Using Ensembles of Clauses to quickly obtain Simpler and Effective Classifiers

No Author Given

No Institute Given

Abstract. Most ILP systems use a greedy covering algorithm to find a set of clauses that best model positive examples. This set of clauses is called a theory and can be seen as an *ensemble* of clauses. It turns out that the search for a theory within the ILP system is very time consuming and often yields overly complex classifiers. One alternative approach to obtain a theory is to use the ILP system to non deterministically learn one clause at a time, several times, and to combine the obtained clauses using ensemble methods. This can be a much faster approach, as this work shows, and produces better classifiers than the theories produced with greedy covering algorithms.

Keywords: ensembles, bagging, boosting, Aleph

1 Introduction

Inductive Logic Programming (ILP) systems have been quite successful in extracting comprehensible models of relational data. Indeed, for over a decade, ILP systems have been used to construct predictive models for data drawn from diverse domains. These include the sciences,engineering, language processing, environment monitoring, software analysis, and pattern learning and link discovery. Most ILP systems use a greedy covering algorithm that repeatedly examines candidate clauses (the "search space") to find good rules (or theories). Ideally, the search will stop when the rules cover nearly all positive examples with only a few negative examples being covered.

This algorithm poses some challenges, since the search space can grow very quickly, sometimes turning unfeasible the search for a good solution. Several techniques have been proposed to improve search efficiency of ILP algorithms. Such techniques include improving computation times at individual nodes [3, 19], better representations of the search [2], sampling the search space [20, 21, 24], parallelism [6, 11, 16, 23, 24, 8], and utilisation of ensemble methods [9].

Ensembles are classifiers that combine the predictions of multiple classifiers to produce a single prediction [7]. Several researchers have been interested in the use of ensemble-based techniques for ILP. To our knowledge, the original work in this area is Quinlan's work on the use of boosting in FOIL [18]. His results suggested that boosting could be beneficial for first-order induction. More recently, Hoche proposed confidence-rated boosting for ILP with good results [13]. Zemke proposed bagging as a method for combining ILP classifiers with other classifiers [25]. Dutra *et al* [9] studied bagging in

the context of the ILP system Aleph [22] learning theories. Their results showed that the applications benefitted from ensembles to a limited extent.

In this work we argue that learning **a single clause at a time** rather than learning **whole theories** (or sets of clauses) at a time can be more cost-effective and produce simpler classifiers. Our alternative approach then is to use *ensembles* of **clauses**. To some extent, an induced theory is an ensemble of clauses. However, finding an induced theory is very time consuming and can produce very complex classifiers. In this work we learn single clauses, and use ensemble methods to combine them, which produce classifiers that are better than any single clause or theory, in much less time than finding a theory.

The paper is organised as follows. First, we present in more detail the ensemble methods used in this work. Next, we discuss our experimental setup and the applications used in our study. We then discuss how ensembles of clauses compare with ensembles of theories. Last, we offer our conclusions and suggest future work.

2 Ensemble Methods

In general, ensemble methods work by combining the predictions of several (hopefully different) weak classifiers to produce one final strong classifier.

Ensemble generation assumes two distinct phases: (1) training, and (2) combining the classifiers. The ensemble methods vary according to the constraints imposed to the training phase, and to the kind of combination used.

Figure 1 shows the structure of an ensemble of logic programs. This structure can also be used for classifiers other than logic programs. In the figure, each program $P_1, P_2, \ldots, P_N$ is trained using a set of training instances. At classification time each program receives the same input and executes on it independently. The outputs of each program are then combined and an output classification reached. Figure 1 illustrates that in order to obtain good classifiers one must address three different problems:

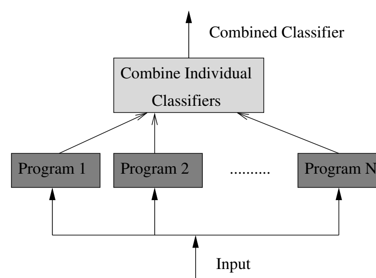


Fig. 1. An Ensemble of Classifiers.

- how to generate the individual programs;
- how many individual programs to generate;

– how to combine their outputs.

Regarding the first problem, research has demonstrated that a good ensemble is one where the individual classifiers are accurate and make their errors in different parts of the instance space [14, 17]. Obviously, the output of several classifiers is useful only if there is disagreement between them. Hansen and Salamon [12] proved that if the average error rate is below 50% and if the component classifiers are independent, the combined error rate can be reduced to 0 as the number of classifiers goes to infinity. In this work, we argue that ensembles of clauses produce rather independent classifiers than ensembles of theories as studied by Dutra *et al* [9].

Methods for creating the individual classifiers therefore focus on producing classifiers with some degree of diversity. In the present work, we follow two approaches to produce such classifiers. We produce clauses and theories (sets of clauses). We believe that clauses have a greater degree of diversity than theories.

The second issue we had to address was the choice of how many individual classifiers to combine. Previous research has shown that most of the reduction in error for ensemble methods occurs with the first few additional classifiers [12]. Larger ensemble sizes have been proposed for decision trees, where gains have been seen up to 25 classifiers.

The last problem concerns the combination algorithm. An effective combining scheme is often to simply average the predictions of the network [1, 5, 14, 15]. An alternate approach relies on a pre-defined *voting threshold*. If the number of clauses or theories that cover an example is above or equal to the threshold, we say that the example is positive, otherwise the example is negative. Thresholds may range from 1 to the ensemble size. A voting threshold of 1 corresponds to a classifier that is the disjunction of all theories. A voting threshold equal to the ensemble size corresponds to a classifier that is the conjunction of all theories. This voting scheme was used in [9].

