SATzilla2007: a New & Improved Algorithm Portfolio for SAT

Lin Xu, Frank Hutter, Holger H. Hoos and Kevin Leyton-Brown
Computer Science Dept., University of British Columbia
Vancouver, BC, Canada
{xulin730, hutter, hoos, kevinlb}@cs.ubc.ca

1 Introduction

Empirical studies often observe that the performance of algorithms across problem domains can be quite uncorrelated. When this occurs, it seems practical to investigate the use of algorithm portfolios that draw on the strengths of multiple algorithms. SATzilla is such an algorithm portfolio for SAT problems; it was first deployed in the 2004 SAT competition [4]. SATzilla is based on *empirical hardness models* [3, 5], learned predictors that estimate each algorithm's runtime on a given SAT instance.

SATzilla2007 is a new version of SATzilla that incorporates new research on empirical hardness models:

- Regression based on partly censored data (i.e. including runs that time out);
- Probabilistic prediction of instance satisfiability;
- Hierarchical models: separate models for satisfiable and unsatisfiable instances, and probabilistic combination of the two for prediction.

SATzilla2007's methodology can be outlined as follows:

Offline, as part of algorithm development:

1. Identify a target distribution of problem instances.
2. Select a set of algorithms having relatively uncorrelated runtimes on this distribution.
3. Using domain knowledge, identify features that characterize problem instances.
4. Compute features and determine algorithm running times.
5. Use regression to construct models of algorithms' runtimes.

Online, given an instance:

1. Compute feature values.
2. Predict each algorithm's running time using learned runtime models.
3. Run the algorithm predicted to be fastest.

2 SATzilla Framework

SATzilla2007 includes the following six solvers: Minisat2.0, March_dl, Vallst, Zchaff_rand, kcnfs2006, and SAPS (with the best fixed parameter setting from [2]). The version submitted to the demonstration division also contains the proprietary solvers Eureka and RSAT.[1]

For training data, we used all available SAT instances from previous SAT competitions (2002 until 2005) and from the SAT Race 2006. Based on these instances, we built three data sets:

- *Random*, consisting of all 2,300 random instances;
- *Crafted*, consisting of all 1,490 handmade/crafted instances;
- *All* containing all 4,811 instances.

For each training instance we ran each algorithm and recorded its runtime. For *Random* and *Crafted*, we ran the algorithms again after preprocessing with HyPre [1]. The timeout value for HyPre was set to 60 seconds, and the timeout for each algorithm run was set to at least 30 minutes (one hour when we could afford it).

We computed 64 characteristic features for each instance, a computationally cheap subset of the features used in [5].[2] These comprised 33 basic features, 7 features from DPLL probes, and 24 features from local search probes. All features were normalized to mean zero and standard deviation one on the training set. The time required for computing basic features was instance-dependent with a timeout of 60 seconds. Two seconds were allocated to computing the local search features for each instance, and one second for DPLL probes. On *Random* and *Crafted*, the average complete feature computation time was five seconds; for *All*, it was 31 seconds.

Since the SAT competition scoring function rewards quick algorithm runs, we cannot afford to do this feature computation for very easy instances (for which runtimes around five seconds are much too large). Thus, we looked at the raw runtime data to identify the algorithm that was most efficient in solving easy instances. This algorithm was March_dl, which solved 32%, 30.5%, and 29.9% of the instances in our *Random*, *Crafted*, and *All* data sets within five seconds. For the remaining instances, we decided to run SAPS for two seconds. We decided to do so because its runtimes are completely uncorrelated with March_dl (Pearson correlation coefficient

[1]Adding and deleting solvers is very straightforward and does not require retraining the existing models. SATzilla2007 will be available online with a description of how to plug in your own solver.

[2]E.g., the linear programming and clause graph features from [5] were discarded since they timed out on too many instances.

-0.014 for the 398 remaining instances that both solvers solved). This SAPS phase solves 12.0%, 6.9%, 3.6% of the remaining *Random*, *Crafted* and *All* instances.

