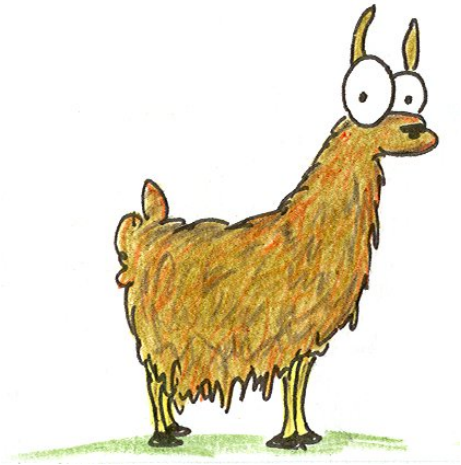
LLAMA: Leveraging Learning to Automatically Manage Algorithms

Lars Kotthoff



Abstract

Algorithm portfolio and selection approaches have achieved remarkable improvements over single solvers. However, the implementation of such systems is often highly customised and specific to the problem domain. This makes it difficult for researchers to explore different techniques for their specific problems. We present LLAMA, a modular and extensible toolkit implemented as an R package that facilitates the exploration of a range of different portfolio techniques on any problem domain. It implements the algorithm selection approaches most commonly used in the literature and leverages the extensive library of machine learning algorithms and techniques in R. We describe the current capabilities and limitations of the toolkit and illustrate its usage on a set of example SAT problems.

This document corresponds to LLAMA version 0.9.1.

Quick start

So you know about algorithm portfolios and selection and just want to get started. Here we go. In your R shell, type

```
install.packages("llama")
library(llama)
```

to install and load LLAMA. We're going to assume that you have two input CSV files for your data – features and times. The rows designate problem instances and the columns feature and solver names. All files must have an 'ID' column that allows to link them. Load them into the data structure required by LLAMA as follows.

```
data = input(read.csv("features.csv"), read.csv("times.csv"))
```

You can also use the SAT solver data that comes with LLAMA by running

```
data(satsolvers)
data = satsolvers
```

Now partition the entire set of instances into training and test sets for cross-validation.

```
folds = cvFolds(data)
```

This will give you 10 folds for cross-validation. Now we're ready to train our first model. To do that, we'll need some machine learning algorithms – we're going to use a random forest classifier. Now train a simple classification model that predicts the best algorithm.

```
model = classify(makeLearner("classif.randomForest"),
    folds)
```

Great! Now let's see how well this model is doing and compare its performance to the virtual best solver (VBS) and the single best solver in terms of average misclassification penalty.

```
mean(misclassificationPenalties(folds, model))
## [1] 73.31597
mean(misclassificationPenalties(data, vbs))
## [1] 0
mean(misclassificationPenalties(data, singleBest))
## [1] 122.3186
```

While we are quite far off the virtual best, our classifier beats the single best algorithm! Not bad for a model trained in a single line of code.

You can use any other classification algorithms instead of `randomForest` of course. You can also train regression or cluster models, use different train/test splits or preprocess the data by selecting the most important features. More details in the on-line documentation, or just continue reading for an in-depth tour of LLAMA.

Contents

1 Background **3**

2 Anatomy of LLAMA **4**
2.1 Implementation 6

3 LLAMA for domestic use **6**
3.1 Installing LLAMA 7
3.2 Reading data 7
3.2.1 Algorithm Selection Benchmark Library 9
3.2.2 Example data 9
3.3 Slicing and dicing the data 9
3.4 Training and evaluating models 10
3.5 Other available model types 13

4 Advanced functionality **15**
4.1 Processing the input data 15
4.1.1 Normalising feature values 16
4.1.2 Imputing censored runtimes 16
4.2 Portfolio analysis 17
4.3 Ensembles and stacking 18
4.4 Parallel execution 21
4.5 Instance weights 21

5 Visualising the data **22**
5.1 LLAMA's plotting functions 22
5.2 Other ways of plotting data 24

6 Case study: SATzilla **32**
6.1 Presolver 34
6.2 Prediction of satisfiability 34
6.3 Prediction of feature computation time 35
6.4 Putting it all together 36

7 The further domestication of LLAMA **37**

1 Background

Throughout this document, we will assume that the reader is somewhat familiar with algorithm portfolios, algorithm selection, and combinatorial search problems. In this section, some of the background is explained and pointers to additional materials given. Readers familiar with the matter may skip ahead to the next section.

We also assume a basic familiarity with how machine learning works. Readers new to this area can find background material in a variety of text books, e.g. [2, 17, 22].

An algorithm portfolio [6, 8] is a collection of state of the art solvers that are all capable of solving the same kind of problem. The rationale of using more than one algorithm or solver for a set of problems is that no single algorithm will be the best for all of these problems. This is known as the no free lunch theorem [24]. If more than one solver is available, we can (at least in theory) choose the best one for each particular problem, thus achieving superior overall performance. The idea of algorithm portfolios was inspired by portfolios in Economics, where a total investment is distributed over multiple securities to minimise the risk.

Many contemporary solvers for artificial intelligence problems have complementing strengths and weaknesses. On a set of problems where one solver exhibits bad performance, another will excel while the picture may be reversed on a different set of problems. Algorithm portfolios exploit this by relating the structure of the problem to solve to the performance of an individual solver or a set of solvers.

SAT is one of the first areas of artificial intelligence that algorithm portfolios and algorithm selection techniques have been applied to, and with great success. The most prominent system is probably SATzilla [27], which has dominated SAT solver competitions when it was introduced. More recent systems include ISAC [11], Hydra [26] and 3S [10].

To use an algorithm portfolio for solving problems, a selection mechanism is required to determine the algorithm to use in the particular case. The concept is closely related to the Algorithm Selection Problem [20], which is concerned with identifying the most suitable algorithm for solving a problem. Usually, some kind of machine learning model is learned to relate the features of a problem instance to the performance of an algorithm or a portfolio. Problem instance features can be anything that describes the instance, for example structural features such as the number of variables in a search problem or probing features such as the progress made after running a benchmark algorithm for a short amount of time on the instance.

There are different ways in which such machine learning models can be used. In the simplest case, a single classification model is trained to predict the best algorithm, given the features of a problem instance. Alternatively, one regression model per algorithm can be trained to predict its performance. The performance predictions can then be used to choose the best algorithm. Another approach is to cluster the problem instances in the training set, determine the best algorithm

for each cluster and assign new instances to the closest cluster. These and many more approaches have been used in the literature. LLAMA supports four fundamentally different approaches and a large number of variations of these involving ensembles and stacking.

A lot more background information can be found in [15] (even more in the extended version [13]) and the overview table of the relevant literature at `http://larskotthoff.github.io/assurvey/`.

The main drawback of the systems described in the literature is that they are highly tailored and customised for the particular problem domain or even set of problems. On top of that, the implementation may not be available, or may require an obsolete version of Matlab and the respective author's special environment that makes it work. Even though the high-level approach can usually be applied to other problems, in practice this is almost always very difficult or even impossible. This makes it very difficult to compare different approaches and prototype new ideas especially for researchers who are not algorithm portfolio experts.

This is exactly what LLAMA addresses. Instead of providing a highly-specialised approach that has been tuned and customised to yield high performance on a specific data set, LLAMA is a framework that provides the building blocks for automatic portfolio selectors. It supports the most common approaches to portfolio selection and offers the possibility to combine them into more sophisticated approaches. It furthermore provides an implementation of the infrastructure that is required to build, evaluate and apply algorithm portfolios in practice.

LLAMA is intended to be used by researchers working in the areas of algorithm portfolios, algorithm selection, and algorithm configuration and tuning. It is not particularly user-friendly or easy to use. It does not offer an industrial-strength C++ implementation that you can use in a high-performance portfolio solver. It can be used as a tool for designing such systems, but it will not do all the required work for you.

2 Anatomy of LLAMA

The main focus of LLAMA is to provide the user with a framework for the implementation and evaluation of different algorithm selection approaches. It is *not* meant to provide turn-key algorithm portfolio systems that can be used in competitions or similar settings. While the functionality it provides can certainly be used to facilitate the creation of such systems, a lot of the technical details for practical algorithm selection systems are highly domain-specific. The main audience LLAMA targets are researchers that wish investigate and explore the performance characteristics of algorithm selection systems in general.

The overall architecture of LLAMA is illustrated in Figure 1. At a high level, LLAMA takes problem feature and solver performance data as input, processes it, and produces the algorithm selection model and a characterisation of its performance as output. There is no explicit support for computing features, as

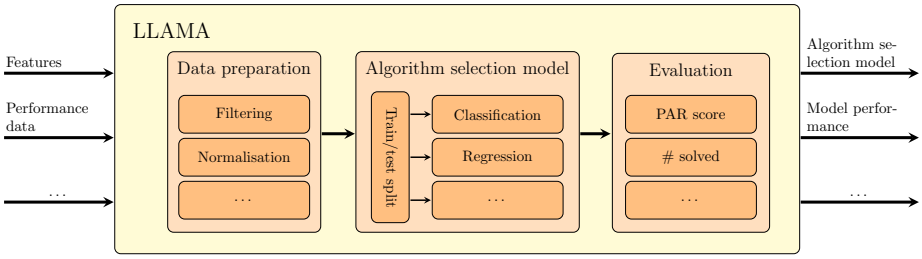


Figure 1: Overview of the architecture of LLAMA.

there are already many domain-specific systems that do this, e.g. SATzilla. The provided functionality falls into three main categories.

