indexing size do have a big impact on both the classification times and predictive performance when considering using this technique this is something to take into consideration. For practical purposes manually tuning the correct index size to the problem at hand is advised. To automate this task could be something to investigate in the future.

Another issue that could well be worth investigating in the future is how to select thresholds between the bits of a bit vector. The linear method presented in this paper could probably be improved upon. One possible alternative solution would be to select the boundaries between bits of a bit vector by inspecting the densities of certain regions. Dense regions would probably benefit from having more bits than sparse regions. This basic observation could be further elaborated if one also takes class labels into account.

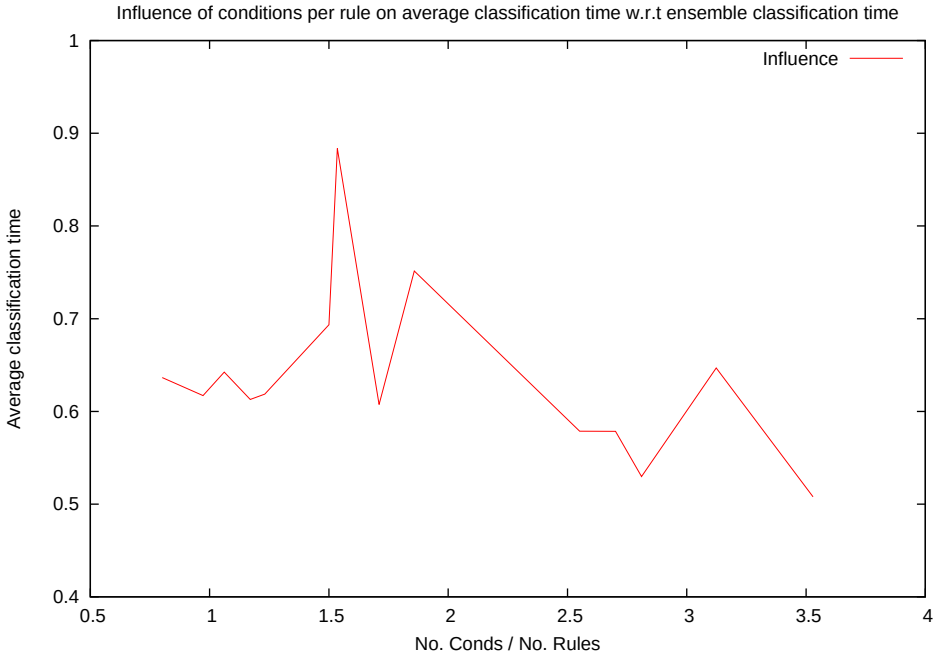


Figure 3: Influence of No. Conditions / No. Rules on average classification time

Acknowledgments

This work has been funded by Scania CV AB and the Vinnova program for Strategic Vehicle Research and Innovation (FFI)-Transport Efficiency.

6. REFERENCES

[1] Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[2] Peter Clark and Robin Boswell. Rule induction with CN2: some recent improvements. In *Machine Learning - EWSL-91, European Working Session on Learning, Porto, Portugal, March 6-8, 1991, Proceedings*, pages 151–163, 1991.

[3] Henrik Boström. Maximizing the area under the ROC curve with decision lists and rule sets. In *Proceedings of the Seventh SIAM International Conference on Data Mining*, pages 27–34, 2007.

[4] Tony Lindgren and Henrik Boström. Resolving rule conflicts with double induction. *Intell. Data Anal.*, 8(5):457–468, 2004.

[5] Tony Lindgren. On handling conflicts between rules with numerical features. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 37–41, 2006.

[6] Johannes Fürnkranz, Dragan Gamberger, and Nada Lavrac. *Foundations of Rule Learning*. Cognitive Technologies. Springer, 2012.

[7] João Gama and Petr Kosina. Learning decision rules from data streams. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 1255–1260, 2011.

[8] Peng Zhang, Jun Li, Peng Wang, Byron J. Gao, Xingquan Zhu, and Li Guo. Enabling fast prediction for ensemble models on data streams. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 177–185, New York, NY, USA, 2011. ACM.

[9] Petr Kosina and João Gama. Very fast decision rules for classification in data streams. *Data Min. Knowl. Discov.*, 29(1):168–202, 2015.

[10] Jonathan Goldstein, John C. Platt, and Christopher J. C. Burges. Redundant bit vectors for quickly searching high-dimensional regions. In Joab Winkler, Mahesan Niranjan, and Neil D. Lawrence, editors, *Deterministic and Statistical Methods in Machine Learning*, volume 3635 of *Lecture Notes in Computer Science*, pages 137–158. Springer, 2004.

[11] Georges Gardarin, Philippe Pucheral, and Fei Wu. Bitmap based algorithms for mining association rules. In *14ème Journées Bases de Données Avancées, 26-30 octobre 1998, Hammamet, Tunisie (Informal Proceedings).*, 1998.

[12] Xavier Llorà and Kumara Sastry. Fast rule matching for learning classifier systems via vector instructions. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1513–1520, New York, NY, USA, 2006. ACM.

[13] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, August 1996.

[14] M. Lichman. UCI machine learning repository, 2013.

[15] Salvador García and Francisco Herrera. An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons. *Journal of Machine Learning Research*, 9:2677–2694, 2008.