METAL

The METAL Machine Learning Experimentation Environment V3.0 (METAL–MLEE)

Manual – Version 3.0

Johann Petrak
johann@ai.univie.ac.at

Austrian Research Institute for Artificial Intelligence

October 17, 2002

Contents

1 Introduction

This manual describes the *METAL Machine Learning Experimentation Environment* (METAL–MLEE for short) and how it is used in a meta–learning setting. METAL–MLEE is a set of programs, supporting files, and standards that allow the organized, self–documenting and distributed execution of machine learning experiments. The results obtained by these experiments can be used as new meta–data for the METAL Data Mining Advisor *(cite: theadvisorstuff)*.

This manual is an abridged version of [Petrak 2002a] and contains additional information about the use of METAL–MLEE in the METAL meta–learning context.

Additional information on other components that are needed for meta–learning can be found in *(cite: rankingstuff?)* and *(cite: DCT/GSI stuff)*. The main source of information for the METAL–tools in the internet is `www.metal-kdd.org`.

METAL–MLEE helps in obtaining the meta–data for new databases and algorithms that is needed for meta–learning. It is a set of programs and Perl–scripts that help you run the necessary experiments in an orderly fashion and create a set of standardized output files.

METAL–MLEE can be used to obtain *error estimates*, *CPU time measurements*, and other data about the performance of machine learning algorithms on a specific database. These measurements will automatically be stored in a set of output files, together with other data like *database characteristics*, a description of the experimentation environment and parameters, and a detailled log of the experiment.

The output files serve as the basis for meta–learning, providing the necessary data for the *data mining advisor (cite: doc about local dma)* or other meta–learning schemes.

A more detailled description of what the METAL–MLEE does is given in Section 3.

2 Installation

2.1 System Requirements

METAL–MLEE has been developed and tested for SPARC computers running the Solaris operating system, however it will work on most LINUX and probably also other UNIX systems. It also works on a MS-Windows (32-bit) system with the support of the free *cygwin* software (see `www.cygwin.com`). The known limitations of functionality for certain configurations and operating systems are mentioned in Section 5 and in the descriptions of the individual programs. The functionality under MS Windows is restricted due to poor support for ressource limitation and CPU time profiling that is available on the command level for this family of operating systems (the CygWin package only provides a subset of the functionality needed to fully support all features of METAL–MLEE).

2.2 Software Requirements

The following software is needed to be installed on your system in order for METAL–MLEE to work.

- Perl version 5.000 or higher. The `perl` command must be in the binary search path. For the Perl-scripts to work as commands, the `perl` binary or a link to it must be available in `/usr/local/bin`. Otherwise, the scripts must be invoked using the `perl scriptname` syntax.
- A C/C++ compiler, preferably `gcc/g++` This is not needed if you have a computer with one of the hardware architectures and operating systems for which precompiled binaries are included (see below)
- Gnu make version 3.77 or higher (earlier version might work but have not been tested). This is only needed if you need to compile binaries yourself.
- `XLISPSTAT` version 3.51 or higher. XLISPSTAT is a portable LISP system with statistical functions, it is available from `http://www.stat.umn.edu/~luke/xls/xlsinfo/xlsinfo.html`. The `xlispstat` command must be in the binary search path.
- the program `md5sum` from the free portable GNU `textutils` package. This package can be obtained from `http://www.gnu.org/software/textutils/textutils.html`. This program is not essential, however. If it is missing, no MD5 hashes will be calculated for the databases. The `md5sum` command should be in the binary search path. If it is not, the location can be specified in the configuration file `config.pm`. The subdirectory `/src/md5` also contains a version of that program that can be built and installed into the `bin` subdirectory using `make`. To use that version, `config.pm` needs to be adapted accordingly.
- In addition you need one or more learning algorithms for use with METAL–MLEE (see Section 6 for more information on the requirements for learning algorithms).

On most LINUX or UNIX–like systems `perl`, `md5sum`, a compiler, and `make` will already be installed.

For MS–Windows computers, the easiest way to get all these for free is to install the CygWin software package (`www.cygwin.com`).

2.3 Obtaining and Installing the Program

METAL–MLEE is distributed as gzipped tarballs from `www.metal-kdd.org/download`. You must download the sources as `mlee-metal-src.tgz`. Optionally, you can download precompiled binaries for x86/linux (`mlee-metal-binpclinux.tgz`), x86/windows with cygwin (`mlee-metal-binpccygwin.tgz`) or sparc/solaris (`mlee-metal-binsparcsolaris.tgz`).

Uncompress and extract the archive `mlee-metal-src.tgz`. This will create a directory with the name `mlee` that contains several subdirectories.

If you also downloaded an archive with binaries, uncompress and extract this one too - it will create an additional subdirectory `binpccygwin`, `binpclinux`, or `binsparcsolaris` within the `mlee` directory. Change into that directory and check if you can run the precompiled binaries: type `./dct -h` and `./shuffle -h`. Both commands should produce as an output a short summary of command line options. If this works, copy all files into the directory `../bin`.

If you cannot or do not want to use the precompiled binaries, compile the programs: Change into the `mlee` directory and type `make all`. This will compile the DCT and shuffle programsand place it into the subdirectory `bin`.

