A Simple Approach to a Multi-Label Classification Problem

Fabrizio Riguzzi

Dipartimento di Ingegneria, Università di Ferrara, Via Saragat 1
44100 Ferrara, Italy,
friguzzi@ing.unife.it

Abstract. The approach to handle multiple label for each gene is to have a learning problem for each label that appears in `yeast.labelled`. In each learning problem, a gene is a positive example if it contains that label, otherwise it is a negative example. In this way we learn one classifier for each label. To label unseen genes, we run each generated classifier on the gene data and we assign the label to the gene if the classifiers gives a positive answer.
As a classifier, we have used Tilde for its speed and good accuracy. In order to finish the experiments before the deadline we had to consider only a subset of the available data, namely the protein secondary structure data.

1 Description of the Data

The aim of the ILP-05 Challenge is to predict the functional classes of yeast genes. The allowed functional classes belong to the FunCat hierarchical classification scheme from MIPS.

We are given the functional annotation of 4012 genes in the file `yeast.labelled`. Each gene is associated to one or more functional classes. This makes the problem different from an usual classification problem because each example can have more than one label.

We are given two types of information regarding the genes: homology data and protein secondary structure data. Both types of information come in the form of Prolog facts divided into files, with one file per gene. The sum of the dimensions of the files for the homology data is 796 MB, while for the protein secondary structure is 9.77 MB.

The aim is to label 674 genes listed in the file `yeast.heldout`.

2 Overview of the Approach

In order to tackle the multi-label problem, we decided to try the simplest approach: we generated a learning problem for each label that appears in `yeast.labelled`. In each learning problem, a gene is a positive example if it contains

the considered label, otherwise it is a negative example. Each learning problem is thus a two-class problem that can be solved with any ILP system.

In order to label unseen genes, we run each generated classifier on the gene data and we assign the label to the gene if the classifier gives a positive answer.

For solving the individual learning problems, we decided to use Tilde [2] for the following reasons:

- it is quite fast, since it adopts a simultaneous covering strategy similar to c4.5
- it contains various speed optimizations (e.g. the use of query packs)
- its accuracy is comparable to other state of the art ILP systems

3 Details of Experiments

In `yeast.labelled` there are 469 different labels. Among these, there is label '99' that means unknown function and the empty label " that we considered as noise. Therefore we generated 467 different learning problems. In the generation of the training sets, we decided to add a gene to the set of positive examples only if the functional class appears among the labels of the gene. We did not add a gene to the set if it is labeled with a child of the considered functional class. For example, if a gene is labeled with ['01.03','02.04.05'] we will not add it to the set of positive examples for the functional classes '01' or '02.04'.

This was actually a misconception: we learned from the referees that this interpretation was wrong, we should have added a gene to a set also if it is labeled with a child of the considered functional class. Unfortunately in the time available it was not possible to repeat the experiments with the correct interpretation.

Then we decided which background knowledge is to be given as input to Tilde. In order to use both homology data and protein secondary structure data, we need to use the "models" format, where all the information regarding a gene is stored together into a "model", i.e., a portion of the input file containing the examples (the file with extension .kb). This format allows Tilde to load the models into main memory in chunks, a feature that is necessary because the input data does not fit entirely in the main memory of the computers we have. In order to exploit this feature, the background knowledge has to contain only the intensional definitions of the comparison predicates (see below). Thus we need to create a file of about 800 MB for each learning problem. As this would lead to learning times that are too long for the experiments to be completed before the deadline, we tried to consider a subset of the homology data: we extracted only the facts relative to the `eval/3` predicate. In this case the input file would be 120 MB large. This gives rise to too long learning times as well.

Therefore we decided to use only the protein secondary structure data. By using only these data we are able to provide the data to Tilde in the "key" format: each gene is represented by a fact of the form `g(gene,class)` where `gene` is the name of the gene and `class` is either `yes` or `no`. We set `class` as the argument to be predicted by Tilde. The protein secondary structure data was

merged in a single file. No other preprocessing was necessary. In particular, the protein secondary structure file was used for holding the background knowledge for all the learning problem: the background file (with extension .bg) for each learning problem loads this file, thus easing the management of data.