Individual classifiers that compose an ensemble can be obtained from different samples of the dataset or from the same dataset. They can also be obtained from one single ILP algorithm (homogeneous classifiers) or from different ILP algorithms (heterogeneous algorithms). In this work we use homogeneous classifiers and the two kinds of datasets: same dataset and different datasets. The classifiers can be independent or dependent, depending on the ensemble method employed.

Several methods have been presented for ensemble generation. The two most popular are *bagging* and *boosting* [10]. Bagging works by training each data set on a random sample from the training set. Classifiers in this case are totally independent from each other. Boosting works by assigning penalties to misclassified examples, and refining the search for clauses that try to cover the misclassified examples. Therefore, in boosting, the classifiers are dependent. The current classifier is dependent on the previous in the training sequence.

We evaluate bagging and two variations of boosting when producing ensembles of clauses. We also use an ensemble method we name *different-seeds* (also used in [9]), which takes advantage of the non-determinism in seed-based search by simply combining different clauses obtained from experimenting with different seed examples, while always using the original training set. We then contrast the results obtained with clause-based learning to the results obtained with theory-based learning, using *different-seeds*-

theories. It is important to note that we implemented the ensemble methods without modifying the ILP system. We take advantage of non deterministic seed selection to implement all ensemble methods.

In this work, we implemented two kinds of boosting, one based on arcing [4] and the other one based on AdaBoost [10] by selecting all examples for all training sets Arcing chooses a training set of size N for classifier K+1 by probabilistically selecting (with replacement) examples from the original N training examples. Unlike bagging, however, the probability of selecting an example is not equal across the training set. This probability depends on how often that example was misclassified by the previous K classifiers. Arcing uses a simple mechanism for determining the probabilities of including examples in the training set. For the *i*th example in the training set, the value m_i refers to the number of times that example was misclassified by the previous K classifiers. The probability p_i for selecting example i to be part of classifier K+1 's training set is defined as

$$p_i = \frac{1 + m_i^4}{\sum_{j=1}^{N}(1 + m_j^4)} \tag{1}$$

Breiman chose the value of the power (4) empirically after trying several different values [4]. This is also known in the literature as arcing-x4.

AdaBoost can use the approach of (a) selecting a set of examples based on the probabilities of the examples, or (b) simply using all of the examples and weight the error of each example by the probability for that example (i.e., examples with higher probabilities have more effect on the error). We chose to use all the examples and weight the error according to equation 1.

3 Methodology

We studied 5 different ensemble methods:

1. *pure-boosting*: based on AdaBoost, works by selecting all examples every iteration. Each training set is the same at each training/test iteration.
2. *hybrid-boosting*[1]: selects examples with replacement as the arcing method.
3. *bagging*: uses different training sets taken from the original dataset with replacement.
4. *different-seeds-clauses*: use the same training set at each iteration and takes advantage of the seed-based algorithm of the ILP system to generate varying clauses.
5. *different-seeds-theories*: the same as different-seeds-clauses, but the training phase generates theories instead of clauses.

We could have chosen bagging or different-seeds to generate theories. We included only different-seeds because all runs for theories take a huge amount of time to finish. [2]

[1] we named it like that because the selection of examples for the next training set is similar to bagging

[2] [9] shows some results for one of our benchmarks using bagging of theories.

Both implementations of boosting compute new probabilities using the probability function used by arcing.

Figure 3 shows an overview of our general algorithm to perform boosting without making modifications to the ILP system. The syntax used in the algorithm is Prolog-like. Arguments with a plus sign are input, and with a minus sign are output.

```
boosting(+ILPOut,+Exs,+Errors,+Weights):-
    1. read_ILP_output(+ILPOut),
    2. read_examples(+Exs),
    3. calc_errors(+Errors,-Sum_pos,-Sum_neg,-P,-N),
    4. write_new_errors(+Errors,+Sum_pos,+Sum_neg),
    5. calc_weights(+Sum_pos,+Sum_neg,-Weigth_P, -Weights_N),
    6. write_new_weights(+Weigths,+Weights_P,+Weights_N),
    7. find_min_weight(+Weights_P, +Weights_N,-Min_P,-Min_N),
    8. generate_new_exs(+P,+Weights_P,+Min_P,-Exs_P,
                        +N,+Weights_N,+Min_N,-Exs_N),
    9. write_new_examples(+Exs,+Exs_P,+Exs_N).
```

Fig. 2. General boosting algorithm

This algorithm is used for both *pure-boosting* and *hybrid-boosting*. *Pure-boosting* does not execute steps 7 to 9, because the training set is not modified from one iteration to the next. The algorithm uses four files: `ILPOut` is an output generated by a filter, after the learning finishes. It basically, collects the clause that was generated by the ILP system, with a map of the coverage for that clause. This is necessary, because later we need to compute new probabilities to the examples. `Exs` is the prefix file name for the examples. `Errors` contains the errors obtained with the previous iteration (this is needed to use the arcing-x4 formula to compute probabilities). `Weights` contains the previous weights of the examples classified by the previous clause.

Steps 1 and 2 read in, respectively, the file that contains the positive and negative examples covered by the current classifier, and the file containing the whole set of examples. Step 3 calculates the errors using the previous `Errors` file, and produces new errors related to the positive (`Sum_pos`) and negative (`Sum_Neg`) examples. It also returns the total number of positive (`P`) and negative (`N`) examples. Step 4 writes the new errors to disk. Steps 5 and 6 calculate and write new weights to disk, using the arcing-x4 formula. Steps 7 to 9, used only by our hybrid-boosting implementation apply a sampling with replacement to the previous data set to generate a new training set based on the errors of the current classifier.