First, functions for data preparation are provided, such as filtering and normalising the feature data, partitioning the data set for evaluation, and analysing the contributions of the solvers in the portfolio to its overall performance. Input data can be read from a variety of sources, e.g. CSV files.

The second category comprises the model-building functionality. This includes functionality required to facilitate a clean evaluation of the learned models, i.e. functions to partition into training and testing sets. All the main approaches used in the literature are represented – one can build models treating algorithm selection as label classification, regression models that predict the performance of each solver in the portfolio, clustering models that assign the best solver to each cluster, and models that predict which solver is faster for each pair of solvers. All this functionality is available through a unified interface – changing the type of algorithm selection model requires only a different function call, changing the type of machine learning used to induce the model requires only a change of parameter to the model-building function call. Similarly, the output produced by these functions implements a common interface.

The third category of functionality contains the functions used to evaluate the learned models. Again, all commonly used evaluation measures, such as number of instances solved and PAR10 score, are supported. These measures can be reported for each individual problem instance or as averages.

Of the data preparation functions, any number can be used on a given data set. One could for example read the data, normalise the feature values, and filter the irrelevant features. In other cases, just reading the data may be sufficient. The processed data can then be used to build one or more algorithm selection models, depending on the requirements of the user. For a given application, only a single model may be required, while for a performance comparison several models would be needed. The learned models are then passed to the evaluation functions. Again it will depend on the application whether computing just one or several evaluation criteria makes sense.

All functions communicate through a set of common interfaces, which make it easy to extend the functionality. To implement a new model-building approach for example, the code to process the input and produce the output can

be reused, and the researcher is free to focus on the actual algorithm, on which no restrictions are imposed.

2.1 Implementation

LLAMA is implemented as an R package. There are many advantages to this approach; one of the main ones is that all the functionality available in R can be used to build algorithm selection models. This is not limited to the functionality that is implemented in R itself – there are interfaces to many other packages, such as the well-known Weka machine learning toolkit [7].

The large number of machine learning approaches and algorithms available in R makes it possible to use LLAMA to quickly evaluate a range of different techniques for algorithm selection on given data, such as presented in [16]. Being able to do so is crucial for achieving good performance in practice. LLAMA requires that machine learning algorithms are available through the mlr package [1].

3 LLAMA for domestic use

LLAMA is implemented as an R package and can be found at `http://cran.r-project.org/web/packages/llama/`, the development repository is at `https://bitbucket.org/lkotthoff/llama`. One of the main advantages of the R package implementation is that all the functionality available in R can be used to build performance model. This is not limited to the functionality that is implemented in R itself – there are interfaces to many other packages, such as the well-known Weka machine learning toolkit [7].

The large number of machine learning approaches and algorithms available in R makes it possible to use LLAMA to quickly evaluate a range of different techniques for algorithm selection on given data, such as presented in [16]. Being able to do so is crucial for achieving good performance in practice.

LLAMA uses the `mlr` package [1] as an interface to R's many machine learning method implementations. The advantage is that a wide range of implementations are supported with a common interface, making it easier for LLAMA to use them than the vanilla functions. Familiarity with `mlr` is helpful, but not required to use LLAMA.

LLAMA provides a number of high-level functions that can be used to create and evaluate algorithm selection models with just a few lines of code. It is helpful to be familiar R and its language, although this document does not assume that you are. You will however need to be somewhat familiar with R to use the more sophisticated functionalities of LLAMA. There are many books on R, e.g. [3].

All of the functions LLAMA provides are documented in R's online help system, usually with examples of how to use it. To access a help page, simply type `?<name of function>`.

3.1 Installing LLAMA

LLAMA is available on CRAN. On a computer connected to the internet, all you have to do is open an R terminal and type

```
install.packages("llama")
```

Alternatively, you can use the graphical package manager your R distribution provides, or download the package file yourself and install it manually.

Once the package is installed, you can load it with

```
library(llama)
```

3.2 Reading data

Let's start at the beginning – getting your data into LLAMA. It uses a special data structure that contains, besides the actual performance and feature data, meta data about which algorithm was the best in which case, how to extract feature and performance values, and other information that is required by the various functions that operate on it. Throughout this document, we will talk about the "performance" of an algorithm – this will usually be its runtime, but can be other things such as the quality of an obtained solution. LLAMA places no restrictions on what "performance" means.

LLAMA's `input` function requires a particular data format, but places no restrictions on where the data comes from. Its first argument is a data frame that contains the features for each problem instance. Each row in the data frame designates a different problem instance, each column holds the values for a different feature. The second argument to `input` is a similar data frame that contains performance information for the algorithms in the portfolio. Both data frames should have a column that holds the ID of the problem instance such that the two data frames can be merged. In fact, LLAMA assumes that any columns that are present in both data frames can be used to merge them.

The third (and optional) argument is a data frame that tells LLAMA whether the run of a particular algorithm on a particular instance was successful or not. The column names should be the same as for the data frame that holds the algorithm performance values, and there should also be an ID column. Each cell holds a Boolean value designating whether the run was a success or not. The definition of "success" depends on the context; it can for example determine whether an algorithm returned a solution within a certain runtime limit. If this argument is specified, an additional way of evaluating the performance of an algorithm selection model is available. There are no other differences; most of the functionality of LLAMA does not require success values.

Another optional argument can be given to specify the cost of computing the feature values. This overhead incurred by the algorithm selection system needs to be taken into account to provide a realistic performance evaluation of the learned models. There are three different ways of specifying feature costs. A single number is assumed to be the cost for each instance. Alternatively, a data frame with an ID column and a column for each feature can be given. The

entries in the rows denote the cost of computing the respective feature value for the respective instance. The third way of specifying feature computation costs is through a list that specifies feature groups and their costs. More details on how to specify feature costs along with examples can be found in LLAMA's on-line help.

If feature costs are specified, they are automatically taken into account during the evaluation of the learned models. Cost and performance are assumed to be additive – that is, the cost can be added to the performance value. This covers the most common case for algorithm selection where the performance is the runtime. In addition to adding the cost, LLAMA also checks whether, with the overhead included, the system would incur a timeout and takes appropriate action if this is the case.

The final (and again optional) argument tells LLAMA whether low performance values are good or bad. It specifies how LLAMA determines the best algorithm, given the performances of the algorithms on an instance. The default behaviour is to assume that smaller values are better (the values give e.g. runtimes). For the opposite behaviour (e.g. quality of solution), specify `minimize=F`.

Assume that your data is in a set of CSV files with the following format.

```
features.csv:
  ID,width,height
  0,1.2,3
  ...more instances...

performance.csv:
  ID,alg1,alg2
  0,2,5
  ...more instances...

success.csv:
  ID,alg1,alg2
  0,T,F
  ...more instances...
```

You can load this data into LLAMA as follows.

```
data = input(read.csv("features.csv"), read.csv("times.csv"),
    read.csv("success.csv"))
```

The `input` function automatically computes all the meta data required by LLAMA – the return value can be used right away. Full details on the returned structure can be found in the on-line documentation.

3.2.1 Algorithm Selection Benchmark Library

The Algorithm Selection Benchmark Library ASlib[1] offers, at the time of writing, 14 different scenarios that have been used in algorithm selection publications. It defines a data format for these scenarios as well.

LLAMA supports the ASlib data format through the conversion functions provided by the `aslib` package[2] and is used to run the benchmark experiments whose results are presented on the ASlib website.

```
library(aslib)
scenario = parseASScenario("/path/to/scenario/dir")
# convert data format
ldf = convertToLlama(scenario)
folds = cvFolds(ldf)
model = classify(makeLearner("classif.J48"), folds)
# convert data format including specified
# cross-validation splits
folds = convertToLlamaCVFolds(scenario)
model = classify(makeLearner("classif.J48"), folds)
```

3.2.2 Example data

LLAMA comes with some example data that you can play around with as a start. The data is runtime data for 19 SAT solvers on 2433 SAT instances [9]. For each instance, 36 features were measured. Success data (i.e. whether an algorithm timed out or not) is also available, but feature computation costs are not.

To use this data, type

```
data(satsolvers)
```

If you want to run the examples in the remainder of this document with this data, run

```
data = satsolvers
```

3.3 Slicing and dicing the data

Machine learning models are usually trained and tested on separate data. This is to avoid so-called overfitting, where the model learned is so specific to the data it was trained on that the predictions on anything else are very inaccurate, and to provide a realistic estimate of generalisation performance. LLAMA provides functions to split a data set into training and test sets. This is one of the tedious and error-prone steps that researchers have to deal with in practice and that LLAMA aims to make less painful. To split the data into 60% training and

[1] `http://aslib.net`
[2] `https://github.com/coseal/aslib-r`

40% test sets, we can run the following command, assuming that your data is available in the **data** variable.

```
split = trainTest(data)
```

The second (optional) argument of the function specifies what fraction of the total data should be used for training. If, instead of a 60-40 split, we want a 70-30 split, all we need to do is run

```
split = trainTest(data, 0.7)
```

By default, the training and test partitions are not stratified. This means that the data is split randomly regardless of the performance characteristics in the data. To turn stratification on, give the additional argument `stratify = TRUE`. Then, the distribution of best-algorithm labels in the partitions will be approximately equal. If, for example, solver A is the best on 90% of the instances and solver B on the remaining 10% in the training set, the same will be the case in the test set.