If you do not have the `md5sum` command already installed and working on your system, type `make md5sum`. This will compile the md5sum program and place it into the subdirectory `bin`. You have to change the setting for the `$MD5BIN` variable in the configuration file `scripts/config.pm` to indicate the complete path of the `bin` directory.

Finally you probably need to adapt METAL–MLEE to the learning algorithms you are using. Section 6 gives instructions how to do that.

METAL–MLEE is now ready for use. If you wish you can place the subdirectory `script` in the binary search path. You can then directly invoke the commands from there by simply entering the name *provided* the `perl` program is installed in `/usr/local/bin`. If `perl` is installed somewhere else you can either make a link in `/usr/local/bin`, or change the Perl–scripts so that the first line indicates the correct position, or simply invoke the scripts like this: `perl <fullpathnameof-script>`.

2.4 Source code by other authors included in the package

Nearly all of the programs included in the METAL–MLEE package has been written by the author of this document. However the following code by other authors is included:

- The `check_database.pl` script has originally been written by Carlos Soares.
- The `shuffle` program uses a C++ wrapper to the C-code of the *Mersenne Twister* pseudo random number generator, written by Takuji Nishimura. The original C-code for PRNG is licensed under the GPL.
- the Source code for the md5sum program included in the `src` directory is copyrighted to RSA Data Security, 1991-1992. See the comment in the file `md5c.c` for details.

3 What METAL–MLEE Does

The main purpose of METAL–MLEE is to obtain meta–data, i.e. data about the performance of learning algorithms on different databases (these databases that are used to gather performance data for the learning algorithms are also called *base*–databases to distinguish them from the databases of meta–data obtained in the process). The term *performance* of an algorithm includes such measurable properties as *estimated error on unseen data*, *CPU time needed* for the training and the evaluation phase, and *complexity* of learned model.

METAL–MLEE can be used to:

- Check if the format of a base–database conforms to the standard database format (see Section 4).
- Carry out an error estimation experiment to obtain error estimates for one or more learning algorithms for a base–database. Error estimation strategies include crossvalidation, holdout estimates and leave-one-out.
- Obtain *data characteristics* for a base–database
- Obtain additional *database measurements* for a base–database.
- Calculate error estimates and other statistical measures to describe the performance of the learning algorithm and statistically compare different learning algorithms.
- Document the details of an experimentation run for future reference.
- Extract a *meta database* from the output files created for each experiment.
- Manage and make comparable experiments that were carried out on different machines.
- Normalize CPU time measurements for measurements obtained on different machines.

For use with the data mining advisor (*(cite: advisor stuff)*), METAL–MLEE is used in a standardized way. This is described in Section 7.

4 Standard Database Format

In order to be usable with METAL–MLEE, databases must be in a standard format. This format is similar to the formats used by the *C4.5* [Quinlan 1993] and *C5.0* machine learning algorithms, but with additional constraints. If your database does not conform to the format explained below in more detail, it needs to be converted.

For each database, two files are required: one file, the *data*–file that contains the actual data in ASCII-coded, comma–separated variables (CSV) format, and another file, the *names*–file that contains the names and types of the variables in the data file. A convention that must be observed for use with METAL–MLEE is that both files must have the same name and be located in the same path, but differ in their file name *extension*: the data file has the extension `.data` while the names file has the extension `.names`. The part of the filename without the extension that is necessary to uniquely identify a specific pair of files for a database is called *filestem*.

Hence, a database for use with METAL–MLEE always consists of two files, the names and data files, and can be specified by the part of the name that is common to both files, the filestem.

METAL–MLEE can handle both regression and classficiation problems, i.e. both numeric and discrete target variables. In both cases, the target variable has to be the last variable in the comma-separated list of fields that make up the individual records in the data file.

The restrictions on the format of the database have been imposed to be able to use as many learning algorithms as possible without having to perform costly database format conversions. Note that depending on which learning algorithms you use and how the interface scripts that plug these learning algorithms into METAL–MLEE are written, it might be possible to use a format that does not obey all of the constraints given below. For example, the limitation that the labels used for classes may not be used for discrete attributes has been introduced to make it easier to support the `ripper` rule learning algorithm. If you do not use this algorithm or if you enhance the interface script for this algorithm, that constraint on the databases need not be enforced any longer.

4.1 Names File

The names file describes the name and types of the fields, or *attributes*, in the data file. The format of the names file differs slightly if the target variable is continuous (i.e. the database is used for a regresion problem) or discrete (the database is used for a classification problem):

- The first line for a classification database contains a comma separated list of possible class labels and is terminated by a dot. Each class label must be a valid atom (see below for the definition) that does not occur as a value of any of the other discrete attributes.
- The first line for a regression database contains the name of the last attribute defined in the names file, followed by a dot.
- All other lines contain attribute descriptions, in order of appearance of the corresponding fields in the data file.
- An attribute description consists of an attribute name that starts in column 1, followed by a colon and a blank, followed by either the word "continuous" for real–valued attributes or a comma separated list of values for a discrete–valued attribute. Discrete values must be atoms and cannot be integers.
- All attribute descriptions must be terminated by a dot.
- Names must be atoms.
- For classification databases, the values for discrete attributes may not include values that are used as class labels.
- The names file contains nothing else. More specifically, it must not contain any comments as allowed for `C5.0` nor any blank lines.
- The missing value indicator is not part of the value list or otherwise listed in the attribute description.