Bagging was implemented starightfowardly. We only needed to generate the bags and run the training/test experiments independently.

Aleph tries to find one hypothesis H in a description language $\mathcal{L}$, such that: (1) H respects the constraints I; (2) The positive examples, E^+, are derivable from B, H, where B is the background knowledge, and (3) The negative examples, E^-, are not derivable from B, H. By default, Aleph uses a simple greedy set cover procedure that constructs such a hypothesis one clause at a time. The final classifier is a collection

of clauses (a theory). In the search for any single clause, Aleph randomly selects an uncovered positive example as the seed example, saturates this example, and performs an admissible search over the space of clauses that subsume this saturation, subject to a user-specified clause length bound. As it was mentioned before this is a very time-consuming process. We contrast this approach with the approach of generating one single clause and stopping, using a randomly chosen example as a seed. In the case of boosting, the example with higher probability is chosen as a seed. If there are many examples with the same high probability, we random select one of them.

These steps are done through a script that runs outside the Aleph code. We wrote our own Prolog code to randomly select a seed, based on the examples weights.

In the case of pure-boosting, we simply compute the errors produced in one iteration, and, in the next iteration, select the example that has higher probability weight to start a search for a new clause. The training set is **not** modified. In the case of hybrid boosting, a new training set is created from one iteration to another, as in the conventional arcing algorithm.

A possible run of our hybrid-boosting implementation is as follows. Assume a set of positive examples $\{$A,B,C,D,E$\}$, and suppose an iteration that calculated probabilities $\{$.25, .25, .3, .1, .1$\}$. We look for the examples with higher probability (in this case, it is example C, with probability .3), and decrement its weight from the least weight (.1). In this example, the weight of example C (.3) will be decreased by .1 becoming .2. C is then, the first example chosen to be in the next training set. At this time we have a new weight list $\{$.25, .25, .2, .1, .1$\}$. Now A or B are candidate examples to be included in the training set. We select one of them randomly. The algorithm proceeds like that until the new training set is complete.

We have elected to perform a detailed study on five datasets, corresponding to five non-trivial ILP applications. For each application we ran Aleph with random re-ordering of the positive examples and hence of seeds. We call this experiment *different-seeds*. Next, we created bagged training sets from the original set, and called Aleph once for each training set. We call this experiment *bagging*. The hybrid boosting experiment creates a new training set at each iteration. The number of runs of *different-seeds* and boosting is the same as the number of bags.

Aleph allows the user to set a number of parameters. We always set the following parameters as follows:

– search strategy: `search`. We set it to be breadth-first search, `bf`. This enumerates shorter clauses before longer ones. At a given clauselength, clauses are re-ordered based on their evaluation. This is the Aleph default strategy that favours shorter clauses to avoid the complexity of refining larger clauses.
– evaluation function: `evalfn`. We set this to be coverage. Clause utility is measured as $P - N$, with P and N being the number of positive and negative examples covered by the clause, respectively.
– chaining of variables: `i`. This Aleph parameter controls variable chaining during saturation: chaining depth of a variable that appears for the first time in a literal $\mathcal{L}_i$, is 1 plus the maximum chaining depth of all variables that appear in previous literals $\mathcal{L}_j, j < i$. We used a value of 5 instead of the default value of 2 in order to obtain more complex relations between literals in a clause.

– max number of nodes allowed: `maxnodes`. This corresponds to the number of clauses in the search space. We set this to be 100,000.
– maximum number of literals in a clause: `maxclauselength`. This was chosen based on some previous experiments and was set to 5.
– minimum clause accuracy: this is the minimum accuracy acceptable when generating a new clause. It was set to 0.9 (i.e., accuracy of 90%).

We used the same set of parameters used by Dutra *et al* [9] in order to compare some results.

Next, we organise our discussion of methodology into experimentation and evaluation.

Experimentation. Our experimental methodology employs five-fold cross-validation. For each fold, we consider ensembles with size varying from 1 to 25. Thus each application ran 25 times. This step generates 25 files with one clause per file in the case of all experiments that learn clauses, and 25 files with one theory per file (set of clauses), in the case of the different-seeds experiment.

Evaluation. For the evaluation phase, we used two techniques to evaluate the quality of the ensembles. First, we studied how average accuracy varies with ensemble size. We present accuracy as $\frac{T_p+T_n}{Total_of_exs}$, where T_p and T_n, are respectively, the number of positive and negative examples, and $Total_of_exs$ is the dataset size.

Second, we studied Receiver Operating Characteristic (ROC) curves for the ensembles. Each point on the ROC plot represents a sensitivity/specificity pair corresponding to a particular decision threshold. A test with perfect discrimination (no overlap in the two distributions) has a ROC plot that passes through the upper left corner (100% sensitivity, 100% specificity). Therefore the closer the ROC plot is to the upper left corner, the higher the overall accuracy of the test [26].

When we have many ROC curves to be analysed, we can look instead to the area under those curves. The value for the area under the ROC curve can be interpreted as follows: an area of 0.84, for example, means that a randomly selected individual from the positive group has a test value larger than that for a randomly chosen individual from the negative group 84% of the time. When the variable under study cannot distinguish between the two groups, i.e. where there is no difference between the two distributions, the area will be equal to 0.5 (as is the case when the ROC curve coincides with the diagonal). When there is a perfect separation of the values of the two groups, i.e. there is no overlapping of the distributions, the area under the ROC curve equals 1 (the ROC curve will reach the upper left corner of the plot).

