Journal of Statistical Computation and Simulation
Vol. 00, No. 00, Month 201X, 1–16

RESEARCH ARTICLE

Runtime and Memory Consumption Analyses for Machine Learning R Programs [SPECIAL ISSUE: StatConf13] [PREPRINT]

Helena Kotthaus[a]*, Ingo Korb[a], Michel Lang[b], Bernd Bischl[b], Jörg Rahnenführer[b] and Peter Marwedel[a]

[a]*Department of Computer Science 12, TU Dortmund University, 44227 Dortmund, Germany*;
[b]*Department of Statistics, TU Dortmund University, 44227 Dortmund, Germany*

(*Received 00 Month 201X; final version received 00 Month 201X*)

R is a multi-paradigm language with a dynamic type system, different object systems and functional characteristics. These characteristics support the development of statistical algorithms at a high level of abstraction. Although R is commonly used in the statistics domain a big disadvantage are its runtime problems when handling computation-intensive algorithms. Especially in the domain of machine learning the execution of pure R programs is often unacceptably slow. Our long-term goal is to resolve these issues and in this contribution we used the traceR tool to analyse the bottlenecks arising in this domain. Here we measured the runtime and overall memory consumption on a well-defined set of classical machine learning applications and gained detailed insights into the performance issues of these programs.

Keywords: performance analyses; machine learning; classification algorithms; profiling

1. Introduction

The R language has become the "lingua franca" in the statistics community. Its main characteristic is a large set of dynamic features which allow the rapid development of new algorithms for statistics. However, this flexibility comes at a price: R is considered to be a rather slow language that needs a large amount of memory during runtime.

The field of machine learning is one area where the runtime problems are a major limit, especially when handling larger input data sets. This is a known problem within the R community and although some approaches for improving R have already been proposed (e.g. in [1]) the problem has not been solved yet. Even though there were alternative languages for statistical computation proposed like Lisp-Stat [2] or RIncanter [3] the R community has shown no interest in moving away from their language of choice even when faced with these issues. In recent years multiple projects were started with the goal to create alternative, more efficient R implementations, for example Renjin [4], Riposte [5] and NQR [6]. Other projects like pqR [7] or ORBIT [8] attempt to provide a faster R by modifying the original R. The original R implementation also recently gained the option to compile R functions into byte code for faster evaluation which provides some improvement especially for programs that use loops. All of these projects have usually shown improvements for simple R programs, but not all of them are yet able to speed up complex real-world applications like machine learning algorithms. Our goal is to provide

*Corresponding author. Email: helena.kotthaus@tu-dortmund.de

insights into the runtime behaviour of these algorithms on the original R interpreter. Thereby alternative R implementations can use the results to provide optimisations that improve the runtime of real-world code. Therefore we improved analysis tools originally developed for R 2.12 at Purdue University [9] and ported them to R version 3.0.2 for use in our analysis.

A major hurdle for general speedups of R programs is that R executes programs by interpretation as opposed to compiling them to machine. Classic ahead-of-time compilation of R code is hindered by the fact that R is highly dynamic. Thus information like data types which are needed for optimisations in the compiler is only available at runtime. For example, when a function is declared in an R program, no types are specified for its arguments. When such a function is called, there are multiple ways to pass the same set of arguments. These features make R very convenient for the programmer but very inconvenient for compiling it before running the program. Other languages with a similarly dynamic nature like Matlab and Python have overcome such speed issues by using just-in-time based compilation approaches (Majic [10] and PyPy [11]), which use knowledge gained at runtime to specifically compile entire program fragments for the time-intensive parts instead of either compiling the entire program at once or interpreting the program statement-by-statement. One popular runtime environment that provides a just-in-time compilation environment is the Java Virtual Machine. At useR! 2012 [12] we presented a concept for implementing an optimised version of R by targeting the Java Virtual Machine. One existing alternative R implementation that targets the JVM is the fastR project [13] which we intend to use as a basis to design optimisations specifically for computationally intensive machine-learning R applications.

As our goal is to reach the best possible optimisation for runtime of R programs it is necessary to closely examine the behaviour of real-world programs to determine their bottlenecks. Morandat et al. [14] analysed bottlenecks for R programs from different fields of statistics. We chose to focus specifically on machine learning algorithms combined with real-world data sets from the UCI repository [15] in order to ensure a realistic scenario when analysing the main reasons for the run-time problems of R. To support this analysis we used the above mentioned tracing tool called traceR [16].

In this paper we present the results of the runtime analyses as well as the memory consumption analyses of machine learning R programs and outline approaches to overcome the identified bottlenecks to support the development of alternative R interpreters as well as guiding changes in the original R interpreter. The rest of the paper is structured as follows: Section 2 gives a detailed overview of the characteristics of the machine learning benchmarks and the real-world input data sets that served as a basis for our analyses. In Section 3 we describe the profiling tools that we used to analyse these benchmarks. The results of the runtime analyses and memory analyses are presented in Section 4 and 5. Section 6 summarises this paper and gives an outlook on future work.

2. Machine learning benchmark setup

We selected a larger number of R implementations of some of the most popular machine learning algorithms to apply them on seven publicly available classification tasks provided by the UCI repository [15].

Our focus in particular is not on classification performance as this has been addressed in many works over the past years. Instead the attention is drawn towards resource requirements and bottlenecks.

The following list references all classification algorithms and their respective R versions we considered in our experiments. The choice is based on both the method's popularity and availability of an implementation. The reader is referred to the respective package

Table 1. Overview of UCI classification tasks after pre-processing. Dataset ID, number of observations and number of features stored as numeric, integer or factor.

Dataset	Observations	Numeric	Integer	Factor
Blood Transfusion Service Center	748	2	2	0
ILPD (Indian Liver Patient Dataset)	579	5	4	1
Pima Indians Diabetes	768	8	0	0
Credit Approval	653	6	0	9
German Credit	1000	7	0	13
Spambase	4601	57	2	0
MAGIC Gamma Telescope	19020	10	0	0

and its supplied citation information for further details on methods and implementations.

- AdaBoost, in package `ada` [17]
- Conditional inference trees, in package `party` [18]
- Gradient boosting machine, in package `gbm` [19]
- k-nearest neighbour classification, in package `kknn` [20]
- Support vector machine, in package `kernlab` [21]
- Linear discriminant analysis, in package `MASS` [22]
- Logistic regression, in package `stats` [22]. Binary classification decision derived using a probability cutpoint of 0.5.
- Least-squares support vector machine, in package `kernlab` [21]
- Naive Bayes, in package `e1071` [23]
- Multi-nominal regression, in package `nnet` [22]
- Random forest, in package `randomForest` [24] using majority voting of classification trees
- Regularized discriminant analysis, in package `klaR` [25]
- Classification tree (CART), in package `rpart` [26]

Most of the listed implementations allow the user to adjust several parameters to increase predictive performance. For our purpose we did not tune any parameters but instead either used the mostly meaningful defaults or, if available, used the implementations internal auto-tuning process. Therefore a fair comparison of the classification performance is not possible as this would require some sort of parameter tuning. Optimal tuning is no way straightforward – it requires deep experience with or knowledge of the algorithms or, alternatively, a good automatic tuning approach.