In addition to a simple train-test split, LLAMA also provides function to create data folds for cross-validation using bootstrap sampling and cross-fold partitioning [12]. These methods are in general seen as more reliable ways of evaluating the performance of a learning algorithm. The main difference between bootstrap and cross-validation sampling is that in the former case, there is no guarantee on the distribution of the instances (in particular individual instances may appear multiple times across several folds), while cross-validation splits the data into n disjunctive sets. To partition the entire data into 10 folds, run

```
folds = bsFolds(satsolvers)  # bootstrap sampling
folds = cvFolds(satsolvers)  # cross-validation
```

The optional second argument `nfolds` allows to specify the number of folds for both functions. LLAMA transparently takes care of training and evaluating models regardless of how many splits or folds you have.

3.4 Training and evaluating models

Now that we have both training and test data, we can train an algorithm selection model. To start with, we will train a simple classification model that, given the features of an instance, predicts the algorithm to use. This approach is used for example in [5]. We train a model using the C4.5 decision tree learner [18]. For this, we call `classify` with the name of the machine learning algorithm and the data folds created above.

```
model = classify(makeLearner("classif.J48"), folds)
```

For the other available machine learning models, please consult the `mlr` documentation [1].

The call to `classify` trains and tests models on each cross-validation fold. That is, for n folds it trains n models using $n-1$ partitions for training and the remaining partition for testing. The predictions on these testing partitions are

returned along with a prediction function that uses a model that was trained on the *entire*, unsplit data set. While the cross-validation predictions allow to assess the expected performance of the model, the returned prediction function can be used as a building block for a portfolio system to obtain predictions on new data. For full details on the returned data structure, see the on-line help.

LLAMA provides several functions to evaluate the performance of an approach based on the predictions made. The misclassification penalty quantifies how much performance we lose because of prediction mistakes. If the performance is e.g. runtime, it measures how much additional time we need, i.e. the difference in runtime between the algorithm chosen by the model and the best possible choice for each problem instance. Another common performance measure for runtime performances is the PAR (penalized average runtime) score. The score is equal to the time it took the algorithm to solve the instance or, if the algorithm was unable to solve it, a constant factor times the time-out value. Usually, PAR10 is used, meaning that time-outs are penalized by a factor of 10. To compute the average PAR10 score and the total number of solved instances of the approach using the C4.5 decision tree, we can run the following code.

```
# mean misclassification penalty
mean(misclassificationPenalties(folds, model))
## [1] 114.4854
# mean PAR10 score
mean(parscores(folds, model))
## [1] 5891.54
# number of successes (solved instances)
sum(successes(folds, model))
## [1] 2039
```

The evaluation functions take the data for which the predictions were made as their first argument and the model that contains the predictions or the function that returns predictions (in the case of virtual best and single best algorithms) as the second. Optional arguments can be given to specify penalty factors and time-out values where applicable. By default, the performance value given for an unsuccessful run is assumed to be the time-out if the performance is runtime.

All evaluation functions return a list of the respective values for the chosen algorithm for each instance. That is, if there are 100 instances in the data, `misclassificationPenalties` will return a vector of 100 values.

```
head(misclassificationPenalties(folds, model))
## [1] 0.002062 0.000000 0.014942 0.000000 0.000000 0.000000
```

The predictions LLAMA models computes are actually not just simple labels. The prediction object is a data frame that contains information on the predictions on every instance and algorithm – the score. The meaning of the score value depend on the model-building function that is used and are explained in the respective on-line help pages. In this case, the score for only a

single algorithm per instance will be 1, meaning that this is the algorithm that was predicted by the one classifier.

```
head(model$predictions)
##   benchmark_id      algorithm score iteration
## 1          408        picosat     1         1
## 2          408        minisat     0         1
## 3          408  cryptominisat     0         1
## 4          408        glucose     0         1
## 5          408 minisat_noelim     0         1
## 6          408      lingeling     0         1
```

The `iteration` column specifies in which iteration of the cross-validation the prediction was obtained.

Computing metrics like the misclassification penalty or the number of successes doesn't give us a very good idea of how good the approach actually is. In the algorithm selection community, two common approaches to compare against are the virtual best solver and the single best solver. The virtual best solver assumes that we have a perfect predictor that will always choose the best algorithm for a particular instance. It determines the best possible performance a selector can achieve. The single best solver is the algorithm in the portfolio that has overall the best performance, e.g. on the largest number of instances in the data set or with respect to another evaluation criterion.

LLAMA provides convenience functions that allow to compare to both virtual best and single best solver. They are used in the same way as the predictions from a model are.

```
mean(misclassificationPenalties(data, vbs))
## [1] 0
mean(parscores(data, vbs))
## [1] 4631.264
sum(successes(data, vbs))
## [1] 2125
mean(misclassificationPenalties(data, singleBest))
## [1] 122.3186
mean(parscores(data, singleBest))
## [1] 5778.983
sum(successes(data, singleBest))
## [1] 2048
```

Comparing those numbers to the ones from the model that we trained should give us a better idea of its performance. Ideally, the model performance should be better than the one of the single best solver and as close to the virtual best as possible.

There are several different definitions for the single best solver, depending on the performance measure used to determine it. The `singleBest` function

determines it as the one that has the best cumulative performance over all problem instances in the data set. LLAMA also provides functions to determine the single best by PAR score (`singleBestByPar`), by number of problem instances solved (`singleBestBySuccesses`), and by number of instances it delivered the overall best performance on (`singleBestByCount`).

As mentioned above, the model-building function also returns a prediction function that allows to work with new data. As an example, we will use it to make predictions for the data that we have. Note that we're getting predictions for the same data that we used to train the model here – do not use these predictions to evaluate the performance, this example is purely to illustrate what code to run.

```
predictions = model$predictor(subset(data$data,
    TRUE, data$features))
head(predictions)
##   id      algorithm score iteration
## 1  1            mxc     1         1
## 2  1        minisat     0         1
## 3  1  cryptominisat     0         1
## 4  1        glucose     0         1
## 5  1 minisat_noelim     0         1
## 6  1      lingeling     0         1
```

Oh and if you want better model performance than the single best, try the `randomForest` classifier from the quick start.

3.5 Other available model types

Instead of using the `J48` decision tree inducer, we can use any other classification algorithm. The only change needed is to give the other machine learning algorithm as the first argument.

Building a classifier to predict the best solver for a problem is only one of the approaches to providing a selector for algorithm portfolios. A different approach is used for example in older versions of SATzilla [27]. For each solver in the portfolio, a regression model is induced to predict the performance of the solver on a particular problem. Given these predictions, the solver with the best predicted performance is chosen.

LLAMA supports this kind of performance model as well. All we have to do is call a different function and pass in a machine learning algorithm that is able to learn models to predict numeric quantities as an argument.

```
model = regression(makeLearner("regr.lm"), folds)
```

You will notice that running this command takes longer than for the classification example. This is because instead of a single classification model, we now need to train one regression model for each algorithm in the portfolio (if you're using the example SAT data, 19 different models).

The structure returned by the call is the same as for classification and performance scores and similar are calculated in the same way. LLAMA offers a unified interface for all its model-building functions that makes it easy to quickly try different approaches. The difference is that the score in the prediction data frame now denotes the predicted performance value.

```
head(model$predictions)
##   benchmark_id   algorithm     score iteration
## 1          408    precosat -36.65360         1
## 2          408     picosat -34.44392         1
## 3          408    qutersat -34.15245         1
## 4          408 MPhaseSAT64 -33.95013         1
## 5          408 glueminisat -32.67685         1
## 6          408        riss -32.65705         1
```

Apart from the fact that the regression model is predicting negative runtimes, it appears to work quite well. It does not really matter that the values are negative in this case, as we only use them to rank the algorithms.

The `regression` function determines whether the lowest performance value denotes the best algorithm by what has been specified when running `input`.

The approach used in the most recent version of SATzilla is to train classifiers that predict the better algorithm for each pair of algorithms [28]. This approach is a hybrid between the single classification model and the regression approach. Its strength comes from the fact that it explicitly considers the relation between two algorithms. It is usually easier to predict which of a pair of algorithms will be better rather than choosing the best from a large set or predicting the performance for each.

The predictions of the individual classifiers are aggregated as votes and the algorithm that has most votes wins. The number of votes for each algorithm can be used to rank all of the portfolio algorithms. This approach is also implemented in LLAMA. The function is called `classifyPairs` and conforms to the usual interface.

```
model = classifyPairs(makeLearner("classif.J48"),
    folds)
```

Running this command on the example SAT data will take quite a long time, as a model for each pair of algorithms needs to be trained. This approach offers great potential for parallelisation though; for more details, see Section 4.4.

The data frame of predictions now looks as follows; the score corresponds to the number of votes.

```
head(model$predictions)
##   benchmark_id     algorithm score iteration
## 1          408 cryptominisat    93         1
## 2          408       minisat    78         1
## 3         1022 cryptominisat   108         1
## 4         1022       minisat    63         1
```

```
## 5         1280 cryptominisat    94         1
## 6         1280       minisat    77         1
```

The model-building function `regressionPairs` follows the same idea, but uses regression models to predict the performance difference between pairs of algorithms instead of simply whether one is better than the other.

A different approach to algorithm selection that is used for example in ISAC [11] is to cluster the training problem instances and assign the best algorithm to each cluster based on the algorithm performances on the instances in the cluster. Again the only change is to call a different function, this time the `cluster` function with the `XMeans` clustering algorithm.

```
model = cluster(makeLearner("cluster.XMeans"),
    folds)
```

The return value corresponds to the usual format. The prediction data frame contains all portfolio algorithms ranked by performance. The score corresponds to the sum of the performances over all training instances in the respective cluster.

```
head(model$predictions)
##   benchmark_id     algorithm    score iteration
## 1          408         clasp 446.3364         1
## 2          408 cryptominisat 471.5320         1
## 3          408      qutersat 479.6736         1
## 4          408           mxc 522.1233         1
## 5          408       glucose 600.1904         1
## 6          408      march_rw 644.0489         1
```

The `cluster` model-builder provides different ways of determining the best algorithm for a cluster that correspond to the different ways of determining the single best algorithm. The method to use is determined by the `bestBy` argument, which defaults to "performance".