We wish to test the effectiveness of different sizes of ensembles, from 1 to 25. Again, we do not repeat the ILP runs themselves to learn entirely new theories for each different ensemble size. Rather, we use the theories from the previous step. Because our results may be distorted by a particularly poor or good choice of these theories, we repeat this selection process 7 times (30% of the total number of ensembles) and average the results.

Figure 3 shows the general algorithm to perform the evaluation step. The loop that goes from line 2 to 8 computes the points for the ROC curves using the pair `(Fp,Tp)`, where `Fp` is the rate of false positives and `Tp` is the rate of true positives. This is done

for 7 sets, where each set contains `ensSize` clauses (or theories) that are selected randomly from the 25. The loop that goes from line 9 to line 14 computes the average of the 7 sets (across the false positives and true positives).

```
1. for ensSize = 1 to 25 do
2.    randomly select 7 sets of size ensSize;
3.    for s = 1 to 7 do
4.         sum[s] = 0
5.         for v = 1 to ensSize do
6.              pointROC[s,v]= pair (Fp,Tp)
7.         endfor
8.    endfor
9.    for v = 1 to ensSize do
10.        for s = 1 to 7 do
11.             sum[s] = sum[s] + pointROC[s,v]
12.        endfor
13.        avgROC[v] = sum[v]/7
14.   endfor
15.   accuracy[ensSize] = better value of avgROC
16.   areaROC[ensSize] = area under avgROC
17.endfor
```

Fig. 3. General algorithm for evaluation step.

Accuracies across the folds are obtained by averaging the sum of all positives and negatives for every fold. Areas across the folds are obtained by simply averaging areas computed for each ensemble size. ROC curves across the folds are obtained by averaging the rates of positives and negatives of each fold.

All experiments were performed using Yap Prolog 4.4.2, and Aleph 3.0. We used three machines to run all experiments: (1) Intel Pentium III 750 MHz, with 256 MBytes of RAM, with Red Hat Linux 7.2, (2) Intel Pentium III 600 MHz, with 256 MBytes of RAM, with Red Hat Linux 9.0, and (3) Intel Pentium IV 1.2 GHz, with 1 GByte of RAM. Although these machines have different characteristics, all experiments for the same application were performed in the same machine. Slower runs were launched in the more powerful machine.

3.1 Benchmark Datasets

Our benchmark set is composed of five datasets that correspond to five non-trivial ILP applications that are also used in other works. We next describe the characteristics of each dataset with its associated ILP application, and present a dataset summary table. The datasets were taken from `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/applications.html`

Amine. Our first learning task is to predict amine re-uptake inhibition to discover new drugs for the Alzheimer disease.

Carcinogenesis. Our second application concerns the prediction of carcinogenicity test outcomes on rodents. This application has a number of attractive features: it is an important practical problem; the background knowledge consists of large numbers of non-determinate predicate definitions; previous experience suggests that a fairly large search space needs to be examined to obtain a good clause.

Choline. This application is also related to drug discovery for the Alzheimer disease. The learning task is to identify the inhibition of the aceto-choline-esterase enzyme.

Mutagenesis. The prediction of mutagenesis is important as it is relevant to the understanding and prediction of carcinogenesis. Not all compounds can be empirically tested for mutagenesis, e.g. antibiotics. The considered dataset has been collected with the intention to search for a method for predicting the mutagenicity of aromatic and heteroaromatic nitro compounds.

Protein. Our last dataset consists of a database of genes and features of the genes or of the proteins for which they code, together with information about which proteins interact with one another and correlations among gene expression patterns. This dataset is taken from the function prediction task of KDD Cup 2001. While the KDD Cup task involved 14 different protein functions, our learning task focuses on the challenging function of "metabolism": predicting which genes code for proteins involved in metabolism. This is not a trivial task for our ILP system.

Table 1. Datasets Characteristics.

Dataset	E+	E-	BK Size
Amine	343	343	232
Carcinogenesis	182	148	24,988
Choline	663	663	232
Mutagenesis	114	57	15,113
Protein	172	690	6,913

Table 1 summarises the main characteristics of each application. The second and third columns correspond to the size of the full datasets. Bags are created by randomly picking elements, with replacement, from the full dataset. Therefore the number of positives or negatives of each bag are not the same as of the full dataset used for *different-seeds*, although the total size is the same. The last column indicates the size of the background knowledge (number of facts and rules). The test sets for each 5-fold cross-validation experiment is obtained by a round-robin distribution of the full dataset. For example, application Carcinogenesis will have 5 positive test sets of sizes: 37, 37, 36, 36 and 36.

4 Results

This section presents our results and analyses the performance of each application. We divided this section into two subsections. The first one presents a qualitative analysis, where for each application we show the average accuracy for positives and negatives, and the area under ROC curves built from 1 to 25 ensemble sizes. In the second subsection, we discuss about execution time and size of the final ensemble, providing a quantitative analysis. We report results for *different-seeds-theories*, *different-seeds*, *bagging*, *pure-boosting* and *hybrid-boosting*. We also show the ROC curve for an ensemble size of 25.

Accuracies presented in the graphs are averaged across all folds, and for each ensemble size, a different voting threshold is used. The voting threshold is the one that produced the best accuracy. For clarity's sake, these parameter values are not shown in the curves.