However, even if we focus on runtimes, predictive performance cannot be completely ignored. It is a sufficient indicator for a normal program flow, e.g. that no bugs resulting in fast but meaningless predictions were triggered. Therefore classification performance, measured by the mean misclassification rate of a 10-fold cross validation, is mainly provided for reference purposes in our analysis.

We used the package `mlr` [27] to conveniently apply all learners on identical cross-validation splits and predict the performance. The unified interface provided by `mlr` causes some overhead which influences both runtime and memory consumption. For example, converting the input data from a matrix to a data frame or vice versa is a frequent pre-processing operation handled internally by `mlr`. Nevertheless, we believe that this does not severely affect the interpretation of our results as we believe that we even come closer to a real-life computer experiment seen as an entire process including such typical transformations normally done by the user.

On the data side we chose the datasets listed in Table 1. The most important criteria for including datasets are (a) being a 2-class classification problem, (b) sufficiently large number of observations to archive accurate results in the tracing and (c) having an even and realistic mixture of data types. We preliminary removed all observations with missing values from the data.

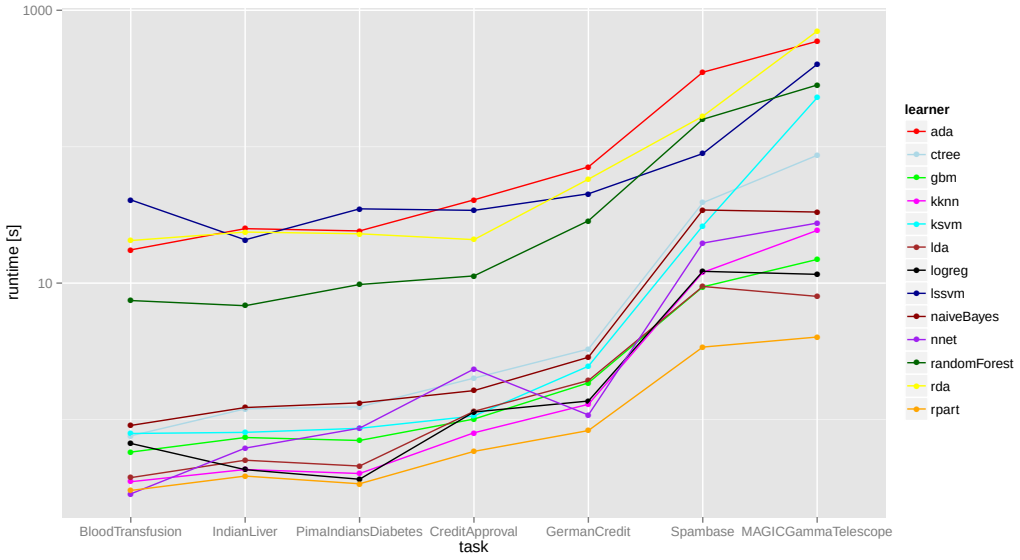

Figure 1. Runtimes for 10-fold cross validation: model fit, prediction and calculation of the misclassification error on the respective datasets. Y axis is on $\log_{10}$ scale.

Table 2. Mean misclassification rates across 10-fold cross validation.

	ada	ctree	gbm	kknn	ksvm	lda	logreg
IndianLiver	0.30	0.28	0.28	0.31	0.29	0.29	0.27
PimaIndiansDiabetes	0.24	0.25	0.35	0.27	0.24	0.23	0.23
GermanCredit	0.23	0.27	0.30	0.28	0.25	0.25	0.25
MAGICGammaTelescope	0.14	0.15	0.35	0.16	0.13	0.22	0.21
Spambase	0.05	0.09	0.39	0.08	0.07	0.11	0.07
BloodTransfusion	0.22	0.22	0.24	0.24	0.21	0.23	0.23
CreditApproval	0.13	0.14	0.45	0.16	0.14	0.13	0.15

	lssvm	naiveBayes	nnet	randomForest	rda	rpart
IndianLiver	0.29	0.45	0.29	0.29	0.31	0.34
PimaIndiansDiabetes	0.23	0.25	0.33	0.23	0.24	0.25
GermanCredit	0.26	0.25	0.30	0.23	0.29	0.27
MAGICGammaTelescope	0.20	0.27	0.24	0.12	0.21	0.18
Spambase	0.25	0.29	0.06	0.05	0.33	0.11
BloodTransfusion	0.26	0.25	0.24	0.25	0.24	0.21
CreditApproval	0.14	0.23	0.19	0.13	0.37	0.15

Figure 1 gives an overview of the runtimes for the datasets and Table 2 documents the mean misclassification rates. Regarding the misclassification rates we see no evidence to believe that any bugs resulting in constant or random predictions were triggered.

In the following analysis we focus on the tasks that use the Magic Gamma Telescope dataset as it provides the highest aggregate runtime over all the algorithms. We expect that this choice provides the clearest view on any bottlenecks and minimizes the influence of one-time start–up costs.

3. Profiling tools

The R interpreter already provides profiling tools like Rprof for profiling the runtime of R programs and also some options for profiling memory usage like tracemem and Rprofmem. Rprof is a sampling profiler, which means that it halts the program execution at regular intervals to check which function is currently executing. A sampling profiler has a relatively small measurement overhead but the results can vary wildly between two

runs, especially when many functions with very small runtimes are involved. A bottleneck could consist of functions that individually have a short runtimes, but are called very often. In this case a sampling profiler may miscalculate the runtime percentage spent in these functions because it is unable to measure the program flow that happens between two of its samples. Furthermore, it operates only at the R function level, but does not provide details about the internals of the R interpreter. Similarly, the memory profiling options available are not detailed enough for our purposes – for example, Rprofmem only reports memory allocations related to certain types of user data, but not those related to interpreter internals like pairlists used for passing arguments in a function call.