4 Advanced functionality

The previous section gave a glimpse of the core functionality of LLAMA. There is much more functionality beyond that though. All the functions we have used previously take additional arguments that allow them to be customised. The model building functions can work with several machine learning algorithms instead of just one. There are more functions that do exciting things[3].

4.1 Processing the input data

The feature and performance data is often messy – there are missing values, the values of some of the features are the same on all instances, or there is

[3]For suitable definitions of "exciting".

no correlation between feature values and performance. All of this can impact the performance of machine learning models. LLAMA provides functionality to mitigate this.

4.1.1 Normalising feature values

For some types of models, it may be desirable to normalise the feature values such that they cover the same range over all features. If, for example to cluster the instances, we compute the distance between two instances based on the feature values in Euclidean space and the values for a particular feature happen to be 1000 times larger than the other ones, this feature will have the highest impact on the result, even though it may not be important.

LLAMA provides a function that allows to normalise the values of features before they are passed to the model learner. To normalise the feature values, scaling factors need to be computed. These same scaling factors need to be applied when working with new data, i.e. when using the predictor function returned by the model builders.

This is why normalisation is implemented as an optional argument to the model-building functions instead of a standalone functions. The scaling factors are computed for the training data and saved in the environment such that they can be applied to new data later.

LLAMA currently provides only a single function, `normalize`, for feature value normalisation. This function scales the feature values such that the range for all features is -1 to 1. It is specified through the `pre` argument.

```
model = cluster(makeLearner("cluster.XMeans"),
    folds, pre = normalize)
mean(misclassificationPenalties(folds, model))
## [1] 104.3893
mean(parscores(folds, model))
## [1] 5721.646
sum(successes(folds, model))
## [1] 2051
```

We are using the cluster model builder here, as clustering is an application that intrinsically relies on distance measures and is likely to be most affected by large differences in the range of feature values. The feature values can be normalized for all of the other model-building functions in the same way.

4.1.2 Imputing censored runtimes

Working with empirical performance data is often difficult. If the problem instances are challenging, some of the algorithms may take a very long time to solve them – longer than one is prepared to wait. Usually, algorithms are run with a time-out. That is, if the algorithm did not find a solution after a certain amount of time, it is terminated – its runtime is *censored*. While this allows to gather data in more reasonable amounts of time, the result makes machine

learning more difficult – if an algorithm timed out, the recorded runtime is not actually the value we want to predict.

One way of addressing this issue is to impute the censored runtimes by learning a machine learning model to predict the runtime on the instances that did not time out and then apply it to the instances that timed out. This process can be repeated to get better models and estimates [21].

This method is implemented in LLAMA in the `imputeCensored` function. Its arguments are a LLAMA data frame, the regression algorithm to model the runtime, and termination conditions. It returns a new data frame with the imputed censored runtimes. Note that, similar to the regression model learner, the imputation function does not check the plausibility of the results – it is possible that the predicted runtimes are less than the time-out!

```
imputed = imputeCensored(data, makeLearner("regr.lm"))
imputedFolds = cvFolds(imputed)
imputedModel = regression(makeLearner("regr.lm"),
    imputedFolds)
mean(misclassificationPenalties(folds, model))
## [1] 104.3893
mean(parscores(folds, model))
## [1] 5721.646
sum(successes(folds, model))
## [1] 2051
mean(misclassificationPenalties(imputedFolds,
    imputedModel))
## [1] 44.84682
mean(parscores(imputedFolds, imputedModel))
## [1] 74.07171
sum(successes(imputedFolds, imputedModel))
## [1] 2433
```

The performance of the new model with imputed performance values is much better in terms of average PAR10 and number of solved instances, but this is to be expected – all instances in the data are "solvable" after imputation, so no penalties will be imposed. However, we can compare the mean misclassification penalty for the old and new models – there is a clear improvement there as well.

4.2 Portfolio analysis

LLAMA provides the `contributions` function to analyse the contributions a single algorithm makes to a portfolio using techniques from game theory [19]. Running this on the example data distributed with the package gives the following result:

```
sort(contributions(satsolvers), decreasing = TRUE)
```

```
##          clasp  cryptominisat       qutersat        glucose
##     42499.1089     8854.7117      -740.2909     -4678.7110
##            mxc       march_rw    MPhaseSAT64       precosat
##    -28069.9128    -33521.0780    -44008.3382    -54136.9820
##        picosat    glueminisat      lingeling      contrasat
##    -64197.8563    -67935.5264    -76489.3100    -78365.7487
##          sat4j           riss     cirminisat        minisat
##    -82804.3613    -88360.4072    -89550.5715    -99151.8748
## minisat_noelim           rsat          kcnfs
##   -108837.5158   -180762.3266   -238407.9801
```

The negative numbers designate solvers that do not contribute anything to the portfolio, but causes its performance to decrease. The solver `clasp` makes the largest contribution.

This analysis can be used to determine which algorithms should be included and which ones should be omitted from a portfolio. Note that it is entirely separate from learning a model to make the predictions necessary to choose a solver from the portfolio for a particular problem instance – the analysis is done for the virtual best portfolio. As such, the theoretical best portfolio might not be the best portfolio to learn models for. To achieve the best performance in practice, the combination of portfolio and algorithm selection model has to be evaluated.

4.3 Ensembles and stacking

One of the main strengths of LLAMA is that meta-learning methods, for example ensembles and stacking, can be applied to all of the model building functions easily. The idea behind these techniques is similar to that of algorithm portfolios – instead of relying on a single machine learning algorithm and the model it learns to deliver good predictions, we use several that (hopefully) complement each other.

The two implemented concepts are ensembles [4] and stacking [23]. In ensemble learning, multiple machine learning algorithms are run on the same data, learning multiple independent models. The predictions of each model are then combined to determine the overall prediction. In stacking on the other hand, several machine learning algorithms are layered on top of each other. That is, the first layer learns a model based on the actual data, while the second layer takes the predictions of the first layer as input. The two approaches can be combined – the predictions of an ensemble may be used as the input to a second layer of machine learning that makes the final prediction.

For the `classify` model building functions, both ensemble learning and stacking have been implemented. To use an ensemble, the first argument becomes a list of classification algorithms instead of a single one. To use stacking, that list should have a member named `.combine`.

Here's an example with an ensemble that contains a `J48` decision tree, a CART tree, and a nearest neighbour classifier. In the second example, the

predictions of these classifiers are combined using another `J48` decision tree.

```
ensembleModel = classify(list(makeLearner("classif.J48"),
    makeLearner("classif.rpart"), makeLearner("classif.knn")),
    folds)
stackedEnsembleModel = classify(list(makeLearner("classif.J48"),
    makeLearner("classif.rpart"), makeLearner("classif.knn"),
    .combine = makeLearner("classif.J48")), folds)
mean(misclassificationPenalties(folds, model))
## [1] 104.3893
mean(parscores(folds, model))
## [1] 5721.646
sum(successes(folds, model))
## [1] 2051
mean(misclassificationPenalties(folds, ensembleModel))
## [1] 101.2529
mean(parscores(folds, ensembleModel))
## [1] 5825.041
sum(successes(folds, ensembleModel))
## [1] 2043
mean(misclassificationPenalties(folds, stackedEnsembleModel))
## [1] 144.1736
mean(parscores(folds, stackedEnsembleModel))
## [1] 6134.288
sum(successes(folds, stackedEnsembleModel))
## [1] 2023
```

Compared to the model based on the single `J48` classifier, ensemble learning improves performance a bit, but stacking does not help.

The prediction data frame for ensemble models has the number of votes for each algorithm as score.

```
head(ensembleModel$predictions)
##   benchmark_id      algorithm score iteration
## 1          408          clasp     2         1
## 2          408        picosat     1         1
## 3          408        minisat     0         1
## 4          408  cryptominisat     0         1
## 5          408        glucose     0         1
## 6          408 minisat_noelim     0         1
```

For regression models, stacking as described in [14] is implemented. Instead of aggregating the predicted performance values by ranking them directly, a classifier is learned to predict, given the performance predictions, the best algorithm. To achieve this, a classification algorithm is given as the `combine`

argument.

```
stackedModel = regression(makeLearner("regr.lm"),
    folds, combine = makeLearner("classif.J48"))
mean(misclassificationPenalties(folds, model))
## [1] 104.3893
mean(parscores(folds, model))
## [1] 5721.646
sum(successes(folds, model))
## [1] 2051
mean(misclassificationPenalties(folds, stackedModel))
## [1] 128.6861
mean(parscores(folds, stackedModel))
## [1] 5985.637
sum(successes(folds, stackedModel))
## [1] 2033
```

On the example data, the performance of the stacked model is worse than for the non-stacked version. The reason for this is that for the combination function, choosing the algorithm with the lowest predicted value is very hard to learn when considering only the performance values independently and not the relation between them.