An *atom* is a string of characters that does not contain blanks, other whitespace, special characters or accented characters and has a maximum length of 32 characters. The string can contain numeric digits, but must start with an alphabetical character.

4.2 Data File

The data file contains one new-line terminated record for each case in the database. Each record is a comma separated list of either numeric values, atoms, or the missing value indicator.

- Every atom that occurs in a field must be mentioned in the corresponding attribute description in the names file.
- Numeric values must be represented in a way that can be read in with the C `scanf` function using the “%g” directive.
- The missing value indicator for both numeric and discrete fields is an unquoted question mark.
- The atoms used for discrete fields must not be quoted.
- Data records must not be terminated by a dot.

4.3 Formal Format Description

Here is a definition of the file formats, in a meta–language similar to Backus-Naur with Perl–like regular expressions. NUMVALUE is not defined – it should be a string that can be parsed to a numeric value by the C `scanf` function and format directive “%g”.

```
atom := [a-z][a-z0-9]{0,29}

namesfile := targetline NEWLINE (attrdefline){1,N}
targetline := attrname DOT | labellist DOT
attrname := atom
labellist := atom [COMMA atom]*
attrdefline := (attrname COLON SPC labellist) |
               (attrname COLON SPC ’continuous’) DOT
DOT := ’.’
COLON := ’:’
COMMA := ’,’
SPC := ’ ’
datafile := (valuelist NEWLINE ){1,N}
valuelist := value COMMA value [COMMA value]*
value := (atom | NUMVALUE)
```

5 The Programs

The following section describes each of the programs in the METAL–MLEE in more detail. Note that all programs will accept the `-h` option that will show an explanation of all valid options and the version and versiondate of the program. The following documentation only contains explanations of those options that are releveant for the use of METAL–MLEE for use with the *datamining advisor*. For a more complete documentation of the programs see [Petrak 2002a].

5.1 Main experimentation program: `run_exp`

The `run_exp` program performs the follwing tasks for a given base database:

- Optionally run the data characterization program
- Run a parallel error estimation for a list of specified learning algorithms for which *interface programs* (see 5.2) exist. A 10-fold crossvalidation procedure will usually be carried out for this, but other estimation procedures can be specified.
- Create a file with the correct target values for each estimation fold.
- Create files with the predicted target values for each fold and learning algorithm.
- Measure thge CPU times for each fold and learning algorithm spent for the training and the evaluation phase.
- Create a log file that contains the details of the experiment and create the *results*–file that contains a machine–readable set of meta–data about the experiment.
- Optionally create performance statistics for all algorithms and algorithm pairs by running the `run_stats` program. This will create the *stats*–file with a machine readable set of meta–data about learning algorithm performance.

For the error estimation, the input base–database will be randomly shuffled and split into one or more pairs of training- and evaluation data. The program `run_exp` needs a random seed to control the random shuffling. The random seed defines the exact way how the data is being shuffled and partitioned. This allows to run the program on different machines, at different times, with different learning algorithms and still obtain comparable error estimates and comparable files with predictions.

The program creates a standardized set of output files in the output directory specified (see Section 8).

5.1.1 Synopsis

```
run_exp -h
run_exp -f stem -s seed [-v] ...
```

5.1.2 Important Options

The following describes just the subset of options that are important for use in the METAL-setting:

`-h`: Show detailled usage information, defaults, and program version.

`-f stem`: The filestem, including the full path to the location of the database. In other words, the full filename of the data or names file without the extension (requried).

`-s seed`: The seed to be used for the random number generator that determines how the data file will be shuffled before the estimation procedure is being carried out. If no seed is given, the value 1 will be used. The special value "norand" will supress random shuffling and keep the ordering of the database file. This parameter is ignored for estimation strategy "leave one out".

`-regr`: Indicate that the database describes a regression problem (i.e. the target variable is numberic). If omitted, a classification problem (i.e. the target variable is discrete) is assumed.

`-dt path`: The path to the directory that should be used to store temporary files. Default is `/tmp`. This directory must be on a device that has enough free space to hold all the intermediate files. Note that unless option `-k` or `-d` or `-lad` is specified, temporary files should get removed at the end of an experiment. However due to several reasons the directory can fill up with leftover files, so be sure to remove unneded ones regularily.

`-d`: Switch on debug mode: this will show much more information in the log file and on the console (-d implies -v which will show everything that goes into the logfile on the console too)

`-lad`: Switch on debugging for interface scripts. This will pass the option `-d` to all the interface scripts, causing a *lot* more output from the interface scripts get logged in the logfile.

5.1.3 Specifying a CPU time limit

METAL–MLEE allows you to specify a CPU time limit for each call of an external learning algorithm programm. This is necessary since otherwise the only way to end an experiment where one of the algorithm loops or takes too much time would be to terminate the whole experiment, loosing all the data for all the algorithms. You can specify the CPI time limit, in seconds, using option `-t`:

```
run_exp -f stem -s 1 -t 3600
```

This example sets the CPU time limit to one hour. (The default is 43200 seconds, or 12 hours, use the value 0 to unlimit CPU time usage)

Note however that not all OS's support this. Currently this is not possible under Windows. On some systems that do not support this but do support killing processes the coded workaround that tries to kill the process after a specfified number of *elapsed* (not CPU!) seconds might work, but this is not guaranteed either.