To analyse the bottlenecks of R it is indispensable to have a more detailed view into the interpreter's internals. For this reason we have chosen the traceR tool as the basis for our analysis. TraceR was originally developed at Purdue University for R 2.12 [9]. Besides porting it to the R version 3.0.2 we also improved its analysis capabilities to provide more detailed insights, e.g. to gather more information about the different sizes of vectors used during the execution of an R program. Our profiling tool is available on GitHub [16]. The main parts of the traceR tool are two instrumented versions of the R interpreter, one to measure the time profile of an R program and one to analyse non-time-related behaviour like memory allocations and details about function parameters. This separation removes the overhead of the memory measurements from the time measurements.

TraceR uses a deterministic profiling approach where each interesting location is instrumented with an explicit call to the profiler for time measurements. This way no calls can be missed, but a larger overhead is incurred compared to the sampling approach. Since we directly measure within the interpreter, we can generate more detailed data than Rprof. For example, we can measure how much of the program is spent in C/Fortran code supplied by R packages or how much time was needed for memory management tasks like garbage collection.

The results of our runtime and memory consumption analysis which were produced by traceR are presented in the following sections.

4. Results: runtime analysis

In our runtime analysis we present an overall runtime profile for each machine learning benchmark to expose their runtime problems and suggest optimisation ideas.

For the following measurements we ran our analysis R interpreters on a computer equipped with two AMD Opteron 2378 processors (quad-core, 2.4 GHz) and 16 GBytes of RAM, using Debian 7.3 as the operating system. Our profiling system traceR is based on R 3.0.2 which was compiled using the default compiler flags (just `-O2`) with GCC version 4.7.2. The default settings were used to compile and run R, so only installed packages were byte-code compiled and the default BLAS was used. The byte-code compiler that is included with the R distribution provides the option to compile R code into a byte-code representation that can be executed faster than standard R functions. It also provides some optimisations which generally are beneficial for explicit loops in R code.

As can be seen from Section 2, the runtime of the benchmarks is generally longest for the MAGIC Gamma Telescope data set. Although we have run our analysis on most of the data sets, we will focus on the MAGIC Gamma Telescope set in the following sections, to get accurate results in our bottleneck analysis.

To give a better overview of the behaviour of an R program, we have summarized our measurements into eleven categories which can be split into three groups. The first group is external code like C or Fortran. The R interpreter provides multiple ways for interfacing with external code to allow both the use of generic external libraries as well as libraries especially written for use within R. The difference between those two is

the way parameters are passed – for the generic interface, the R interpreter handles all required type conversions itself. Libraries written especially for R receive the internal representation of values instead and handle any required conversions themselves. External libraries represent the lowest optimisation potential as they are executed outside of the R interpreter. The second group are functions directly provided by the R interpreter like arithmetic operations or built-ins. Their optimisation potential varies by the specific function. The third group with the highest optimisation potential are the internal tasks of the R interpreter that are not directly visible to an R programmer, but still important for the execution of an R program like memory management tasks.

Figure 2 shows the percentage of runtime spent in the various categories by the benchmarks. We consider a higher proportion of time spent in a category to be a valid hint for the optimisation potential of this category. However, as explained above the optimisation potential varies strongly depending on the group the category belongs to. Additionally optimisations in one category may influence the time needed in other categories – for example, reducing the number of memory allocations would reduce the time needed for these allocations (*MemAlloc*) and also influence the time spent in garbage collection (*GC*) since it needs to check fewer objects in memory. Since our goal is to develop optimisations that are beneficial to all machine-learning algorithms and not just a single one, we need also to focus on the total time spent in a specific category over all benchmarks to approximate the potential for optimisation. The runtime of the individual benchmarks varies greatly, so we use relative proportions to ensure comparability. For further details see Table A2 in the appendix.

The benchmarks are sorted by the category *External* which is the time spent on non-R-code like C or Fortran code. This proportion varies strongly between benchmarks as some of them are mostly implemented in R while others have been augmented with external code. The highest amount of time spent on external code appears in the `nnet` and `randomForest` benchmarks, which both spend 79.0% of their runtime outside of the R code. This demonstrates that the authors of these packages considered the runtime issues of the R interpreter to be serious enough that they implemented large parts of their algorithms in a faster language like C, even though this generally requires more effort than directly implementing an algorithm in R. On the other end of the scale the `rda` and `lssvm` benchmarks both spend less than 0.1% of their runtime in external code and are also two of the slowest programs within the benchmark set. This is not directly correlated with the use of R versus external code though as can be seen by `ada` and `randomForest`: Although they are among the slowest in the set, both spend more than 45.0% of their runtime in external code. Thus even the use of external code does not guarantee fast runtimes as the complexity of the algorithm itself will largely influence its runtime and memory footprint. When considering the sum of time spent in external code by all benchmarks, only 15% of the total runtimes is spent in external code. Thus optimisations on the R code side can still result in large improvements of runtime.

As already mentioned, the categories with the highest optimisation potential are the interpreter internal tasks, especially for those benchmarks that spend less time in external code like `rda`, `lssvm` and `naiveBayes`. Here the amount of time spent in *Lookup* which looks up variables and functions can be up to 16.1% of the total runtime. The reason for this is that before a function can be executed or a variable can be accessed, the interpreter has to look up its definition or value through a chain of environments. Such an environment provides a mapping from a symbolic name to a variable or function. Initially there is only the global environment, also known as the user workspace. Each function call adds one more environment to the end of this chain. This search has to be repeated every time the R program uses the name of a variable or function. This could be avoided by caching the result of the lookup. Again, the R language creates additional problems for such an approach: R is a very dynamic language that allows a program

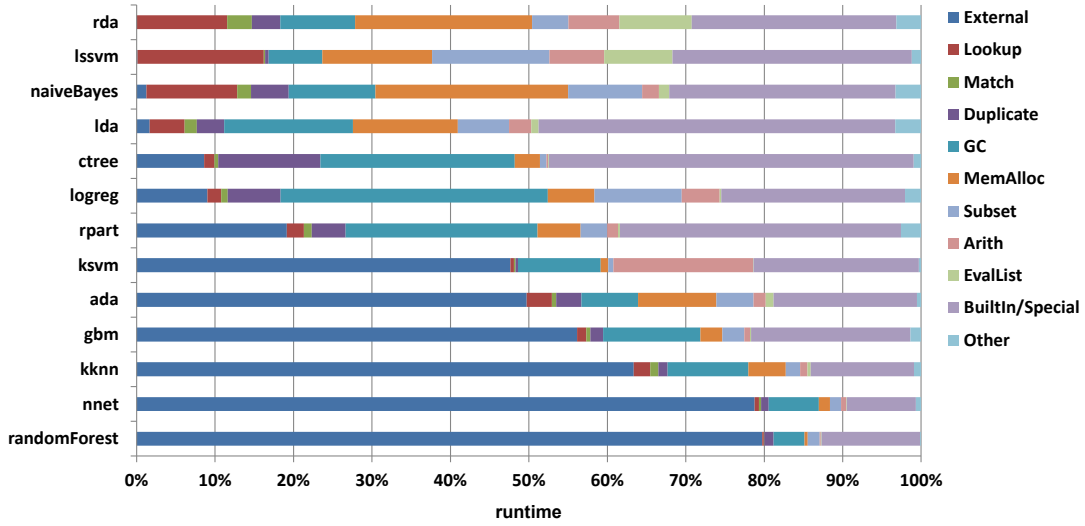

Figure 2. Runtime profiles relative to the total runtime of each benchmark.

to change the definition of a function during runtime. This complicates a lookup cache as it would need to ensure that it never returns a cached lookup that has already been redefined by the R program.