4.1 Qualitative Analysis

Figure 4 shows the average accuracies for all applications with all ensemble methods.

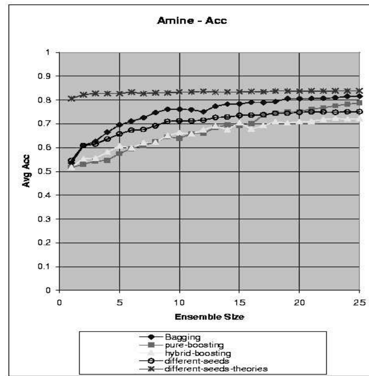
Figure 4(a) shows the accuracies obtained by the five ensemble methods for the application Amine. In this figure all methods based on clause learning produce good accuracies between 70% and 83% for ensemble sizes greater than 15. Different-seeds-theories achieves accuracies above 80% even with ensemble of size 1. Each point of the curve can have a different voting threshold.

The first ppoint to observe in this application is that theory-based learning does not benefit much from ensembles keeping an average accuracy of 83% across all ensemble sizes starting from 2. In contrast, the methods based on ensembles of clauses have a clear improvement as the ensemble sizes increase. Bagging is the method that achieves better performance of all, almost reaching the best performance of different-seeds-theories. Pure-boosting improves linearly when we increase the size of the ensemble. The performance of hybrid-boosting also increases linearly achieving better accuracy values than pure-boosting up to ensemble of size 15. After that, accuracies still improve but not in a so good rate as pure-boosting. This happens because, this method generates very specific clauses to the misclassified examples that get repeated in the training set. Figure 5 shows an example of this situation happening in fold 1. This figure shows the accuracy obtained for each clause generated at each iteration for the training set and for the test set. We can observe that after iteration 19, both accuracies (train and test) start to fall apart. Clauses learned after iteration 19 become too specific for the training set, and therefore fail to cover examples of the test set.

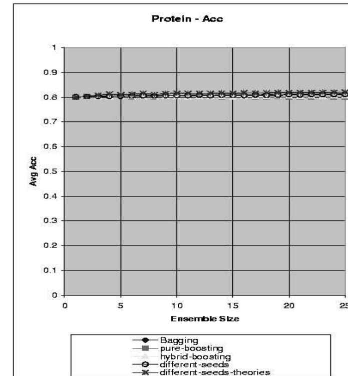
One important thing to note at this point is that, for this application, each individual theory has an average size of 70 clauses. When we generate the ensemble of theories this can become a very complex classifier that is difficult to interpret. In the case of clause-based learning, each individual classifier is composed by only one clause. So in the worst case, our final classifier will have size 25.

The accuracy, as presented in Figure 4(a), may lead to an interpretation that all methods are doing well when classifying positive and negative examples. In order to have a more detailed picture of the behaviour of the methods on both positive and

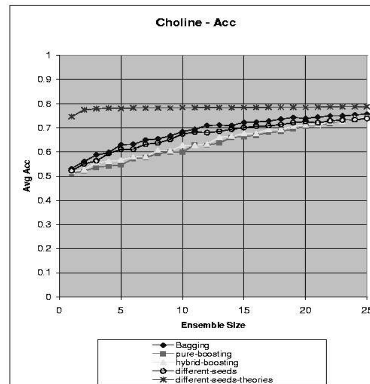
(a) Amine



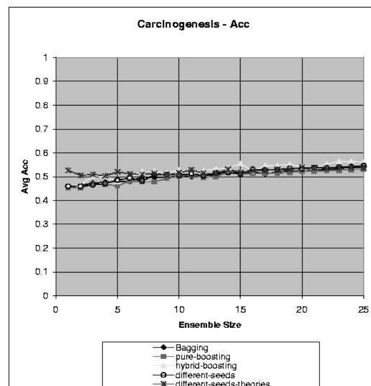
(b) Protein



(c) Choline



(d) Carcinogenesis



(e) Mutagenesis

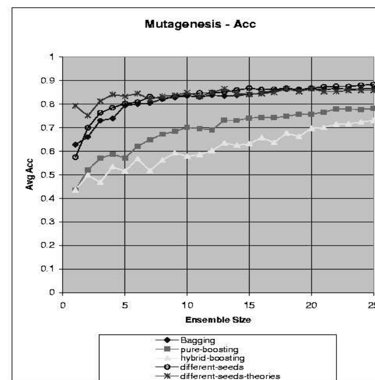


Fig. 4. Average Accuracies for the Five Applications.

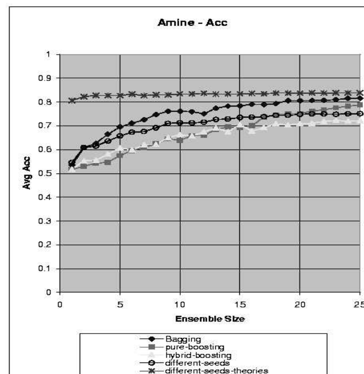


Fig. 5. Amine: Accuracies - fold 1.

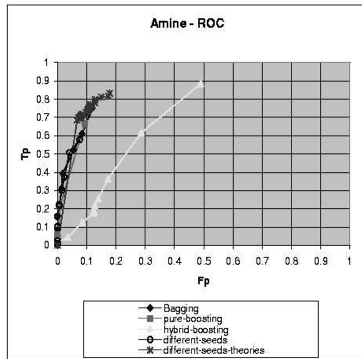
negative classification, we show the ROC curve for this application with ensemble size 25. In this figure, hybrid boosting is clearly classifying very well positive examples (covering almost 90% of them for ensemble size 1) at the cost of misclassifying negative examples, for small ensemble sizes. The other methods are behaving quite well, with bagging behaving better than the other methods.