LLAMA does support an additional argument for `regression` that allows to make this task easier. The **expand** argument allows to specify a function that, given the performance predictions, can augment the inputs to the classifier. In this case, we want to add the pairwise absolute differences between the predicted performances of algorithms.

```
stackedExpandedModel = regression(makeLearner("regr.lm"),
    folds, combine = makeLearner("classif.J48"),
    expand = function(x) {
        cbind(x, combn(c(1:ncol(x)), 2, function(y) {
            abs(x[, y[1]] - x[, y[2]])
        }))
    })
mean(misclassificationPenalties(folds, stackedModel))
## [1] 128.6861
mean(parscores(folds, stackedModel))
## [1] 5985.637
sum(successes(folds, stackedModel))
## [1] 2033
mean(misclassificationPenalties(folds, stackedExpandedModel))
## [1] 137.5572
mean(parscores(folds, stackedExpandedModel))
```

```
## [1] 6127.672
sum(successes(folds, stackedExpandedModel))
## [1] 2023
```

While the function given here may appear cryptic at first, it demonstrates one o the advantages of the implementation of LLAMA as an R package. It allows to use arbitrary R functions to process data. This version achieves an improvement over the stacked model without expansion.

For the `classifyPairs` and `regressionPairs` model builders, LLAMA supports stacking in the same way as for `regression`. Stacking can be used to provide a classification algorithm in the `combine` argument that learns to predict the best algorithm, given the predictions of the underlying machine learning models for pairs of algorithms.

The `cluster` model builder supports these techniques in the same way as `classify`. More details and examples can be found in the on-line documentation for each of the model builders.

4.4 Parallel execution

Some of the models can take a very long time to train. Most of the operations are independent though and can be parallelised easily. LLAMA uses the `parallelMap` construct from the parallelMap package[4] to parallelise execution across cross-validation folds. That is, the models for each iteration will be trained and tested in parallel. The `parallelMap` construct provides transparent parallelisation that executes sequentially if no suitable parallel backend is loaded. All the user has to do to enable parallel execution is to load a parallel backend, for example through `parallelStartSocket`. Libraries used in the spawned processes should be specified through `parallelLibrary`.

```
library(parallelMap)
parallelStartSocket(2)  # use 2 CPUs
parallelLibrary("llama", "mlr")
# LLAMA code
parallelStop()
```

After running these commands, all subsequent calls to LLAMA model building functions will be parallelised across 2 CPUs.

Note that the functions provided by RWeka rely on a Java interface that is not thread-safe. The Weka machine learning algorithms can still be used with parallel execution though if a backend is used that runs separate processes, such as the sockets backend used above.

4.5 Instance weights

In algorithm selection, not all problem instances are equal. On some of them, it doesn't really matter which algorithm we choose, either because the performance

[4] `https://github.com/berndbischl/parallelMap`

difference is small, or because the performance is so good (e.g. the runtime is so low) that in practice making a wrong decision won't matter that much. On the other hand, there are instance that we really do want to get right because of massive differences in performance that matter a lot in practice.

LLAMA supports instance weights for all selection model building functions except `cluster`, where such weights cannot be incorporated in a meaningful way. If the underlying machine learning algorithm supports case weights, each instance has the performance difference between the best and the worst algorithm attached to it. This happens by default, but you can explicitly disable weights with the `use.weights = FALSE` argument to the model building functions.

5 Visualising the data

This part is going to be a bit more graphic than the previous rather dry sections – we are going to have a look at visualising the data we have been working with. R offers many possibilities for doing so and surveying them all is far beyond the scope of this manual. We only give a flavour of what visualisations can be done.

A simple way of "visualising" predictions is to simply print a table of how many times each algorithm was predicted. LLAMA provides the function `predTable` for this purpose. It takes the list of predictions as argument.

```
predTable(vbs(satsolvers))
## algorithms
##  cryptominisat minisat_noelim        picosat         glucose
##            377            356            293             289
##       march_rw          clasp            mxc     cirminisat
##            267            215            168              83
##    glueminisat        minisat    MPhaseSAT64       precosat
##             70             53             45              40
##       qutersat      contrasat      lingeling           rsat
##             33             32             32              30
##           riss          kcnfs          sat4j
##             21             15             14
predTable(model$predictions)
## algorithms
##         clasp       glucose cryptominisat
##          1778           437           218
```

5.1 LLAMA's plotting functions

LLAMA provides visualisation functionality that is tailored for algorithm selection, but this is by no means the only to plot and try to make sense of the data. In particular, it provides the function `perfScatterPlot`, which allows you to

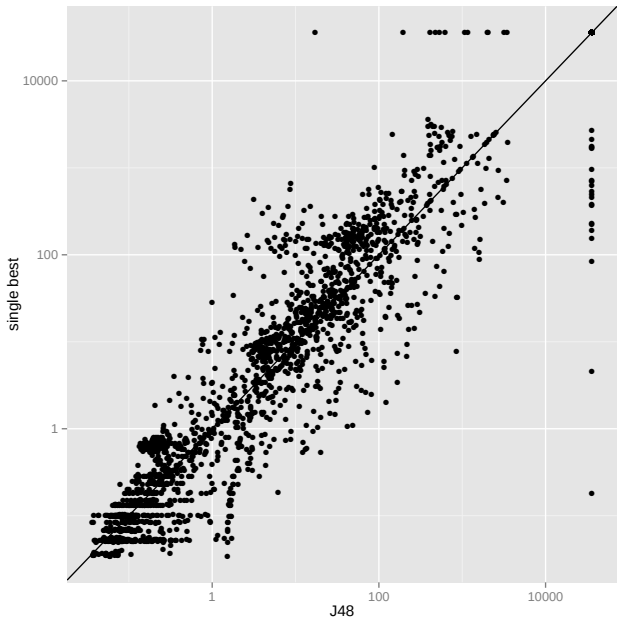


Figure 2: PAR10 performance comparison between J48 decision tree selection model and single best.

plot a scatter plot that compares the performance of two selectors. What is does is best illustrated through an example.

```
model = classify(makeLearner("classif.J48"), folds)
perfScatterPlot(parscores, model, singleBest,
    folds, satsolvers) + scale_x_log10() + scale_y_log10() +
    xlab("J48") + ylab("single best")
```

The first argument to `perfScatterPlot` is the metric to use for evaluation, the second and third the models to compare (which may be functions such as `singleBest`), and the last two arguments the data to evaluate the respective models on. Here, we need two different data sets as the learned model should be evaluated on the split data, while the single best is computed on the entire data. If both models are of the same type and should be evaluated on the same data, it needs to be given only once.

The resulting plot can be seen in Figure 2. Each point corresponds to a problem instance in the dataset. Points on the line mean that both selectors (the `J48` decision tree classifier selection model and the single best algorithm) achieved the same performance. Points below the diagonal mean that our trained selector was better and above the diagonal the single best algorithm.

Scatter plots like this are useful for inspecting what a selection model does and where its weaknesses lie. LLAMA allows to pass additional information

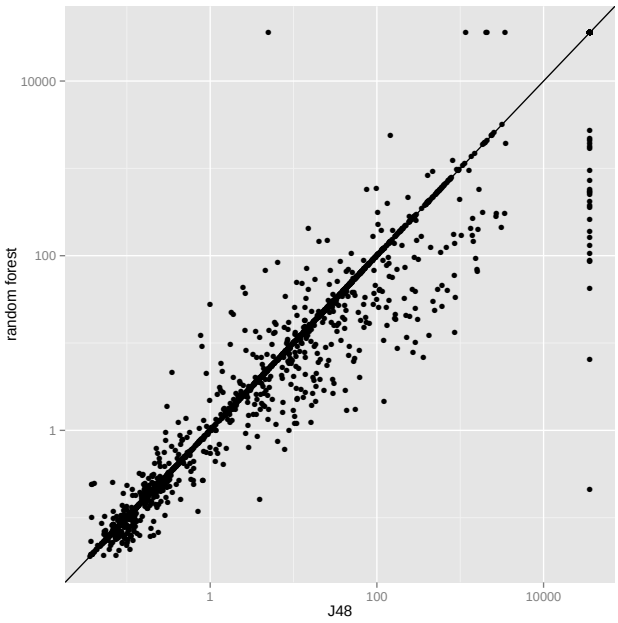


Figure 3: PAR10 performance comparison between J48 decision tree and random forest selection models.

that can be used to e.g. change the shape or colour of the points according to the group the problem instance belongs to. Internally, `ggplot2` is used for plotting and its power is available for customising the plot. The `ggplot` object is returned from the function to allow further modifications.