After the interpreter has located a function definition that should be called, it needs to *Match* the arguments given in the call to the arguments given in the definition. Function arguments can be passed by name, by position or via the `...` argument, which is used to pass a variable number of arguments. The time spent on argument matching is up to 3.1% of the total runtime as seen in the `rda` benchmark, and over all benchmarks only 1.9% of the total runtime is needed for argument matching. Our results show that there were no function calls with more than 17 parameters and 84.8% of all calls over all benchmarks had no or just one parameter. This shows that the machine learning benchmarks rarely use the full flexibility of argument passing that R provides, which has a positive effect on runtime and is thus not a bottleneck compared to other R programs.

R uses a copy-on-write scheme for function arguments, the value of an argument is generally only duplicated when it is modified by the called function, although exceptions exist. Duplication is marked as *Duplicate* in Figure 2. Its proportion varies between 0.3% (`ksvm`) and 13.0% (`ctree`) of the total runtime. Although duplication itself contributes only 3.0% of the total runtime of all benchmarks, it causes an increase in the proportion of memory allocation (*MemAlloc*) and garbage collection (*GC*) time because the duplicated values need to be stored and removed later. Besides the function lookup, memory allocation and garbage collection are the two main contributors to the runtime of the benchmarks from the group of interpreter-internal tasks.

The garbage collection (*GC*) scans the data that was allocated by the R program for values that are no longer in use and removes them. The runtime spent on garbage collection varies between 3.9% for `randomForest` and 34.0% for `logreg`. Over all benchmarks, 9.0% of the total runtime is spent in *GC*. This value is influenced by the number of memory allocations and also by the memory footprint that is needed for the data structures of the R program. The influence of the memory footprint can be seen in the `ksvm` benchmark, where 10.6% of the runtime is spent in garbage collection, even though only 0.9% is spent in memory allocation. Here memory allocation percentage is low because the benchmark uses a small number of data structures (in this case vectors), but each of those has a large size and thus a large memory footprint. For other benchmarks the memory allocation can be a more important part of the runtime with a maximum of

24.6% in `naiveBayes`. This benchmark allocates a large amount of vectors with a small memory footprint. The dependence between memory allocation, memory footprint and garbage collection will be explained in more detail in Section 5 as this is one of the most important bottlenecks within the internal interpreter tasks.

The last group in our runtime bottleneck analysis are the functions provided by the R interpreter. We have picked two sets from the group of interpreter-provided functions as they are a significant contributor to the overall runtime of a benchmark. The first is the set of subsetting operations (*Subset*) used for the evaluation of vector index expressions. Those operations are very important for machine learning programs to generate training data subsets. The highest proportion of runtime for subsetting operations occurs in the `lssvm` benchmark with a value of 14.9%. Since subsetting has to allocate new space to return the results also it directly influences the time spent on memory allocation and thus garbage collection.

The second set from the group of interpreter-provided functions are the basic arithmetical operations (*Arith*) like addition or matrix multiplication. Since R usually operates on vectors, these operations could benefit from the use of vector-oriented instructions in the CPU, which R currently utilises only if special libraries are used. However, over all benchmarks just 5.5% of the total runtime is needed for arithmetic operations. This implies that optimising the runtime of these functions is unlikely to have a large impact on the overall runtime.

Before a real arithmetic operation happens the R interpreter has to run several pre-processing steps, like data type checks or data type conversion to ensure that the data has a valid format for operation. This supports the dynamic type system of R. For example, different calculations need to be performed for the multiplication of integer values compared to complex values. Those pre-processing steps are not only needed for arithmetic operations but basically for all functions. The overhead of these pre-processing steps could be reduced by the use of function specialisation, which is a common compiler optimisation. This optimisation takes a generic function that can accept any data type and converts it into specialised version that accept only specific data types, which avoids the overhead needed for type checking. Such a specialisation has been implemented on the byte-code level in the ORBIT VM [8] next to other optimisations, yielding a total speedup of 3.5x over the standard byte code interpreter on a set of R benchmarks that were mainly looping over data.

Although we cannot demonstrate function specialisation directly in the R interpreter, we can illustrate the principle with the example program in Figure 3. Here, a simple function using S3 dispatch is defined and called both via the standard UseMethod mechanism as well as directly. The latter case would correspond to a specialized call where the interpreter has determined during runtime that this particular call only receives arguments of class "special". In this simple example, the direct call that avoids UseMethod is nearly two times faster than the generic function call because the overhead of the type check can be avoided.

For a subset of functions provided by the R interpreter that are called built-in functions, another required pre-processing step is shown as *EvalList* in Figure 1. In this step all arguments of the built-in function are evaluated before they are passed, which needs 6.3% of the total runtime of all benchmarks. R usually delays this evaluation when other types of functions like user functions or special functions are called.

The last important category *Builtin/Special* contains the time spent on special and built-in functions except for arithmetic and subset operations which we have already examined separately. Special functions include control-flow structures like `if` or `return`. When considering the sum of time spent in this category over all benchmarks, 24.8% of the total runtime is spent in these functions, including the time needed for type checks and data conversion that could be optimised by the previously mentioned function

```
1  # define a simple S3 function
2  addOne         <- function (x) UseMethod("addOne",x)
3  addOne.special <- function (x) x + 1
4
5  # define a variable of class "special"
6  specialvar         <- 0
7  class(specialvar) <- "special"
8
9  # dispatch via UseMethod
10 addOne(specialvar)
11
12 # directly call the specialized version of addOne
13 addOne.special(specialvar)
```

Figure 3. Illustration of function specialisation.

specialisation.

The final category called *Other* includes the remainder of the runtime like the start–up time of the interpreter and does not represent a viable optimisation target.

5. Results: memory consumption analysis

For our memory analysis we concentrate on analysing the most important data structures needed while executing an R program. This analysis is important to develop different optimisations which on the one hand may reduce the footprint of the data structures used by the interpreter and on the other hand reduce the amount of allocated data structures. Additionally the memory usage of an R program and its run time influence each other. For example, a program that uses a lot of function calls needs memory to hold the parameter lists which in turn requires more time for memory management since these lists need to be allocated and later removed.