5.1.4 Specifying learning algorithms

Learning algorithms are always invoked through *interface scripts*. If you want to use a learning algorithm with METAL–MLEE for which there is not already an interface script included in the `scripts` subdirectory, you need to create a new one (see Section 6). Interface scripts for learning algorithms are named `run_cla_laname` for classification algorithms and `run_rla_laname` for regression algorithms, where `laname` is the name under which the learning algorithm should be known to METAL–MLEE.

To invoke one or more algorithms for an experiment, give this name of the algorithm as an argument to the `run_exp` option `-l`. In order to use more than one algorithm, specify the option `-l` multiple times, e.g.:

```
run_exp -f somestem -s 1 -l alg1 -l alg2
```

This example shows how to specify to run algorithms `alg1` and `alg2`.

Instead of specifying the list of learning algorithms every time, you can specify the list to be used as a default in the configuration file `config.pm`.

5.1.5 Passing parameters to the learning algorithms

Interface scripts both execute the training and the prediction phase of a learning algorithm. In order to specify to which phase the parameter should be passed, you need to specify a “sub option”:

```
run_exp -f stem -s 1 -l "alg1 -at -A"
  -l "alg2 -ae '-r 1.1 -s 2.1' "
```

This example shows how to add the option `-A` to the call of algorithm `alg1` for the training phase and options `-r 1.1 -s 2.1` to the testing phase of algorithm `alg2`. You can use suboption `-a` to specify what to pass to both the training and the testing phase calls.

You can also use the same algorithm twice with different parameter settings. However for this to work, you also have to specify different *algorithm suffices* for each of the different calls:

```
run_exp -f somestem -s 1 -l 'alg1 -at "-c 0.1" -asuf c0.1'
  -l 'alg1 -at "-c 0.2" -asuf c0.2'
```

This suffix will be appended everywhere the algorithm name is mentioned, i.e. statistics, the log and results file will now contain entries for an algorithm `alg1c0.1` and an algorithm `alg1c0.2`.

5.2 Algorithm interface programs

Interface programs are used to provide the main experimentation program `run_exp` with one standard interface to many different learning algorithms. In order for this to work, learning algorithms must fulfill some requirements that are listed in Section 6 which also explains how to adapt and add interface programs for new learning algorithms.

Interface programs must reside in the same directory as the `run_exp`. The follow a simple naming scheme: `run_cla_xxx` for an interface program to a classification learning algorithm named `xxx` and `run_rla_yyyy` for an interface program to a regression learning algorithm named `yyyy`. To specify a learning algorithm as an argument to `run_exp` or in the configuration file `config.pm` only the name of the learning algorithm must be given (i.e. `xxx` or `yyyy` only).

All interface programs take the same set of options. For testing purposes, or when debugging problems encountered during the execution of `run_exp` it can be useful to directly run an interface program. For this, a pair of training and testing datasets and a names file must exists (i.e. three files with the same filestem and the following extensions: `.data`, `.test` anda `.names`).

Here are the most important options for manually running an interface script:

`-h`: Show all possible options and a short explanation.

`-istem STEM`: The file stem (including the path) that identifies the three files (data, test, and names file) needed. When invoked from within the `run_exp` program the filestem will usually also include the seed and the process ID to avoid duplicate file names for the temporaryly created files.

`-tmppath PATH`: Where to store intermediate or temporary data. This is currnetly not used by `run_exp` since the training/test/names files are stored in the temporary directory anyways and it is easier to derive other filenames for temporary files directly from this filestem.

`-a args`: Pass additional arguments to all calls of the algorithm (training and testing)
`-at args`: Additional arguments for the training call
`-ae args`: Additional arguments for the testing (evaluation) call
`-cpulimit n`: Try to limit the CPU time limit to that many seconds (might not work on all systems)
`-kmodel file`: Copy the model to this file
`-nopgm` dont actually call external programs, for debuggin
`-portable`/`-noportable`: Usually the program tries to figure out how to limit CPU time and how to determine the system/user CPU time needed for the algorithm on a specific system. The `-portable` switch can be used to run (experimental) code that will try to do everything with Perl–code that is as portable as possible. Note that portable mode still has its flaws – especially the termination of processes does not work correctly on most systems. If the `-portable` option is used, `-cputime` limit will be interpreted as a limit for *elapsed runnning* time instead of CPU time.
`-k`: Do not delete intermediate datasets
`-d`: Switch on debug mode
`-v`: Switch on verbose mode

5.3 Extracting information from the results: `parse_results`

This program will make it easier to extract the interesting information from the files generated for an experiment. The standard files that are normally created are the files ending in the following extensions: =.results=, =.dct=, =.log=, and =.stats= (see Section 8). The =.log= file contains a log of all actions performed and the other three files contain result data (and are often collectively referred to as *result files*). These three files contain lines of the format:

```
Some qualified variablename: value
```

Each line contains a value for a variable. The value is everything after the colon (a value can be multidimensional, i.e. consist of more than one word, but usually just is a single word or number). The variable name is everything before the colon and consist of several words. The following line gives the value of the error estimate for algorithm =c50boost= in cross validation fold 2 of repitition 0 in a `.stats` file:

```
Error c50boost 0 2: 0.34123110000
```

The =parse_results= program can be used to extract the values for certain variables and create a file that contains just the values of these variables, separated by commas organized by lines. The program can be used to generate one line for each filestem, one line for each filestem/algorithm combination or one line for each filestem/algorithm/crossvalidation-fold combination or one line for each filestem/pair-of-algorithms combination.