Figure 3 shows another scatter plot that compares the performance of two "real" models instead of a model and a baseline.

```
model1 = classify(makeLearner("classif.J48"),
    folds)
model2 = classify(makeLearner("classif.randomForest"),
    folds)
perfScatterPlot(parscores, model1, model2, folds) +
    scale_x_log10() + scale_y_log10() + xlab("J48") +
    ylab("random forest")
```

5.2 Other ways of plotting data

It is always useful to have a look at the data one will be working with. LLAMA does not provide any functions for visualisation of the raw data, but it is easy enough to do in R. In our case, there are two main groups of data – the performance values and the features. Let's have a look at the performance values

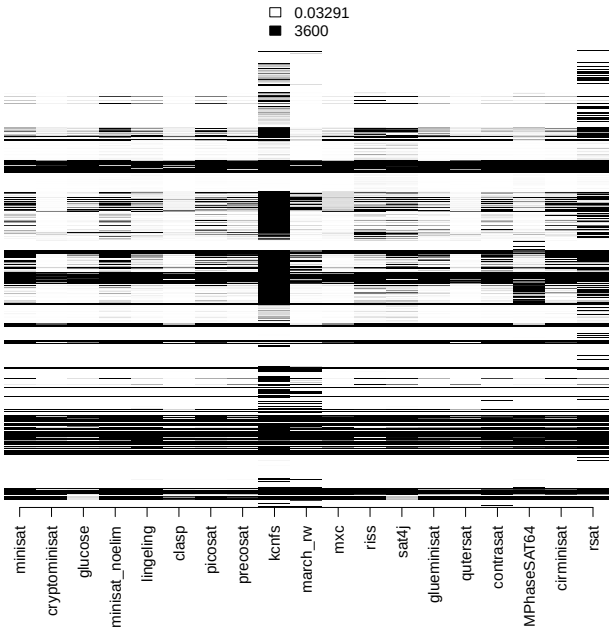


Figure 4: Runtimes by solver and instance from almost instantaneous solve (white) to timeout (black).

first. Throughout this section, we will be using the example SAT data, but the same methodology applies to any other data in LLAMA format. As we have a large amount of data to visualise, we are going to use heatmaps.

```
data(satsolvers)
times = subset(satsolvers$data, T, satsolvers$performance)
par(mar = c(7, 1, 3, 1))
cols = gray(seq(1, 0, length.out = 255))
image(t(as.matrix(times)), axes = F, col = cols)
axis(1, labels = satsolvers$performance, at = seq(0,
    1, 1/(length(satsolvers$performance) - 1)),
    las = 2)
legend("top", legend = c(min(times), max(times)),
    fill = c("white", "black"), bty = "n", inset = -0.12,
    xpd = NA)
```

The resulting plot is shown in Figure 4. After extracting the performance values (runtimes in this case) from the LLAMA data frame, setting some plot parameters and creating a black and white colour scale, we plot the heatmap, axis, and legend. The map shows the time each solver takes on each instance.

There is a large spread of runtimes in our data, with most instances being solved either instantaneously or timing out. The figure looks accordingly, with

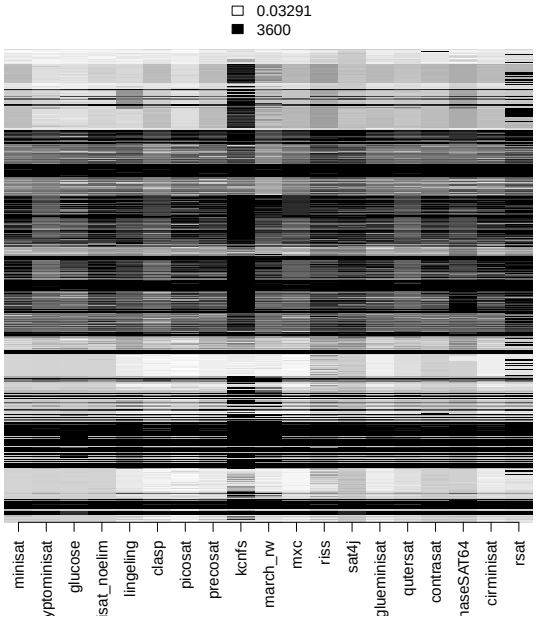


Figure 5: Log runtimes by solver and instance from almost instantaneous solve (white) to timeout (black).

most colours being either white or black and very little grey in between. Plotting the log of the runtime is more informative (Figure 5).

```
image(log10(t(as.matrix(times))), axes = F, col = cols)
axis(1, labels = satsolvers$performance, at = seq(0,
    1, 1/(length(satsolvers$performance) - 1)),
    las = 2)
legend("top", legend = c(min(times), max(times)),
    fill = c("white", "black"), bty = "n", inset = -0.12,
    xpd = NA)
```

When doing algorithm selection, we are usually more interested in how the performance of a solver compares to the other solvers in the portfolio rather than the absolute value. We can plot the rank of each solver on each instance in a similar fashion to before (Figure 6).

```
image(apply(times, 1, order), axes = F, col = cols)
axis(1, labels = satsolvers$performance, at = seq(0,
    1, 1/(length(satsolvers$performance) - 1)),
    las = 2)
legend("top", legend = c(1, length(satsolvers$performance)),
    fill = c("white", "black"), bty = "n", inset = -0.12,
```

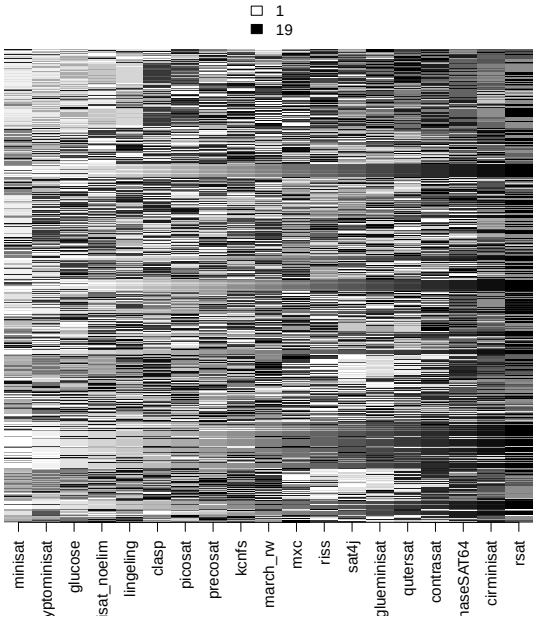


Figure 6: Solver ranks on each instance from first (white) to last (black).

```
    xpd = NA)
```

The same kind of visual analysis can be applied to the features. This is probably even more important that analysing the performance data, as the features are used to create the models we are going to use for algorithm selection. If the features are bad, the models will not be good either. Let's do the same kind of plot we did for the solvers for the features.

```
features = subset(satsolvers$data, T, satsolvers$features)
par(mar = c(10, 1, 3, 1))
image(t(as.matrix(features)), axes = F, col = cols)
axis(1, labels = satsolvers$features, at = seq(0,
    1, 1/(length(satsolvers$features) - 1)), las = 2)
legend("top", legend = c(min(features), max(features)),
    fill = c("white", "black"), bty = "n", inset = -0.12,
    xpd = NA)
```

The resulting plot is shown in Figure 7. The values for four of the features are much higher than the rest. It is hard to see the variance of the feature values for a particular feature over the set of instances because of this.

We get a better plot from the normalised feature values (Figure 8).

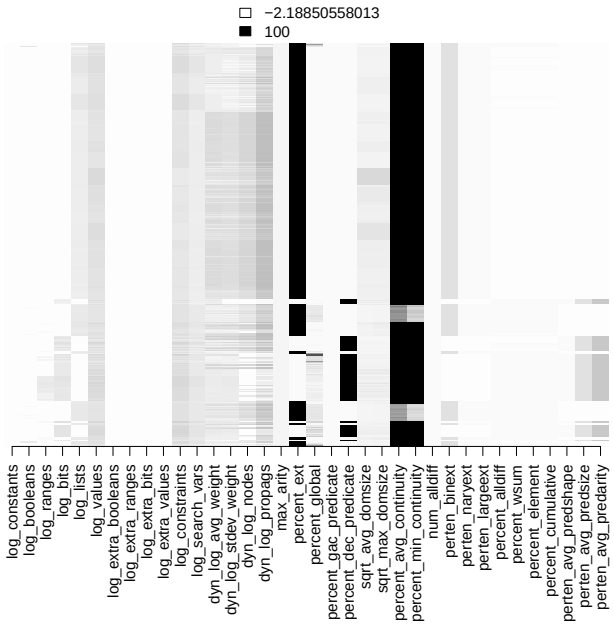


Figure 7: Feature values on each instance from lowest (white) to highest (black).

```
nFeatures = normalize(features)
par(mar = c(10, 1, 1, 1))
image(t(as.matrix(nFeatures$features)), axes = F,
    col = cols)
axis(1, labels = satsolvers$features, at = seq(0,
    1, 1/(length(satsolvers$features) - 1)), las = 2)
```

The features for the example data are not very good. For quite a lot of them, there is almost no variation and others have only two different values.

We can apply the same kind of analysis to the prediction results. This allows us to get an idea of what the machine learning algorithms are doing, where they work well and where things could be improved. The following code creates a heatmap of the PAR10 scores of a classification model on all instances. We are using the log of the PAR10 scores to get more shades of grey.

```
folds = cvFolds(satsolvers)
model = classify(makeLearner("classif.J48"), folds)
scores = parscores(folds, model)
par(mar = c(1, 1, 1, 10))
image(log10(t(as.matrix(scores))), axes = F, col = cols)
legend("right", legend = quantile(scores), fill = gray(seq(1,
    0, length.out = 5)), bty = "n", inset = -0.3,
```

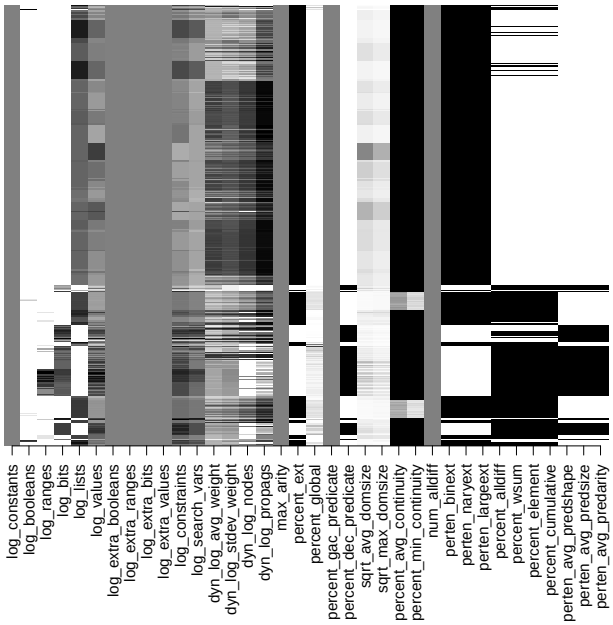


Figure 8: Normalised feature values on each instance from -1 (white) to 1 (black).