For each of our data sets we then built predictive models of runtime for all our algorithms except SAPS[3] for the remainder of the instances. For *Random* and *Crafted*, we also built separate predictive models for each algorithm when the instance is first preprocessed with HyPre. Since these latter models include preprocessing time we can view the combination of two preprocessings (HyPre/none) and five algorithms as 10 different solvers, for each of which we have a predictive model of runtime.

Since predictive models are typically not perfect, it is inadvisable to use too many solvers in a portfolio: when a solver performs poorly on an instance but is predicted to perform best, then dropping this solver from the portfolio can increase overall performance. Picking the optimal subset of solvers is a a simple subset selection problem: pick the portfolio that, when run using the predictive models for each solver, achieves the overall lowest runtime. We performed exhaustive subset selection for our 10 solvers (seven for *All*). This led us to select the following combinations of solvers for each of our data sets ("+p" stands for "plus preprocessing"):

- *Random*: March_dl, kcnfs2006, Minisat2.0+p;
- *Crafted*: Minisat2.0, Minisat2.0+p, March_dl+p, March_dl, Vallst+p, Zchaff_rand+p;
- *All*: Minisat2.0, Eureka, March_dl, kcnfs2006, Vallst, Zchaff_rand.

In summary, when SATzilla is asked to solve an instance, it first runs March_dl for five seconds, then runs SAPS for two seconds, then computes features (typically in less than five seconds), feeds the computed features into each of the predictive models to get runtime predictions (a matter of milliseconds) and then runs the best predicted solver until timeout. If feature computation times out, a default solver is used. If a solver crashes, the next best one is run using the time that remains.

3 New Technologies

SATzilla implements a number of (as yet unpublished) improvements for building empirical hardness models.

Censored data. A large portion (for each solver at least about half) of the runs we used for training timed out. In statistics, data points for which all that is known is a lower bound on the response variable are called *right-censored*. The previous version of SATzilla and other published work on empirical hardness models [3, 5, 2] have either pretended that a censored run finished at the cutoff time, or have dropped censored data. Both approaches produce biased models. Here, we follow the iterated reweighted least squares (IRLS) approach from [6] to dealing with censored data, leading to much better results. We start by learning a model only from uncensored data. This is followed by two steps: (1) the response variable for the censored data is filled in by model predictions, and (2) the model is refitted. These steps are iterated until the model converges.

Predictions of satisfiability status. For many instances, it is possible to predict with surprisingly high accuracy whether the instance is satisfiable or not. The features we use are exactly the same features used for building the predictive models of runtime. We implement probabilistic classification via logistic linear regression. If this technique is used for a hard classification of instances into satisfiable and unsatisfiable, classification performance is 95.2% on data set *Random* and 80.8% on *Crafted* (we did not use this approach for data set *All* because it includes many instances with unknown satisfiability status).

Hierarchical models. Finally, we exploit the above probabilistic classification into satisfiable/unsatisfiable, and the fact that we can build more accurate models conditional on knowing the satisfiability status of instances. We train separate models using only satisfiable/unsatisfiable instances, and combine their predictions. This approach substantially outperformed a naïve approach that only trains a single model for both satisfiable and unsatisfiable instances.

4 Expected Behaviour

We submit three different version of SATzilla, one for each of our training data sets. We expect satzilla-r (random) to perform well on random instances, satzilla-c to perform well on crafted instances, and satzilla to perform well overall. The latter includes two proprietary solvers and is thus submitted to the demonstration division.

References

[1] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *LNCS 2919: Revised Selected Papers of SAT 2003*, pages 341–355. Springer, 2004.

[2] F. Hutter, Y. Hamadi, K. Leyton-Brown, and H. H. Hoos. Performance prediction and automated tuning of randomized and parametric algorithms. In *Proc. of CP-06*, pages 213–228, 2006.

[3] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Proc. of CP-02*, 2002.

[4] E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, and H. Hoos. SATzilla: An algorithm portfolio for SAT, 2004.

[5] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proc. of CP-04*, 2004.

[6] J. Schmee and G. J. Hahn. A simple method for regression analysis with censored data. *Technometrics*, 21(4):417–432, 1979.

[3]Even though it is possible to build highly predictive models of runtime for local search algorithms like SAPS (see [2]), the limitation of local search to satisfiable instances leads us to only use it for a short amount of time in the beginning but not include it in the portfolio.