The following example demonstrates how the program can be used to extract different types of data:

```
% ls
allrep_2.dct      allrep_2.stats  led24_2.results  segment_2.dct      segment_2.stats
allrep_2.results  led24_2.dct     led24_2.stats    segment_2.results

% parse_results *.* -f %DS -f %LA -f stats.Error
allrep_2,basedef,0.032873806998939555
allrep_2,basedef200,0.032873806998939555
allrep_2,baserand,0.9899257688229056
```

```
allrep_2,c50boost,0.009544008483563097
allrep_2,c50rules,0.009278897136797455
...
allrep_2,clemRBFN,0.032873806998939555
allrep_2,lindiscr,0.08510074231177095
allrep_2,ltree,0.008748674443266172
allrep_2,mlcib1,0.024920466595970307
allrep_2,mlcnb,0.05726405090137858
allrep_2,ripper,0.010604453870625663

% parse_results *.* -breakup ds -f %DS -f results.DBSize -f results.N_discrete_attr
allrep_2,3772,21
led24_2,3200,24
segment_2,2310,0
```

5.3.1 Synopsis and options

```
parse_results -h
parse_results filelist -f fieldspec [-f fieldspec ...]
  [-breakup ds | la | lapair | foldla]
  [-o outfile] [-n outnamesfile] [-fn] [-hostnorm] [-algnorm alg]
  [-s sep] [-m mv] [-mnp x] [-strip]
  [ignoredc] [-ignoreresults] [-ignorestats]
```

`filelist`: The list of files to process. The easiest way to do this is to use a glob-pattern. For example, if there is a subdirectory below the current directory for each filestem and you want to process all results files for all filestems, the simplest way to specify this is “`*/*.{dct,results,stats}`”.

`-f fieldspec`: This option can occur more than once and specifies (in order) the list of fields to include in the output. A `fieldspec` is either a qualified fieldname, a special fieldname or a function. A qualified fieldname is of the form `filspec.fieldname` where `filespec` is one of `stats`, `dct`, or `results` and the `fieldname` is the name portion of one of the fields that occur in that file, e.g. `dct.Nr_attributes` or `stats.Error`. The following special field names canbe used: `%LA` the name the learning algorithm (not for breakup=ds); `%DS` the filestem as extracted from the file processed (i.e. this will usually include the seed and eny suffixes - the ’true’ filestem can be extracted using `results.Filestem` or `results.InFilestem`; `%FLD` the fold number (only for breakup=foldla); `%REP` the repeat number; `%LA1` and `%LA2` the names of both learning algorithms for breakup=lapair. Functions must get specified in the form `NAME(arg)`. The following functions are currently defined: `AVG`, `SUM`, `COUNT`, `MIN`, `MAX` will all calculate the corresponding function over all fields that match a regular field name pattern. For example to find the maximum value for all fields with a name that starts with `Attr_Count_All_Value` in the dct file, use `’MAX(dct.Attr_Count_All_Value.*)’`. Note that the pattern must be a Perl–type regular expression, not a glob pattern. This featrue cannot be used to calculated functions over qualified variable names, e.g. `’MAX(results.Traintime.*)’` with breakup=ds will *not* work. The function `ACC(field)` will calculate `1-field`.

`-breakup x`: Specify for which level of detail the program will create individual lines in the output. the default is `la`, which produces one line for each combination of filestem and learning algorithm. The option `lapair` will generate one line of output for each combination of filestem and pairs of learning algorithms, `ds` generates one line of output for each filestem and `foldla` generates one line for each combination of filestem, learning algorithm and crossvalidation fold

`-o filename`: Specify a file where to write the output to (if not given: standard output).

`-n filename`: Specify a file where to write a C4.5 names file for the output – the program will try to guess the type and possible values of attributes and will also try to convert field names to something that is usable with most learning algorithms that use C4.5 format. Note that the generated file will just contain a line for each field in the output and is thus not directly usable for C4.5 (for this you need to remove the line for the last field and add a class label definition line at the beginning instead).

`-fn`: include a line with fieldnames as the first line of output – this is useful for many programs that can process CSV files (e.g. R).

`-hostnorm file -algnorm alg`: Specify the name of a file that contains host normalization data. All fields continaing the string "time" will then automatically get normalized based on the timing factors for each host. If `algnorm alg` is given, the times will be expressed as a multiple of the time the algorithm `alg` needed. For more information on time normalization see the next section.

`-s sep`: Use `sep` to separate fields instead of commas

`-m mv`: Use `mv` instead of a question mark to indicate missing values.

`-mnp x`: Use `x` instead of `mv` to indicate a value for which no field has been found in the input files.

`-strip`: Strip strange characters from all non-numeric output. This can help to make the output more easily digestable by other programs.

`-ignoredct`, `-ignoreresults`, `ignorestats`: Do not process the corresponding files. This can speed up processing significantly.