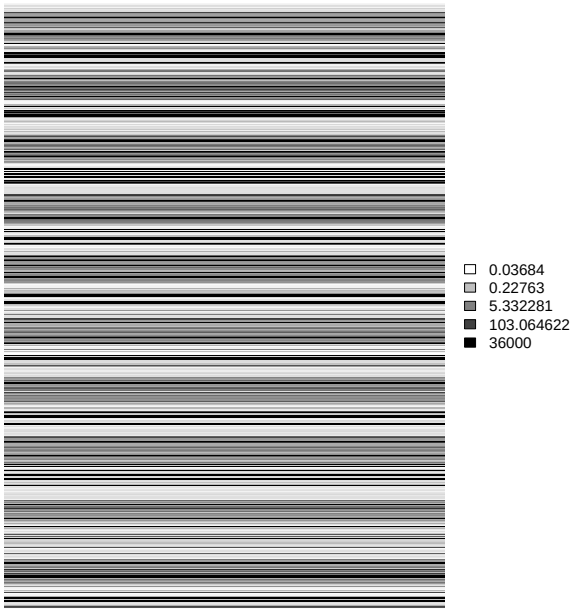


Figure 9: PAR10 scores for a extttJ48 classification model.

```
    xpd = NA)
```

The resulting plot (Figure 9) gives us some idea of what's going on, but looks more like a bar code than anything else.

The power of heatmaps comes from being able to compare multiple entities. A more useful plot compares the performance of the classification model to the performance of other models. The following code can be used to generate such a plot (Figure 10).

```
folds = cvFolds(satsolvers)
model1 = classify(makeLearner("classif.J48"),
    folds)
model2 = regression(makeLearner("regr.lm"), folds)
model3 = classifyPairs(makeLearner("classif.J48"),
    folds)
model4 = regressionPairs(makeLearner("regr.lm"),
    folds)
model5 = cluster(makeLearner("cluster.XMeans"),
    folds)
scores1 = parscores(folds, model1)
scores2 = parscores(folds, model2)
scores3 = parscores(folds, model3)
```

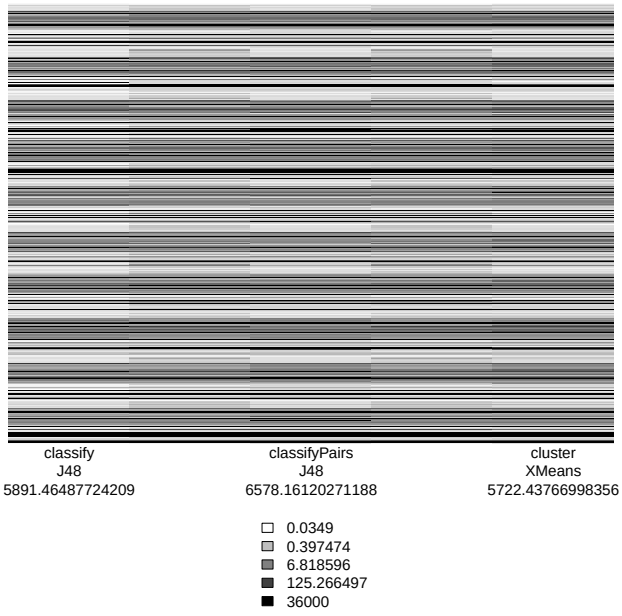


Figure 10: PAR10 scores for five different models. The number in each label shows the mean PAR10 score across all instances.

```
scores4 = parscores(folds, model4)
scores5 = parscores(folds, model5)
par(mar = c(10, 1, 1, 1))
names = c(paste("classify", "J48", mean(scores1),
    sep = "\n"), paste("regression", "lm", mean(scores2),
    sep = "\n"), paste("classifyPairs", "J48",
    mean(scores3), sep = "\n"), paste("regressionPairs",
    "lm", mean(scores4), sep = "\n"), paste("cluster",
    "XMeans", mean(scores5), sep = "\n"))
scores = c(scores1, scores2, scores3, scores4,
    scores5)
image(log10(t(matrix(scores, ncol = 5))), axes = F,
    col = cols)
axis(1, labels = names, at = seq(0, 1, 1/(length(names) -
    1)), las = 1, tick = F, line = 1)
legend("bottom", legend = quantile(scores), fill = gray(seq(1,
    0, length.out = 5)), bty = "n", inset = -0.4,
    xpd = NA)
```

This comparison is actually quite insightful. The predictions of the regression model usually incur a higher PAR10 score (overall, the column is darker than

the others), but its mean PAR10 score is lower than for the other models. It also shows us that for this specific scenario, there is almost no difference between the single and pairwise classification models.

As a final example, we will plot the differences in predicted and actual rank for the regression model (Figure 11).

```
par(mar = c(7, 1, 1, 5))
ranks = t(ddply(model$predictions, .(benchmark_id),
    function(x) {
        as.vector(sapply(x$algorithm, function(z) {
            which(z == satsolvers$performance)
        }))
    })[-1])
diffs = apply(times, 1, order) - ranks
cols = c(colorRampPalette(c("red", "white"))(128),
    colorRampPalette(c("white", "blue"))(128))
breaks = c(seq(min(diffs), -1/128, length.out = 128),
    0, seq(1/128, max(diffs), length.out = 128))
image(diffs, axes = F, col = cols, breaks = breaks)
axis(1, labels = satsolvers$performance, at = seq(0,
    1, 1/(length(satsolvers$performance) - 1)),
    las = 2)
legend("right", legend = quantile(diffs), fill = cols[quantile(1:length(cols))],
    bty = "n", inset = -0.15, xpd = NA)
```

The plot is a bit messy, but there are some insights to be gained. Looking at the solvers that perform well overall (e.g. minisat, leftmost column), we can see that they are predominantly red, i.e. most of the time they are predicted to be worse than they actually are. For the solvers that exhibit bad performance (e.g. rsat, rightmost column), we observe that they are predominantly blue, meaning that they are usually predicted to be better than they actually are. This indicates that improvements are to be had by training better machine learning models in this particular scenario.

The examples above only scratch the surface of the visualisations that are possible in R. While none of the examples shown depends on LLAMA, the data structures it provides facilitate the creation of such plots to some extent.

6 Case study: SATzilla

Algorithm selection systems often have additional components for which there is no explicit support in LLAMA. This is not necessarily an obstacle though, much additional functionality can be achieved by partitioning the data appropriately and using other functionality that R provides.

This section outlines how to implement a SATzilla-like system with the help of LLAMA and R. A full implementation is beyond the scope of this document; instead, this section serves as a guide to researchers wishing to implement such

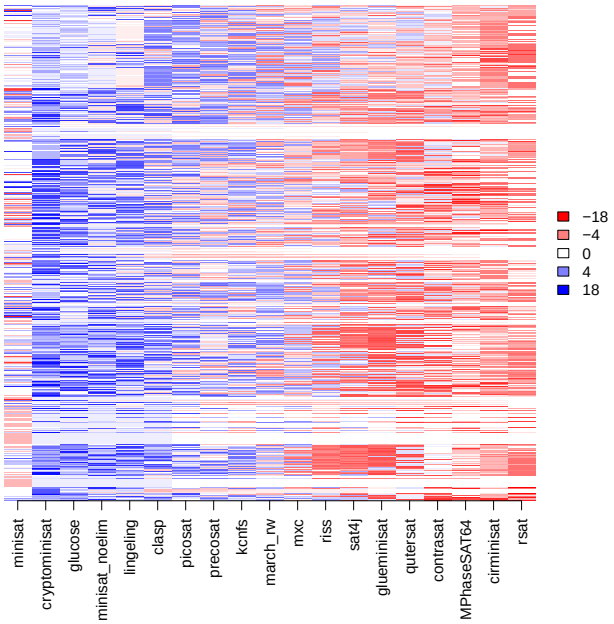


Figure 11: Difference between actual and predicted rank for the regression model. Red means that an algorithm was predicted to be worse than it actually was, blue that it was predicted to be better. Predicted and actual rank agree for white.

measures. There are several techniques that the various versions of SATzilla use to make algorithm portfolios more performance in practice. We will sketch the implementation of each in turn before putting it all together. The techniques used in this section are described in more details in the SATzilla papers (e.g. [25, 27, 28]).

The main difference of the implementation sketched here and SATzilla is that we treat all problem instances the same way, without computing any weights that determine the "importance" of the problem instance. In practice, using weights is almost always a good idea.

6.1 Presolver

There are problem instances that can be solved almost instantaneously. For these, the overhead of computing features, running the algorithm selection model, and selecting a solver is too high. Using a portfolio is always going to be slower for these problems. To avoid this issue, we can run a presolver for a short amount of time.

This step is largely independent of training algorithm selection models. We do however need to take into account that the problems that are solved by the presolver within its time limit do not require algorithms to be selected for them, so we should remove them from the data before training the model. This is done easily in R.

```
performanceData = read.csv("performance.csv")
presolveLimit = 1
presolver = "minisat"
newPerformanceData = subset(performanceData, performanceData[presolver] >
    presolveLimit)
data = input(read.csv("features.csv"), newPerformanceData)
```

Assuming that our presolver is minisat with a time-out of 1 second, we can filter the data as above before passing it to `input`. There is no need to filter the feature data, as LLAMA will discard any items it cannot match. The resulting LLAMA data frame can be used as usual to train and evaluate models.