The first Section 5.1 gives an overview of the memory allocation behaviour for the examined benchmarks while the second Section 5.2 focuses specifically on vectors and their memory footprint. Similar to the runtime analysis in Section 4 we focus on the MAGIC Gamma Telescope data set, using the same execution environment.

5.1 *Memory consumption overview*

In this section we concentrate on the entire amount of memory allocated during the runtime of each benchmark. Therefore we focus on allocations of new data in memory, but ignore their later removal by the garbage collector to provide a clearer view on the influence of the memory allocation on the overall runtime. The garbage collector periodically scans all objects allocated within the interpreter to determine if they are still in use and deallocates those for which it can determine that they are not in use anymore. This makes the memory available again for reuse by objects that are allocated later. Later in this section we also consider the actual peak memory consumption compared to the total memory allocated to provide insights into the gains provided by the garbage collector.

Figure 4 shows the distribution of the memory allocations between different categories that can be split into two groups: Data structures that are primarily used for interpreter-internal tasks and structures that primarily hold user data. Since the memory allocations of the benchmarks vary, we used relative proportions to ensure better comparability. For further details see Table A1 in the appendix.

The first category are *Pairlists* that are mostly used for internal data of the interpreter

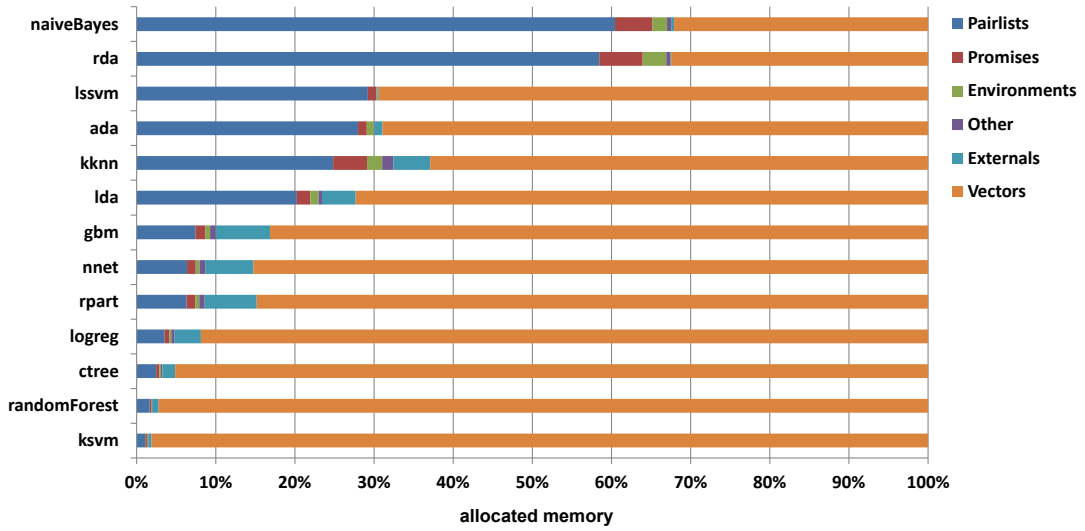

Figure 4. Memory profiles relative to the total memory allocation of each benchmark.

itself. Although pairlists have many uses within the R interpreter, one major contributor to their allocation is the creation of argument lists that are needed for each function call. In the machine learning benchmarks we analysed that the length of the argument list is mostly irrelevant: As we have mentioned in Section 4 most function calls have zero or one arguments, thus for our benchmarks the number of calls that triggers the pairlist creation is a much more important factor than the number of arguments per call. Programs with a high amount of R function calls suffer from a higher memory overhead which negatively influences their runtime.

Memory allocated for pairlists ranges from 1.1% for `ksvm` to 58.5%-60.4% for `rda` and `naiveBayes`. The high amount of allocated pairlists in `rda` corresponds to its low use of external code which is shown in Figure 2. The large proportion of time spent on lookups and argument matches also confirms that `rda` use many calls to R functions. It is also the slowest benchmark in our set. The example of `ctree` however shows that a high runtime percentage in R code does not necessarily correspond to a large allocation of pairlists or a slow runtime. `ctree` only spends 8.6% of its runtime in external code, but merely uses 2.5% of its allocated memory for pairlists. Its low runtime percentage for lookups and matching confirms that it uses relatively few function calls compared to `rda`, so fewer pairlists are needed for the arguments.

The next category in the group of the interpreter-internal data structures are *Promises*. Due to the functional characteristics of the R language, a function argument can not only be bound to a simple value, but also to a complex expression like other function calls. Such an expression is only evaluated when its result is really needed. This can happen directly in the called function or later when the called function passes a not-yet-evaluated argument to another function where it is needed. The mechanism for this is called lazy evaluation and R implements it by boxing each argument in a so-called "promise". A promise contains a reference to the original argument and the corresponding environment. Since those two references do not require much memory, a promise needs just 56 bytes (on a 64 bit system) and summed over all benchmarks only 2.5% of the total allocated memory is used for promises. Even though this is just a low percentage compared to allocations of other data structures, the absolute values reveal some optimisation potential. Three of the benchmarks (`lssvm`, `naiveBayes`, and `rda`) use more than 1 gigabyte of memory allocations just for promises with a maximum of almost 6 gigabytes for `rda`. Our analysis tools show that in an average of 86.4% of all cases the creation of a promise was

not necessary because its evaluation happened in the same function that it was initially created for. As explained in Morandat et al. [14] it is not always possible to replace the creation of promises by eager evaluation.

The last important category in the group of interpreter-internal data structures are *Environments*. Similar to promises they only contain references to other values and are mainly created when an R function is called. Considering all benchmarks, 1.2% of their total memory allocations is used for environments. Since the interpreter needs to search through the chain of environments during lookups, this apparently small value has a strong influence on the time required for lookups. As described in Section 4 lookups are the second-largest bottleneck in the interpreter internal tasks.

There are also a few additional interpreter-internal data structures that require memory allocation, but since they use less than one percent of the total allocated memory of all benchmarks, the *Other* category is not an interesting target for optimisation. Taken together, all of the described interpreter-internal data structures (ignoring the vector headers) sum up to 40% of the total allocated memory. Thus almost half of the allocated memory is used for executing the R program and not for the user data it processes, which adds more overhead for memory allocation and garbage collection.