5.4 Normalize time measurements: `parse_times`

The METAL–MLEE package is intended to simplify the process of obtaining machine learning experimentation results that possibly get carried out on different hosts. The `run_exp` script collects the timing information returned from the interface scrips and puts them into the `.results` file. However, CPU time measurements obtained on different hosts are not comparable. The task of `parse_times` is to analyze the experimentation results that were obtained on different machines for the same dataset, using the same seed and algorithms. From the times measured on different machines, the program will create a table of factors which roughly represent the relative performance increase or decrease relative to one reference host. The table generated can then be used by the `parse_results` script normalize all time measurements to the reference machine.

WARNING: this feature should be used with extreme caution! You should be aware that the factor can only be used as a very rough aproximation to the speed differences between two machines. Several factors make this approach rather inaccurate:

- the CPU time measurement itself can depend on the load on the system and other factors that vary over time on the same system.
- any inaccuracies will be multiplied if the measurements are close to the measurement resolution of the system. Because of this the `parse_times` program will ignore all time measures < 0.1.

- different machines will optimize different instruction mixes and thus the speedup depends on the instruction mix needed for a specific execution. This means that different learning algorithms on the same dataset can show different speedups, and that the same algorithm will show different speedups for different datasets.

5.4.1 Synopsis and options

```
parse_times -host hostname -from YYYYMMDD -to YYYYMMDD
  [-calc avg | last | median] [-xlispstat filename]  filelist
```

`filelist` A list of `.results.` files to process, each containing timing information for the same set of learning algorithms on the same dataset.

`-calc x` What to do if several measurements for the same algorithm and host are found (this will be the case if the experiment gets repeated on the same machine and the `run_exp` option `-o` is *not* given, causing all results to get appended in the same file instead of overwriting old results). Possible values are: `avg` – calculate the average; `median` – calculate the median; and `last` – use the last (most recent) value found.

`-xlispstat filename` write data for subsequent processing in XLISPSTAT or LISP to this file.

`-from YYYYMMDD -to YYYYMMDD`: The generated table will contain this date as the date identifying the start and end of the validity period for the factors. Since machines can get upgraded or other things can change significantly over time that will influence the speedup factor, you can restrict the validity of the factor to a certain time period. The `parse_results` program will automatically use the factor from the correct time period based on the experimenation date found in the results files.

5.5 Checking the database format: `check_database.pl`

The script `check_database.pl` will check the format of a database for compliance with the standard database format needed by METAL (see Section 4). Note that unless you specify the option `-nocheckformat`, this script will automatically get called from `run_exp` in order to make invalid results caused by a wrong format – which might otherwise go undetected – less likely.

5.5.1 Synopsis and options

```
check_database.pl -f filestem [-regr] [-limit maxerrs] [-max maxlines]
  [-dbg] [-o]
```

`-f filestem`: Filestem (and path) of the database to process. The files `<filstem>.data` and `<filestem>.names` must exist.

`-regr`: Indicate that the database is for a regression, not classification problem.

`-limit n`: Limit the number of errors reported to `n`.

`-max n`: Limit the number of input records to be processed. This will increase speed but decrease to likelihood of finding rare errors.

`-dbg`: Switch on debug mode

`-o`: Save the output in a file with the name `<filestem>.check_metal`

5.6 Checking experiment output: `check_results.pl`

A single run of `run_exp` can create many files and a a very large `.log` file, so it is often hard to quickly determine if some algorithm failed and in which fold of the experiment. The `check_results.pl` makes this easier.

5.6.1 Synopsis and options

```
  check_results.pl -h
  check_results.pl -f stem [-N n] [-l alg1 [-l alg2] ...] [-v] [-d] [-dd]
```

`-f stem` The file stem of the files to check *including* the seed, i.e that part of the filename up and including the seed.
`-N n` The number of folds. If this is not specified the program will guess from the files it finds.
`-l alg` Can be specified more than once to provide the list of learning algorithms. If none is specified the program will try to guess the list of learning algorithms from what is there.
`-v` More verbose output.
`-d` Debug – implies -v.
`-dd` Even more debug messages.

5.7 Other programs and helper files included in the distribution

5.7.1 Calculate quick measures from names files: `parse_names`

The `parse_names` program calculates a few measures about the number of attributes, and number of values for discrete attributes from a names file. These measures are included in the `.results` file.

```
  parse_names -f namesfile
```

The output shows:

`Type_data`: The type of data file: `class` or `regr`
`N_continuous_attr`: The number of numeric attributes
`N_discrete_attr`: The number of nun-numeric attributes
`N_total_discrete_vals`: The total number of values added up over all discrete attributes.
`Avg_discrete_vals`: The average number of values over all discrete attributes.
`Log_discrete_combinations`: The natural logarithm of the product of the number of values of all discrete attributes ($\log(\Pi_1^k|a_i|)$ where $|a_i|$ is the number of values for discrete attribute number i)
`Avg_discrete_combinations`: The value of `Log_discrete_combinations` divided by the number of discrete attributes.
`N_classes`: The number of classes.

5.7.2 Select a subset of features: `project`

This script selects a list of attributes from the input files specified by the infilestem and writes a set of output files specified by the outfilestem:

```
project infilestem outfilestem attrlist
```

`attrlist` should be a comma-separated list of attribute numbers, where numbering starts with one. To pass this as a single argument it might be necessary to enclose the list in single or double quotes (depending on the shell you are using).