6.2 Prediction of satisfiability

SAT solvers often behave differently depending on whether an instance is satisfiable or not. In practice, it can make sense to distinguish between these instances in algorithm portfolios. This requires a larger number of machine learning models. First, we need to be able to predict whether a given instance is satisfiable or not. Second, we need different algorithm selection models for the satisfiable and unsatisfiable cases.

To achieve this, we require additional data on whether an instance is satisfiable or not.

```
performanceData = read.csv("performance.csv")
featureData = read.csv("features.csv")
```

```
featureNames = names(featureData)[-1]
satisfiable = read.csv("satisfiable.csv")
satisfiableTask = makeClassifTask("satisfiable",
    target = "satisfiable", data = cbind(satisfiable,
        subset(featureData, T, featureNames)))
satisfiableModel = train(makeLearner("classif.J48"),
    task = satisfiableTask)
satisfiableData = input(featureData, subset(performanceData,
    satisfiable$satisfiable))
unsatisfiableData = input(featureData, subset(performanceData,
    !satisfiable$satisfiable))
```

For simplicity, we assume that the first column of the feature file is an ID and that the order of the instances in the feature, performance, and satisfiable files is the same. The `satisfiableModel` can now be used to predict the satisfiability of a new instance, while the LLAMA data frames `satisfiableData` and `unsatisfiableData` can be used to train and evaluate algorithm selection models for the respective parts of the instance space.

6.3 Prediction of feature computation time

Computing the features of an instance is integral to algorithm selection systems. This information can be related to the performance of the algorithms to make predictions as to which algorithm to use in a particular case. However, sometimes just computing the features can take longer than it would take to solve the instance. It is clearly desirable to identify these cases before starting to compute the features.

To achieve this, we can train yet another machine learning model that, given a small and cheaply computable subset of the features, predicts the time required to compute the remaining features. If the time is too high, we simply run a backup solver on the instance. Such instances should not be used in the training and evaluation of the algorithm selection model.

```
featureTimes = read.csv("featureTimes.csv")
reducedFeatureData = read.csv("reducedFeatures.csv")
reducedFeatureNames = names(reducedFeatureData)[-1]
featureTimeTask = makeRegrTask("featureTime",
    target = "time", data = cbind(featureTimes,
        subset(reducedFeatureData, T, reducedFeatureNames)))
featureTimeModel = train(makeLearner("regr.lm"),
    task = featureTimeTask)
featureTimeLimit = 5
data = subset(subset(read.csv("features.csv"),
    featureTimes$time < featureTimeLimit), read.csv("performance.csv"))
```

Similar assumptions as in the section above are made. The LLAMA data frame can now again be used in the usual manner to train algorithm selection

models.

6.4 Putting it all together

The techniques outlined in the previous section can now be put together to implement a system similar to SATzilla.

```
performanceData = read.csv("performance.csv")
featureData = read.csv("features.csv")
featureNames = names(featureData)[-1]
reducedFeatureData = read.csv("reducedFeatures.csv")
reducedFeatureNames = names(reducedFeatureData)[-1]
featureTimes = read.csv("featureTimes.csv")
satisfiable = read.csv("satisfiable.csv")
presolveLimit = 1
presolver = "minisat"
featureTimeLimit = 5
toSolve = (performanceData[presolver] > presolveLimit) &
    (featureTimes$time < featureTimeLimit)
featureTimeTask = makeRegrTask("featureTime",
    target = "time", data = cbind(featureTimes,
        subset(reducedFeatureData, T, reducedFeatureNames)))
featureTimeModel = train(makeLearner("regr.lm"),
    task = featureTimeTask)
satisfiableTask = makeClassifTask("satisfiable",
    target = "satisfiable", data = cbind(satisfiable,
        subset(featureData, T, featureNames)))
satisfiableModel = train(makeLearner("classif.J48"),
    task = satisfiableTask)
satisfiableData = input(featureData, subset(performanceData,
    satisfiable$satisfiable & toSolve))
unsatisfiableData = input(featureData, subset(performanceData,
    (!satisfiable$satisfiable) & toSolve))
satisfiableFolds = cvFolds(satisfiableData)
unsatisfiableFolds = cvFolds(unsatisfiableData)
satisfiableModel = classifyPairs(makeLearner("classif.randomForest"),
    satisfiableFolds)
unsatisfiableModel = classifyPairs(makeLearner("classif.randomForest"),
    unsatisfiableFolds)
```

Let's walk through it step by step. First, we read all the relevant data, as in the individual steps above. Then we set our time limits and the presolver. The data we want to train algorithm selection models for comprises the instances that are not solved by the presolver and for which the feature prediction time is not too long. We save the Boolean mask that encodes these conditions in `toSolve` to be able to filter the data later.

After that, we train our first two models – one to predict the feature com-

putation time and the other to predict whether an instance is satisfiable or not. For the feature computation time, we need to consider all the data, whereas for the prediction of satisfiability we only need the instances that we want algorithm selection models for.

Then, we partition the data based on satisfiability and pass them to `input`, filtering the instances that we do not require algorithm selection for. Finally, we proceed in the usual LLAMA manner by partitioning the data into folds and learning a SATzilla 2012-style pairwise classification model.

To make predictions on new data, we would use the two preliminary models to determine feature computation time and satisfiability and then the `predictor` member of either the `satisfiableModel` or the `unsatisfiableModel`.

Our SATzilla-style solver now looks something like this.

```
myPortfolioSolver = function(instance) {
    presolver = "minisat"
    presolveTimeout = 1
    backupsolver = "clasp"
    featureTimeout = 5
    result = runSolver(instance, presolver, timeout = presolveTimeout)
    if (!isSolved(result)) {
        simpleFeatures = getSimpleFeatures(instance)
        featureTime = predict(featureTimeMode,
            newdata = simpleFeatures)$data$response
        if ((featureTime > featureTimeLimit)) {
            return(runSolver(instance, backupsolver))
        } else {
            features = getFeatures(instance)
            satisfiable = predict(satisfiableModel,
                newdata = features)$data$response
            if (satisfiable) {
                solver = satisfiableModel$predictor(features)
            } else {
                solver = unsatisfiableModel$predictor(features)
            }
            return(runSolver(instance, solver))
        }
    } else {
        return(result)
    }
}
```

7 The further domestication of LLAMA

The functionality currently implemented in LLAMA facilitates the exploration of the performance of many different approaches to algorithm selection. It is

our intention for LLAMA to develop into a platform that not only facilitates the exploration and comparison of different existing approaches, but also the rapid prototyping of new approaches. The functions it provides take care of the infrastructure required to train and evaluate machine learning models in a scientifically rigorous way. Researchers wishing to improve algorithm portfolio selection do not need to concern themselves with the infrastructure, but can concentrate on the actual research.

LLAMA is maturing rapidly and has a comprehensive test suite. Nevertheless, the responsibility of interpreting and validating the results lies with the user.

It has an issue tracker at `https://bitbucket.org/lkotthoff/llama/issues` for bug reports and feature requests.

Acknowledgements

This work was supported by EU FP7 ICT-FET grant 284715 (ICON) and an IRC "New Foundations" grant. The drawing of a Llama is courtesy of `http://www.bluebison.net/`. We wish to thank all the people who contributed to LLAMA.

References

[1] Bernd Bischl, Michel Lang, Jakob Richter, Jakob Bossek, Leonard Judt, Tobias Kuehn, Erich Studerus, and Lars Kotthoff. *mlr: Machine Learning in R.*, 2014. R package version 2.2.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007.

[3] Michael J. Crawley. *The R Book*. Wiley Publishing, 1st edition, 2007.

[4] Thomas G. Dietterich. Ensemble methods in machine learning. In *International Workshop on Multiple Classifier Systems*, pages 1–15, 2000.

[5] Ian P. Gent, Christopher A. Jefferson, Lars Kotthoff, Ian Miguel, Neil Moore, Peter Nightingale, and Karen E. Petrie. Learning when to use lazy learning in constraint solving. In *19th European Conference on Artificial Intelligence*, pages 873–878, August 2010.

[6] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[7] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[8] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[9] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. In *CPAIOR*, May 2014.

[10] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *17th International Conference on Principles and Practice of Constraint Programming*, pages 454–469, 2011.

[11] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC instance-specific algorithm configuration. In *Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 751–756, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.

[12] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1137–1143. Morgan Kaufmann, 1995.

[13] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. Technical Report arXiv:1210.7959, University College Cork, 2012.

[14] Lars Kotthoff. Hybrid regression-classification models for algorithm selection. In *20th European Conference on Artificial Intelligence*, pages 480–485, August 2012.

[15] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 2014. Forthcoming.

[16] Lars Kotthoff, Ian P. Gent, and Ian Miguel. An evaluation of machine learning in algorithm selection for search problems. *AI Communications*, 25(3):257–270, 2012.

[17] Brett Lantz. *Machine Learning with R*. Packt, 2013.

[18] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1 edition, January 1993.

[19] Talal Rahwan and Tomasz P. Michalak. A game theoretic approach to measure contributions in algorithm portfolios. Technical Report RR-13-11, DCS, 2013.

[20] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

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

[22] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2011.

[23] David H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[24] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[25] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hierarchical hardness models for SAT. In *CP*, pages 696–711, 2007.

[26] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Twenty-Fourth Conference of the Association for the Advancement of Artificial Intelligence*, pages 210–216, 2010.

[27] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.

[28] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Hydra-MIP: automated algorithm configuration and selection for mixed integer programming. In *RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, pages 16–30, 2011.