The user data allocations are divided in allocations that are triggered from external code (*Externals*) and the allocation of *Vectors*. Memory allocation from external libraries requires just 0.5% of the total allocated memory while vectors account for 61.2% of the total allocated memory over all benchmarks and thus they are the biggest consumer of memory allocations in the R interpreter. Therefore we will take a more detailed look at vector allocations in the next section.

5.2 *Memory consumption for vectors*

In this section we focus on vectors as they are the most important data structure in the R language and higher-dimensional structures like matrices and arrays are internally constructed from it. When R allocates memory for a vector it differentiates between small and large vectors. Small vectors can store up to 16 double or 32 integer or logical values, large vectors are used when the number of elements exceeds this limit. In addition to these two classes we have separated the class of vectors with exactly one element from the small vectors as well as the class of vectors with zero elements, which for example can be created when all elements are removed from a vector. Figure 5 shows the proportions of these allocations in relation to the total vector memory allocation of each benchmark. For most benchmarks the *Large Vectors* dominate. Even though the Figure 5 shows only the ratios for the processing of the Magic Gamma Telescope data set, the distribution between *Large* and *Small/Single-Element Vectors* is roughly the same for all input data sets. `NaiveBayes` is the only benchmark where a non-negligible amount of *Zero-Element Vectors* appears, reaching about 4.0% of the total vector memory allocated compared to less than 0.2% for all other benchmarks.

The benchmarks `rda` and `naiveBayes` use a much higher percentage of single-element and small vectors compared to the other benchmarks. This is another factor besides the large number of pairlists that influence the time spent on *MemAlloc* (Figure 2) for these two benchmarks. Since the number of allocations increases when the same amount of data is processed using vectors of a smaller size, the time for allocation increases. Over all benchmarks, 20.4% of all allocated vector memory is used for single-element or small vectors, thus for most of the benchmarks the large vectors are the main factor that influences the vector memory allocation. On the other hand the single-element and small vectors have the best potential for optimisations: Each vector needs a header block so it can be integrated in the memory management subsystem of R. The size of this header

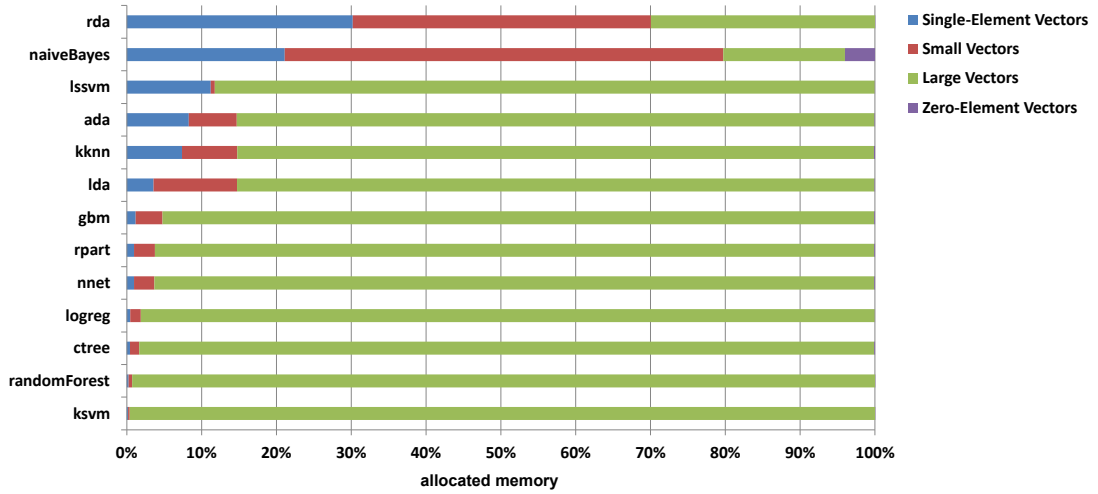

Figure 5. Vector memory profile relative to the total memory allocation of vectors for each benchmark.

is 40 bytes (on a 64 bit system) for each vector. For vectors with fewer elements, the relative overhead of this header increases. The worst case are single-element vectors: A 40 byte header is needed to manage an object whose size is just 4 or 8 bytes, thus in the worst case the header needs 10 times more memory compared to the real data. This suggests a big optimisation potential for introducing scalar values that are not boxed within a vector and thus could use a smaller header. If such a scalar value is only used within a function, a just-in-time compiler may even be able to keep the value in a CPU register instead of storing it in main memory, saving the time for both allocation and garbage collection for scalar values.

To show the optimisation potential for scalars the number of vectors of each class is important. The higher the number of single-element vectors used by a benchmark the more memory can be saved by the use of scalar values. Table 3 shows the proportion of vectors for the single-element, small and large classes with the Average line showing the proportion over the total number of vectors used by all benchmarks. The benchmark `lssvm` uses the highest percentage of single-element vectors with 95.52% while over all benchmarks 56.96% of all vectors allocated are single-element vectors. For `lssvm` just the headers for these vectors require about 6.1 gigabytes of allocated memory which is 6.5% of the total memory allocation that could be reduced by introducing scalar values. Even though over all benchmarks only up to 5.4% of the total allocated memory could be saved by using scalar values instead of single-element vectors, this small percentage has a high influence on the runtime since it influences the time spent on memory allocation and garbage collection which are two of the most important bottlenecks in the runtime analysis (Section 4).

In addition to the total memory allocation we have also measured the peak memory usage of the R interpreter as reported by the operating system using the `getrusage` system function. This function reports the maximum amount of memory that the program has requested from the operating system. The value is lower than the total allocated memory value because the garbage collection removes unused values from memory during the runtime. Figure 6 shows the maximum memory usage compared to the total memory allocation for each benchmark. Over all benchmarks, 55.7 times more memory was allocated compared to the maximum amount used at once. This ratio explains the amount of time spent on garbage collection. While the total memory allocated is influenced by the number of data structures that were allocated during runtime, the memory used is influenced by the footprint of those data structures.