In Figure 7(a) we give a summarised picture of the behaviour of all methods for all ensemble sizes by presenting the areas under the ROC curves for each ensemble size, for the application Amine. In this figure, the greater the area, the more precise is the classifier. From this picture we can observe that different-seeds-theories benefit very little from ensembles, as mentioned before, starting at ensemble size 5, and does not seem that will get better after ensemble size 25. The other methods have the tendency of always improving with the increase of the ensemble, with bagging being the best method.

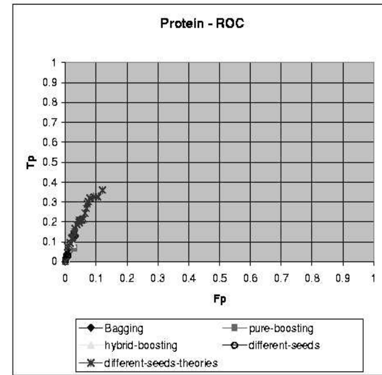
Figure 4(b) shows the average accuracies obtained by the application Protein for all ensemble methods. Contrary to the results produced by the application Amine, where different-seeds-theories produced slightly better accuracies than the other ensembles, the results produced by the application Protein shows similar performance (approximately 80%) to all ensemble methods. These results were obtained before in a previous work with different-seeds-theories and bagging applied to theories, using the same Aleph parameters [9]. Surprisingly, ensembles of single classifiers produce the same performance. Even a single clause (ensemble of size 1 in Figure 4(b)) produced similar performance to the other ensemble sizes. The reason for that is the similarity among the clauses and theories found by Aleph, and confirms that ensembles do not do very well when the classifiers do not vary.

Although Protein has achieved 80% of accuracy with all ensemble methods for all ensemble sizes, this result was obtained from a good classification of negative examples only. Protein misclassified most of the positive examples, with all ensemble methods, as it can be seen in Figure 6(b). The average accuracies were good because Protein has 4

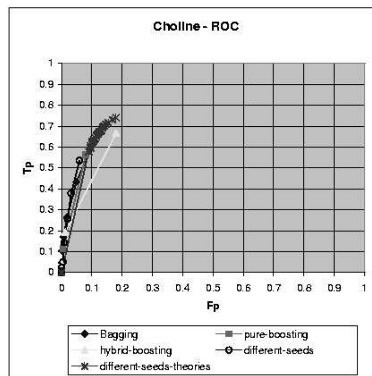
(a) Amine



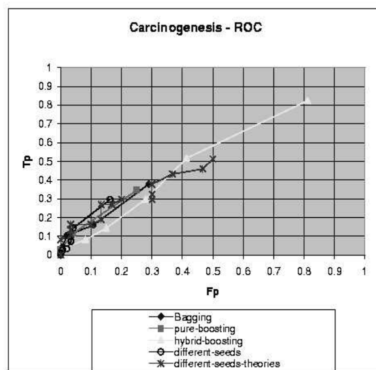
(b) Protein



(c) Choline



(d) Carcinogenesis



(e) Mutagenesis

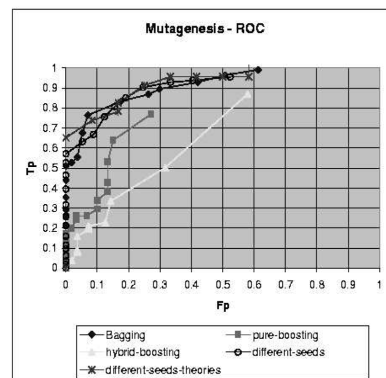
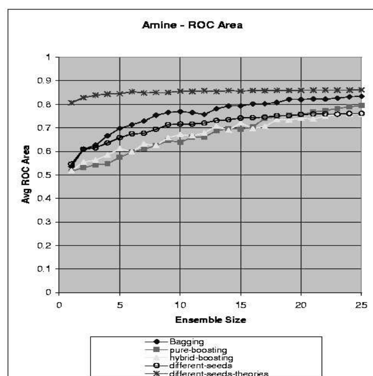
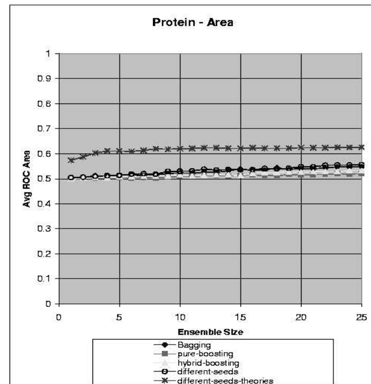


Fig. 6. ROCs for the Five Applications.