The language bias was the following. The protein secondary structure data contain two predicates:

```
struc(gene,order,type,length).
struc_dist(gene,type,percentage).
```

`type` is a categorical attribute that can assume the values `a`, `b` or `c`. `order` and `length` are integer attributes that assume respectively 484 and 240 different values. `order` assumes values between 1 and 484 while `length` assumes values between 1 and 599. `percentage` is real attribute that can assume the values between 0 and 100.

As regards the language bias, we allowed the addition of `struc(G,O,T,L)` and `struc_dist(G,T,P)` to the conjunctions associated to nodes in the tree. Moreover, literals can be added that compare the attributes `T`, `O`, `L`, and `P` with constants. In particular, `T` is compared using equality (predicate `eq/2`) with the constants `a`, `b` or `c`. `O`, `L` and `P` are compared to thresholds using the predicates smaller or equal than (`smeq/2`) and greater or equal than (`greq/2`). The predicates `eq/2`, `smeq/2` and `greq/2` are defined in the background file. The thresholds are obtained by discretizing the attributes `O`, `L` and `P` into a number of intervals using the algorithm described in [1] and available in Tilde. The thresholds are the bounds of the intervals. The number of thresholds is chosen by the user, we used 5.

We also had to specify lookahead statements: these are used in order to allow Tilde to add more than one literal at a time during refinement. In particular, we specified that, when `struc(G,O,T,L)` or `struc_dist(G,T,P)` are present in a conjunction, then any of the comparison literals can be added. We set the maximum lookahead levels to 3.

As regards the other settings, we used the default values for all the settings apart from `minimal_cases` and `pruning`. `minimal_cases` specifies the minimal number of cases that a leaf has to cover: its default value is 2, we set it to 1. `pruning` specifies the type of pruning of the tree: it defaults to `c4.5` while we set it to `none`, i.e., no pruning is performed. We used these values because the number of positive examples in each learning problem is usually small (the average number of positive examples is 27.33 over a total of 4012 examples), so with the default settings we would usually obtain a single leaf tree with label `no`. Thus, in order to accurately model all positive examples, we run the risk of overfitting.

The experiments have been performed by distributing the learning problems on a number of machines. Each machine has a Pentium 4 at 3.00 GHz with 512 MB of RAM, running the Windows 2000 operating system.

When we noticed that a learning problem was taking too much time (more than eight hours), it was stopped. Of the 467 runs, 66 were stopped before completion and 401 were completed. The average execution time of the runs is

2.27 hours (the sum of discretization time and induction time), with a standard deviation of 8.09 hours. The distribution of execution times is actually quite skewed, with a maximum of 75.50 hours and 10 runs taking more than 30 hours.

For the class 01 we obtained the tree shown in Figure 3. It contains 9 nodes and 13 literals. The tree was learned in 8.39 seconds of CPU time, of which 3.1

```
g(A,B)
struc(A,C,D,E),gteq(E,30) ?
+--yes: smeq(C,3) ?
|       +--yes: struc(A,F,G,H),greq(H,42) ?
|       |       +--yes: [no] [305.0/305.0]
|       |       +--no:  smeq(C,2) ?
|       |               +--yes: [no] [68.0/68.0]
|       |               +--no:  greq(E,36) ?
|       |                       +--yes: [no] [20.0/20.0]
|       |                       +--no:  struc(A,L,M,N),greq(N,37) ?
|       |                               +--yes: smeq(L,25) ?
|       |                               |       +--yes: [no] [3.0/3.0]
|       |                               |       +--no:  [yes] [1.0/1.0]
|       |                               +--no:  struc(A,Q,R,S),greq(Q,89) ?
|       |                                       +--yes: eq(R,a) ?
|       |                                       |       +--yes: [no] [8.0/8.0]
|       |                                       |       +--no:  [yes] [1.0/1.0]
|       |                                       +--no:  [no] [17.0/17.0]
|       +--no:  [no] [1709.0/1709.0]
+--no:  [no] [1880.0/1880.0]
```

Fig. 1. The tree learned for function 01.

were spent in discretization and 5.29 in induction.