Table 3. Relative number of vectors for single-element, small and large vectors.

benchmark	single-element [%]	small [%]	large [%]
ada	58.36	40.83	0.81
ctree	34.69	60.68	4.63
gbm	30.39	67.13	2.48
kknn	52.61	46.65	0.74
ksvm	47.75	50.34	1.92
lda	29.67	69.78	0.55
logreg	28.93	67.00	4.06
lssvm	95.52	4.00	0.48
naiveBayes	28.58	70.59	0.83
nnet	31.70	65.21	3.09
randomForest	38.94	59.69	1.37
rda	50.20	49.72	0.08
rpart	31.24	65.88	2.88
Average	56.96	42.65	0.40

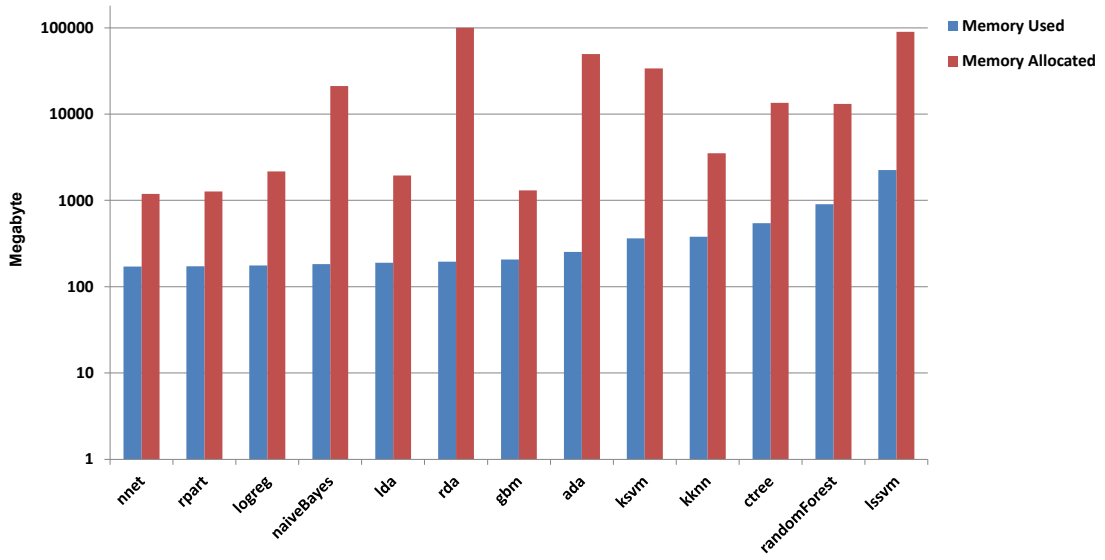

Figure 6. Maximum memory usage versus total memory allocation for each benchmark. Y axis is on log scale.

Benchmarks that use mostly data structures with a big footprint like large vectors also have a low ratio between allocated and used memory because they need to allocate less data structures for the same input data set. The benchmarks `gbm`, `nnet` and `rpart` for example have a low ratio between allocated and used memory because large vectors form the largest part of their allocated memory and thus the overall number of allocated data structures is low. The benchmark with the highest ratio between allocated and used memory is `rda` with a factor of 512. This value also influences the 32.1% of runtime that it needs for memory allocation and garbage collection (see Figure 2). As can be seen from Figure 4 `rda` uses 58.5% of its allocated memory for pairlists. Pairlists have a small footprint, `rda` uses a large number of allocations for them and thus spends a significant portion of its runtime in memory allocation. This demonstrates that for a representative analysis of runtime and memory bottlenecks not only the overall allocated memory of a program has to be examined, but also the memory footprint and amount of each data structure has to be taken into account.

6. Conclusion

This paper is the first one to present detailed insights into the runtime and memory behaviour of the most popular R implementations of machine learning algorithms when running on the R interpreter. For our analysis we applied the machine learning algorithms to real-world data sets from the UCI [15] repository. To uncover the bottlenecks of those R program benchmarks we have improved different profiling tools originally developed for R 2.12 at Purdue University [9] and ported them to the version 3.0.2 of R.

On the runtime bottleneck side, we were able to demonstrate that memory management is a major contributor to the total runtime of the benchmarks. Since the time spent on memory management is influenced both by the number of data structures that are allocated as well as their footprint, we propose to introduce a new data type for scalar values that could be used to replace the frequent usage of single-element vectors. The introduction of scalars appears to be a promising approach to bypass the inherent overhead of single-element vectors and thus the total memory footprint of an R program.

For some of the benchmarks whose algorithms are implemented mostly as R code the overhead incurred by function calls is also a significant contributor to their runtime. A dynamic compilation approach may be able to provide improvements for this bottleneck as it could reduce the number of function calls by inlining small functions into their caller. Another area where such an approach can be helpful is the time spent on functions provided by the R interpreter: to support the dynamic type system of the R language, these functions must perform type checking and conversion of their arguments. Using function specialisation, this overhead could be avoided whenever the data types used in the call are known to the compiler.

On the memory bottleneck side, we have shown that vectors and pairlists are the two main contributors to the total allocated memory. As for vectors our results show that a large number of them are created to store just a single element, which incurs an unacceptably high memory overhead in the R interpreter. Here our proposed scalar optimisation would reduce the amount of allocated memory as well as the number of data structures created during the runtime of an R program, which are both factors that influence the time needed for memory management.

Our results support both the development of alternative R interpreters that could attempt to integrate our optimisations ideas to avoid the bottlenecks we have shown as well as guiding changes within the original R interpreter, even though this may require major changes to its code base. One existing alternative R implementation that targets the JVM is the fastR [13] project, which represents one possible basis for our future work to design optimisations specifically for computationally intensive machine-learning R applications.

Acknowledgement

This work was partly supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876, project A3. The authors thank Uwe Ligges, Jan Vitek, Floréal Morandat and Luke Tierney for providing us with feedback.

References

[1] Tierney L. Compiling R: A Preliminary Report. In: Proceedings of the 2nd international workshop on distributed statistical computing. 2001.
[2] Tierney L. Lisp-Stat. 2014. Available from: http://homepage.cs.uiowa.edu/ luke/xls/xlsinfo.