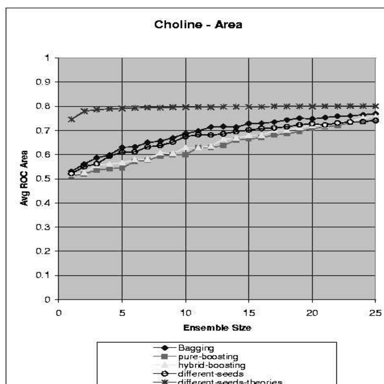
(a) Amine



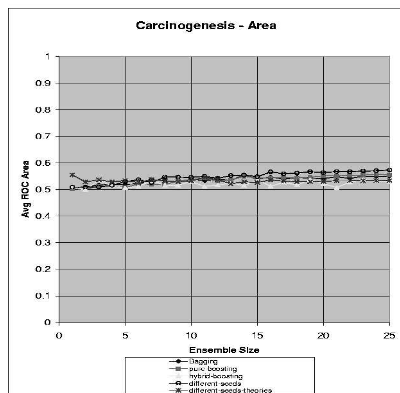
(b) Protein



(c) Choline



(d) Carcinogenesis



(e) Mutagenesis

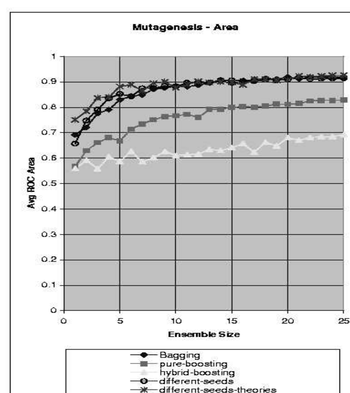


Fig. 7. Areas under the ROCs for the Five Applications.

times more negative than positive examples. According to Figure 6(b), different-seeds-theories is the best at classifying positive examples, but results are still poor.

If we look at the area under the ROC curves (Figure 7(b)), we confirm that different-seeds-theories produces the best result, but areas are only slightly over 60%. All methods had a small improvement from the ensemble of size 1 to ensemble of size 25.

Accuracies for the dataset Choline are shown in Figure 4(c). Similarly to the dataset Amine, Choline has a significant improvement in performance as we increase the ensemble size, for all methods that combine single clauses. The gain in performance is approximately 33% when we compare the accuracies of ensemble of size 1 with ensemble of size 25. Once more, different-seeds-theories does not benefit much from ensembles, although achieves slightly better performance than the other methods for ensemble of size 25. From the methods that combine clauses, the two ones based on boosting seem to have better behaviour with a linear gain in performance. Bagging and different-seeds have better behaviour to small ensemble sizes, but start to flatten out after ensemble of size 20. In the ROC curve (Figure 6(c)), different-seeds-theories is the method that achieves better performance by correctly classifying 75% of the positive examples and misclassifying only 18% of the negatives, for ensemble of size 1. Bagging performs a better classification of the negative examples for small ensemble sizes, but it does not perform so well as different seeds at the positive examples. The same happens to different-seeds, and pure boosting. Hybrid-boosting starts well for the positive examples, but as we increase the ensemble size, it starts to loose performance.

Figure 7(c) shows the overall performance of all methods, for all ensemble sizes. Different-seeds-theories slightly benefitted from ensembles, but seems to flatten out after ensemble of size 10. All the other methods improve linearly as we increase the ensemble size. Although different-seeds-theories, for ensemble of size 1, presents an accuracy that is similar to the accuracies produced by the other ensemble methods, at ensemble size 25, the time taken to obtain the first theory with different-seeds-theories was much higher than the time taken, for training and ensemble generation, by the other methods.

If we look at the average accuracies achieved by the Carcinogenesis application 4(d), we can observe that all methods are behaving similarly. This somehow contradicts results in the literature that say that a fairly large search space is needed to obtain a good clause. Different-seeds-theories exploited a very large search space and results are somewhat similar to the ones produced when learning single clauses where a much smaller search space was explored.

4.2 Quantitative Analysis

In the previous subsection we discussed about the quality of the implemented methods, and concluded that different-seeds-theories produced the best results, although it did not benefit much from the ensemble methods. However the difference between the best accuracy of different-seeds-theories and the best accuracy of the other methods is only 0.02% (ref. accuracy of Amine, ensemble size 25, with bagging).

Table 2 shows that the price to pay to have this 0.02% better value is too high when we look at the time spent by the ILP system to obtain the classifiers. This table shows the execution times of the cross-validation for all applications (we are not counting the

time to generate the final classifier, through the voting mechanism. This would add a proportional extra time to all slots of the table).

Table 2. Execution Times.

Datasets	*Bagging*	*Pure-boosting*	*Hybrid-boosting*	*Different-seeds*	*Different-seeds-theories*
Amine	12 min	9 min	6 min	7 min	12 h 55 min
Carcinogenesis	6 h 19 min	6 h 29 min	3 h 28 min	5 h 59 min	91 h 8 min
Choline	20 min	20 min	17 min	22 min	63 h 41 min
Mutagenesis	34 h 51 min	35 h 22 min	6 h 47 min	47 h 47 min	259 h 16 min
Protein	18 min	15 min	15 min	21 min	27 h 43 min

The datasets Amine and Protein were executed on the Pentium III, 750 MHz machine, Choline was executed on a Pentium III, 600 MHz, and the other datasets were executed on the Pentium IV, 1.2 GHz.

In general, the fastest method was hybrid-boosting, while different-seeds-theories is by far the worst of all methods taking more than 10 days to learn theories only for fold 1 for our slowest applications, carcinogenesis and mutagenesis. These are the only slots where we only count time for fold 1. All other slots show the learning times for all folds.

A second disadvantage of a theory as a classifier, and of ensemble of theories, is its complexity. Some of our results produced theories of size 70. Learning single clauses and iterating 25 times, as we did in this work, produces classifiers whose length is at most 25.