The script expects a `.data`, a `.names`, and a `.test` file to exist and will create the corresponding output files.

NOTE: The script uses the `cut` command internally to select the attributes. Many preinstalled `cut` commands only allow for a small number of fields and short records to be processed. Therefore, for most databases, the GNU-cut command or an equivalent version without these limitations should be used. You can specify the path to the `cut` command in the `config.pm` configuration file if it should differ from the one in the binary path.

5.7.3 Sample interface scripts

The `scripts` subdirectory in the METAL–MLEE distribution contains several interface scripts for classification learning algorithms, regression learning algorithms, preprocessing algorithms and landmark measurement algorithms. These files can be used to adapt METAL–MLEE to other learning algorithms by using them as templates.

The following interface scripts for classification learning algorithms are included:

`run_cla_TEMPLATE` A template file that is explained in greater detail in Section 6.

`run_cla_basedef`, `run_cla_basedef200`: An interface to the `baseclearn` learning algorithm, which essentially “learns” the most frequent class from the input database. The `basedef200` interface runs the `baseclearn` learning algorithm for only the first 200 records in the database. The learning algorithm is available for download from `http://www.ai.univie.ac.at/oefai/ml/metal/software/index.html#baseclearn`.

`run_cla_baserand`: Uses the `baseclearn` learning algorithm internally, but uses a random class label determined from the names file instead of the most frequent one.

`run_cla_c45rules`, `run_cla_c45tree`: An interface to a modified version of the `c4.5` and `c4.5rules` programs. The modified version of c4.5 is available from `http://www.ai.unvie.ac.at/~johann/c45oefai` (The modified version adds several new features, but the necessary features are: portability to Win32 and a program to assign class labels to a test dataset)

`run_cla_c50boost`, `run_cla_c50rules`, `run_cla`c50tree= These are interface scripts to the commercially available C50 learning algorithm, a successor of c4.5. The programs are available at `http://www.rulequest.com`. You also need a modified version of the program that assigns the class labels, which is available at `http://www.ai.univie.ac.at/oefai/ml/metal/software/index.html#c5test`.

`run_cla_clemMLP`, `run_cla_clemRBFN` These are interface scripts to the Clementine learning algorithms MLP and RBFN, respectively. The interface scripts use the program `run_clem` internally to call the Clementine learning algorithm in batch mode. See the description of that program for details.

`run_cla_lindiscr` The interface to the linear discriminant algorithm `LinDiscr`. (availabilty details?)

`run_cla_ltree` The interface to the linear tree learning algorithm `Ltree` (availabilty details?)

`run_cla_mlcib1` The interface to a 1NN learning algorithm that is based on the MLC++ machine learning library (availability?)

`run_cla_nb` The interface to a naive-bayes learning algorithm that is based on the MLC++ machine learning library

`run_cla_ripper` The interface to the `ripper` learning algorithm. The program is available from ???

The following interface scripts for regression learning algorithms are included:

`run_rla_baggedrt` The interface script to the regression tree algorithm `rt4.1`. Available from ????

`run_ral_cart` The interface to the cart learning algorithm that is implemented in `rt4.1`.

`run_rla_clemMLP`, `run_rla_clemRBFN` The interface to the MLP and RBFN learning algorithms of Clementine.

`run_rla_cubist` The interface to the `cubist` regression rule algorithm, available from `http://www.rulequest.com` You also need a modified version of the program that predicts new values, which is available at `http://www.ai.univie.ac.at/oefai/ml/metal/software/index.html#cubist_test`.

`run_rla_cubistdemo` The interface to the demo version of the `cubist` program (will only process a limited number of records) You also need a modified version of the program that predicts new values, which is available at `http://www.ai.univie.ac.at/oefai/ml/metal/software/index.html#cubist_test`.

`run_rla_kernel` The interface to the kernel regression learning algorithm that is implemented in `rt4.1`

`run_rla_lr` The interface to a linear regression model learner that is implemented in `rt4.1`.

`run_rla_mars` The interface to the `mars` learning algorithm (availability?)

`run_rla_rtplt` The interface to the ??? learning algorithm that is implemented in `rt4.1`

`run_rla_svmtorch` The interface to the support vector machine algorithm `svmtorch`, available from ???

The following interface scripts for measuring/landmarking algorithms are included:

`run_cma_lindiscr` This interface to the linear discriminant algorithm `LinDiscr` (see classification learning algorithm interfaces).

`run_cma_lm1` (experimental)

`run_cma_mlcnb` Use the `mlcnb` learning algorithm as a landmark.

`run_cma_nodes` This interfaces to the `landmarks.pl` script that calculates several landmarks. See below a more detailled description of `landmarks.pl`

The following interface scripts for preprocessing algorithms are included:

`run_cpa_disc` The interface to the discretization program `discretiser`. The interface script also needs the wrapper script `disc_wrapper.perl`. Both programs are available from ????

`run_cpa_fselC50T` The interface to a simple feature selection algorithm that uses the `c5.0` decision tree learning algorithm for a quick guess to find relevant attributes. The script also needs the `atrib_list` program and the `project` program internally. The `atrib_list` program is available from `http://www.ai.univie.ac.at/oefai/ml/metal/software/index.html#atrib_list`.