[3] RIncanter - Use embedded R from Clojure and Incanter. 2014. Available from: https://github.com/jolby/rincanter.
[4] Bertram A. Renjin: JVM-based Interpreter for the R Language for Statistical Computing. 2014. Available from: http://www.renjin.org.
[5] Talbot J, DeVito Z, Hanrahan P. Riposte: a trace-driven compiler and parallel VM for vector code in R. In: Proceedings of the 21st international conference on parallel architectures and compilation techniques. PACT '12. Minneapolis, Minnesota, USA. New York, NY, USA: ACM. 2012. p. 43–52.
[6] Kane MJ, Emerson JW. Not Quite R for the Parrot VM. 2014. Available from: https://github.com/NQRCore.
[7] Neal R. pqR - a pretty quick version of R. 2014. Available from: https://github.com/radfordneal/pqR.
[8] Wang H, Wu P, Padua D. Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization. In: Proceedings of the international symposium on code generation and optimization (cgo). CGO '14. Orlando, Florida. 2014.
[9] The Reactor Project. 2014. Available from: http://r.cs.purdue.edu.
[10] Almasi G, Padua D. MaJIC: A Matlab Just-In-Time Compiler. In: Languages and compilers for parallel computing. Springer Berlin Heidelberg. 2001. p. 68–81.
[11] Bolz CF, Cuni A, Fijalkowski M, Rigo A. Tracing the meta-level: PyPy's tracing JIT compiler. In: Proceedings of the 4th workshop on the implementation, compilation, optimization of object-oriented languages and programming systems. ICOOOLPS '09. Genova, Italy. ACM. 2009. p. 18–25.
[12] Kotthaus H, Plazar S, Marwedel P. A JVM-based Compiler Strategy for the R Language. Research Poster at The 8th International R User Conference. 2012.
[13] FastR. 2014. Available from: https://github.com/allr/fastr.
[14] Morandat F, Hill B, Osvald L, Vitek J. Evaluating the design of the R language: objects and functions for data analysis. In: Proceedings of the 26th european conference on object-oriented programming. Beijing, China. Springer-Verlag. 2012. p. 104–131.
[15] Bache K, Lichman M. UCI machine learning repository. 2014. Available from: http://archive.ics.uci.edu/ml.
[16] traceR. 2014. Available from: https://github.com/allr/tracer.
[17] Culp M, Johnson K, Michailidis G. ada: an R Package for stochastic Boosting. 2012. R package version 2.0-3. Available from: http://CRAN.R-project.org/package=ada.
[18] Hothorn T, Hornik K, Zeileis A. Unbiased Recursive Partitioning: A Conditional Inference Framework. Journal of Computational and Graphical Statistics. 2006;15(3):651–674. R package version "1.0-13".
[19] Ridway G, et al.. gbm: Generalized Boosted Regression Models. 2013. R package version 2.1. Available from: http://CRAN.R-project.org/package=gbm.
[20] Hechenbichler KSK. kknn: Weighted k-Nearest Neighbors. 2013. R package version 1.2-5. Available from: http://CRAN.R-project.org/package=kknn.
[21] Karatzoglou A, Smola A, Hornik K, Zeileis A. kernlab - An S4 Package for Kernel Methods in R. Journal of Statistical Software. 2004;11(9):1–20. R package version 0.9-19.
[22] Venables WN, Ripley BD. Modern Applied Statistics with S. 4th ed. New York: Springer. 2002. R package version 7.3-29. Available from: http://www.stats.ox.ac.uk/pub/MASS4.
[23] Meyer D, Dimitriadou E, Hornik K, Weingessel A, Leisch F. e1071: Misc Functions of the Department of Statistics (e1071), TU Wien. 2012. R package version 1.6-2. Available from: http://CRAN.R-project.org/package=e1071.
[24] Liaw A, Wiener M. Classification and Regression by randomForest. R News. 2002;2(3):18–22. R package version 4.6-7. Available from: http://CRAN.R-project.org/doc/Rnews/.
[25] Weihs C, Ligges U, Luebke K, Raabe N. klaR: Analyzing German Business Cycles. In: Baier D, Decker R, Schmidt-Thieme L, editors. Data analysis and decision support. Springer-Verlag. 2005. p. 335–343. R package version 0.6-9.
[26] Therneau T, Atkinson B, Ripley B. rpart: Recursive Partitioning. 2013. R package version 4.1-3. Available from: http://CRAN.R-project.org/package=rpart.
[27] Bischl B. mlr: Machine Learning in R. 2013. R package version 1.0-2612. Available from: http://CRAN.R-project.org/package=mlr.

Appendix A. Runtime and Memory Profiles of all Benchmarks

Table A1. Distribution of the memory allocated between six different categories of each benchmark. The categories can be split into two groups: *User data*, which are data structures that primarily hold user data and *Internal data* which are structures that are primarily used for interpreter-internal tasks.

	User data		Internal data			
Benchmark	External	Vectors	Pairlists	Promises	Environments	Other
ada	1.04	68.89	27.97	1.08	0.88	0.06
ctree	1.62	95.12	2.48	0.41	0.20	0.17
gbm	6.85	83.08	7.41	1.27	0.57	0.74
kknn	4.64	62.90	24.81	4.28	1.92	1.41
ksvm	0.36	98.12	1.11	0.20	0.11	0.09
lda	4.24	72.30	20.23	1.72	1.01	0.45
logreg	3.34	91.85	3.52	0.58	0.27	0.40
lssvm	0.14	69.43	29.22	1.08	0.09	0.05
naiveBayes	0.37	32.09	60.46	4.71	1.85	0.52
nnet	6.10	85.16	6.35	1.10	0.50	0.71
randomForest	0.80	97.24	1.64	0.15	0.10	0.07
rda	0.09	32.46	58.47	5.48	2.99	0.51
rpart	6.55	84.78	6.31	1.08	0.50	0.70

Table A2. Distribution of the runtime spent in eleven different categories of each benchmark. The categories can be split into three groups: *External* which is the time spent in external code like C or Fortran libraries, *R-Provided* which are functions directly provided by the R interpreter, *R-Internal* which are the internal tasks of the R interpreter that are not directly visible to an R programmer.

		R-Provided			R-Internal						
Benchmark	External	Subset	Arith	Builtin/Special	Lookup	Match	Duplicate	GC	MemAlloc	EvalList	Other
ada	49.73	4.73	1.57	18.28	3.20	0.57	3.19	7.22	9.99	0.99	0.54
ctree	8.62	0.81	0.23	46.49	1.29	0.51	13.00	24.79	3.22	0.10	0.95
gbm	56.14	2.80	0.74	20.33	1.17	0.54	1.62	12.38	2.80	0.12	1.35
kknn	63.35	1.83	0.95	13.18	2.13	1.03	1.17	10.28	4.78	0.38	0.91
ksvm	47.63	0.68	17.79	21.07	0.47	0.20	0.30	10.55	0.95	0.07	0.29
lda	1.65	6.54	2.83	45.45	4.46	1.56	3.51	16.38	13.39	0.92	3.31
logreg	9.03	11.12	4.87	23.44	1.75	0.82	6.76	34.05	5.94	0.19	2.04
lssvm	0.09	14.93	6.99	30.51	16.11	0.17	0.45	6.86	14.00	8.70	1.19
naiveBayes	1.26	9.42	2.15	28.87	11.56	1.77	4.80	11.08	24.56	1.29	3.26
nnet	78.77	1.43	0.66	8.80	0.57	0.27	0.97	6.40	1.40	0.06	0.68
randomForest	79.76	1.52	0.14	12.55	0.20	0.06	1.19	3.91	0.43	0.11	0.14
rda	0.02	4.63	6.50	26.10	11.54	3.13	3.63	9.54	22.56	9.21	3.14
rpart	19.14	3.38	1.37	35.85	2.17	1.03	4.29	24.48	5.50	0.24	2.56