5 Discussion

The results shown before demonstrate several important facts in ILP learning:

- Ensembles are very powerful methods to obtain good accuracies in a small amount of time;
- Boosting methods are effective on obtaining good quality classifiers, even when the individual classifier is as weak as a single clause;
- Ensemble sizes greater than 25 may produce better accuracies for methods that learn clauses;
- Learning theories can be unfeasible depending on the application, and learning clauses can produce equal or better results;
- Theories do not benefit much from ensembles. We believe that this is because a theory is already an ensemble;
- Very weak individual classifiers such as single clauses can benefit significantly from ensembles, and producing a final classifier takes time that is several orders of magnitude less than the time spent to learn theories;

– Clauses are classifiers simpler than theories. Consequently, an ensemble of clauses is simpler than an ensemble of theories, which may help an expert to interpret the results.

6 Conclusions and Future Work

This work presented an empirical study of ensemble methods, in the Inductive Logic Programming setting. We chose to apply ensembles to two different individual classifiers: (1) classifiers composed of one single clause, faster to obtain; and (2) classifiers composed of one or more clauses (theory), that take a huge amount of time to obtain. We applied ensemble methods to classifiers obtained with the Aleph system. We implemented four different kinds of ensembles: different seeds (takes advantage of the random choice of seed of Aleph to produce clauses or theories), bagging, and two kinds of boosting. We tested five ILP applications already used in the literature.

Our results show that ensembles built from single clauses are more cost-effective than ensembles built from theories. Clauses are much faster to be learned, and an ensemble of clauses produces more readable classifiers.

As future work we intend to extend this methodology to other applications and investigate other ensemble approaches. We also would like to study how boosting incorporated to the Aleph system would behave when compared to the experiments performed here, without any modifications to the Aleph system.

References

1. E. Alpaydin. Multiple networks for function learning. In *IEEE International Conference on Neural Networks*, pages 9–14, 1993.
2. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 60–77. Springer-Verlag, 2000.
3. H. Blockeel, B. Demoen, G. Janssens, H. Vandecasteele, and W. Van Laer. Two advanced transformations for improving the efficiency of an ILP system. In J. Cussens and A. Frisch, editors, *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, pages 43–59, 2000.
4. L. Breiman. Bias, variance, and arcing classifiers. Technical Report 460, UC-Berkeley, Berkeley, CA, 1996.
5. L. Breiman. Stacked Regressions. *Machine Learning*, 24(1):49–64, 1996.
6. L. Dehaspe and L. De Raedt. Parallel inductive logic programming. In *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, 1995.
7. T. Dietterich. Ensemble methods in machine learning. In J. Kittler and F. Roli, editors, *First International Workshop on Multiple Classifier Systems, Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2000.
8. I. C. Dutra, D. Page, V. Santos Costa, J. Shavlik, and M. Waddell. Toward Automatic Management of Embarrassingly Parallel Applications. In *Proceedings of the EUROPAR'03*, pages 509–516, Aug 2003. Klagenfurt, Austria.

9. I. C. Dutra, D. Page, and J. Shavlik V. Santos Costa. An empirical evaluation of bagging in inductive logic programming. In *Proceedings of the 12th International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, pages 48–65. Springer-Verlag, September 2002.
10. Y. Freund and R. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 148–156. Morgan Kaufman, 1996.
11. J. Graham, D. Page, and A. Wild. Parallel inductive logic programming. In *Proceedings of the Systems, Man, and Cybernetics Conference*, 2000.
12. L. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, October 1990.
13. S. Hoche and S. Wrobel. Relational learning using constrained confidence-rated boosting. In Céline Rouveirol and Michèle Sebag, editors, *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 51–64. Springer-Verlag, September 2001.
14. A. Krogh and J. Vedelsby. Neural network ensembles, cross validation, and active learning. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 231–238. The MIT Press, 1995.
15. N. Lincoln and J. Skrzypek. Synergy of clustering multiple backpropagation networks. In *Advances in Neural Information Processing Systems*. Morgan Kaufmann, 1989.
16. T. Matsui, N. Inuzuka, H. Seki, and H. Ito. Parallel induction algorithms for large samples. In S. Arikawa and H. Motoda, editors, *Proceedings of the First International Conference on Discovery Science*, volume 1532 of *Lecture Notes in Artificial Intelligence*, pages 397–398. Springer-Verlag, December 1998.
17. D. W. Opitz and J. W. Shavlik. Actively searching for an effective neural-network ensemble. *Connection Science*, 8(3/4):337–353, 1996.
18. J. R. Quinlan. Boosting first-order learning. *Algorithmic Learning Theory, 7th International Workshop, Lecture Notes in Computer Science*, 1160:143–155, 1996.
19. V. Santos Costa, A. Srinivasan, and R. Camacho. A note on two simple transformations for improving the efficiency of an ILP system. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 225–242. Springer-Verlag, 2000.
20. M. Sebag and C. Rouveirol. Tractable induction and classification in first-order logic via stochastic matching. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 888–893. Morgan Kaufmann, 1997.
21. A. Srinivasan. A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95–123, 1999.
22. A. Srinivasan. *The Aleph Manual*, 2001.
23. J. Struyf and H. Blockeel. Efficient cross-validation in ILP. In Céline Rouveirol and Michèle Sebag, editors, *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 228–239. Springer-Verlag, September 2001.
24. F. Zelezny, A. Srinivasan, and D. Page. Lattice-search runtime distributions may be heavy-tailed. In *Proceedings of the 12th International Conference on Inductive Logic Programming*. Springer Verlag, July 2002.
25. S. Zemke. Bagging imperfect predictors. In *Proceedings of the International Conference on Artificial Neural Networks in Engineering, St. Louis, MI, USA*. ASME Press, 1999.
26. M. Zweig and G. Campbell. Receiver-operative characteristic. *Clinical Chemistry*, 39:561–577, 1993.