The output of Tilde for each learning problem was then parsed in order to isolate the prolog program that corresponds to the tree, so that it can be used to classify unseen genes.

Each program was then run against each gene in the `yeast.heldout` file. If the tree gives a positive answer, the function associated to the tree is added to the set of the gene label.

4 Evaluation of the Proposed Approach

The choice of generating a learning problem for each label of a multi-labeled problem is simple and allows one to use any ILP system that does classification. The approach can be adopted even when the amount of data available is large if one adopts a clever way of managing the large files, as for example the on the fly generation of them. We adopted Tilde as an ILP system because it is a fast learner with a good accuracy (similar to other state of the art ILP systems). It is fast both because it adopts a greedy covering algorithm and because it contains various speed optimizations (query packs and selective loading of chunks of data). With selective loading of chunks of data it was tested on files up to 2 GB in size [3].

Thus Tilde would be able to manage the data available for yeast genes which total 800 MB. However, the time available before the Challenge deadline prevented us from exploring this approach. Rather, we decided to use only a portion of the data available, namely the protein secondary structure data, thus having a background file of 9.77 MB.

Even with this reduced dataset size, the average execution time of each run was 2.27 hours. So, overall, the experiments required 909 hours of CPU time.

Since the number of positive example is quite small, we decided to set the minimum number of example per leaf to 1 and to avoid post-pruning the tree. This was done in order to try to model as accurately as possible the available positive examples. Unfortunately, in some cases, this lead to overfitting: very large trees were produced.

We have looked at the trees generated for some of the learning problems and we observed that none of them used the predicate `struc_dist/3`. This is probably due to the fact that the fact for `struc_dist/3` are much less than those for `struc/4`.

So the trees separate positive from negative examples by considering secondary structure elements and by comparing the values of the attributes order, type and length with constants. The interpretation of the trees is made difficult by the fact that each gene has usually many secondary structure elements and that the tests can refer to attributes from different elements. However, we feel this is a problem of the first-order representation in general, rather than of first-order logical decision trees.

For example, in the case of Figure 1, the classifier could be translated into English in the following way: "if the gene has a secondary structure element whose length is below 30, then it does not have the function, otherwise if the order of the same element is greater than 3 then it does not have the function, otherwise if it has a secondary structure element whose length is smaller than 42 then it does not have the function, otherwise if the order of the first element is smaller than 2 then it does not have the function, otherwise if the length of the first element is greater than 36 then it does not have the function, ...".

5 Conclusions

We have presented a simple approach for learning models of the functions of yeast genes. The approach uses brute force in two ways: a learning problem for each function is generated and the available data is provided as it is to an ILP learner, without any elaboration.

In particular, we did not try to produce propositional features from the data, for example by aggregating over the secondary structure elements. This was done in order to investigate the effect of a purely relational approach to learning. The evaluation of the results will tell whether this is sufficient in this domain. In general, however, the computation of aggregated values should be combined with a relational approach.

The paper proves that a relational approach is possible for predicting the functions of yeast genes.

The approach can be improved in several ways. First, we can generate the training sets for each function by taking into account also the genes labeled with ancestors of the functions. Second, the settings used lead to overfitting in some cases, so we should revert to the standard settings. Third, given more time available, we can exploit all of the available knowledge. Fourth, propositional features can be added by aggregating numeric values over the elements of the domain.

References

1. H. Blockeel and L. De Raedt. Lookahead and discretization in ilp. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–84. Springer-Verlag, 1997.
2. Hendrik Blockeel and Luc De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
3. Jan Struyf, 2005. Personal communication.