`run_cpa_fselQ1` The interface to a simple feature selection algorithm that compares the class-posterior means of attributes to guess their relevance, `fselQ1`. The program is available from `http://www.ai.univie.ac.at/oefai/ml/metal/software/index.html#fselQ1`. The interface uses the `project` program internally.

5.7.4 The Clementine command line interface: `run_clem`

The `run_clem` program simplifies the use of Clementine learning algorithms from the command line. The program analyzes the input files and creates the necessary information to modify a template Clementine stream file which is then used in a batch-mode run of Clemeninte. The `script` directory contains stream templates for the learning algorithms MLP, RBFN and C5, these are called `c5.str`, `mlp.str`, and `rbfn.str`.

WARNING: the method described has only been tested with version 5.0.1. There was an unresolved problem whith the version 5.1 when it first came out but this has not been rechecked since (which????)

```
run_clem -h
run_clem -f filestem -m method {-train|-test} [-p n|c] [-d path]
  [-r stem] [-cmd cmd] [-nc] [-i] [-s seed] [-c4] [-v] [-vl]
```

`-f filestem`: The input filestem - there must be a `.names` and a `.data` file for training mode, or a `.names` and a `-m method`: The complete filename (including the path if necessary) of the stream file template to be used.

`-train | -test`: Indicate training or test mode. In training mode, a model file is created, in test mode, the model file is used to create a file containing the predicted values for the target variable.

`-p n|c`: This is needed internally for modifying the stream file. Usually it can be guessed from the input names file. Use “n” for numeric and “c” for discrete target variables.

`d path`: The directory where generated files should be stored. These are the modified stream files, the model file, and the generated “analysis” and “matrix” files.

`r stem`: The filestem to use for the generated files. The default is `originalstem>.<method>`.

`-cmd cmd`: The command to use to call the Clementine program (default: `clementine`).

`-nc`: Do not remove temporary files after termination – useful for debugging.

`-i`: Interactive – run the generated stream in an interactive Clementine session, invoking the Clemetine GUI. This can help with finding problems and checking if everything is done correctly.

`-s seed`: A random seed - this can be used for streams that need a randomization seed internally.

`c4`: Accept data and test files where the records are terminated by a dot (if not specified: dont expect/accept the terminating dot)

`v`: Verbose output

`-vl`: Show logfile and verion info but not all the info that is shown with the `-v` option.

5.7.5 Calculate landmarks: `landmark.pl`

This Perl–script calculates landmark measurements for a database and is used internally by the landmark interface scripts.

6 Adapting METAL–MLEE

If you want to use METAL–MLEE with other learning algorithms than those for which interface scripts (see 5.2) already exist, you need to create interface scripts for that purpose.

In a similar way, you can also add additional preprocessing algorithms.

Adapting METAL–MLEE to additional algorithms essentially consists in adding the necessary interface programs. The best way to do this is to copy and adapt an existing interface program for a similar algorithm. The interface programs are written in Perl, some knowledge of Perl will be necessary to create a new interface program.

For each type of algorithm, there is a heavily commented template file that can be used as a basis for a new interface program.

In order to run a certain list of interface scipts automatically (insteading of specifying them using the `-l` option of the `run_exp` script), edit the `config.pm` script and change the lists of script names given there for the default classification, default regression, default classification data–measurement and default regression data–measurement algorithms.

6.1 Adding Learning Algorithm Interface Scripts

In order to be usable with METAL–MLEE, the folliwng conditions must be fulfilled:

- The learning algorithm should be able to read a training file that is in a format similar to the format of the `data` file described in Section 4. Similarily, the meta–data required be the algorithm should be in a from that can be derived from the names file. Unless the learning algorithm can use the data and names files directly, it will be necessary to include a conversion filter in the interface program.
- The learning algorithm should generate a persistent model file that can be used to later assign values for the target variable for new data records.
- It should be possible to separately call the training and prediction (testing) phases of the algorithm.
- The prediction/testing function of the algorithm should take the model file generated in the training phase and a `data` file and generate a file that contains only the predictions for each of the cases in the `data` file, one prediction in each line.
- NOTE: the prediction function of the algorithm should output the prediction error which needs to be captured in the interface script and returned to the main driver program `run_exp`. This program will regard a learning algorithm for which no prediction error is returned as failed and abort the testing for that learning

algorithm. If the prediction function does not output the error, a dummy value must be used unless the something went wrong (see below). The correct prediction error will be calculated by the `run_stats` program and be available in the `.stats` file anyway.

For classification learning algorithms refer to the template `run_cla_TEMPLATE`, for regression learning algorithms refer to `run_rla_TEMPLATE`.

```
#!/usr/local/bin/perl
01 use vars qw($pgmname $pgmpath $trainargs %k $args
02            $testargs  $predfile  $filestem);
03 use Getopt::Long;
04 use File::Basename;
05 $pgmname = $0;$pgmpath = dirname($pgmname);
06 push(@INC,$pgmpath);
07 require run_lib;
08 require config;
09 beginLA("LANAME","VERSION");
10 startCMD("mylearner -f $filestem -model $filestem.$la.model
   $args $trainargs");
11 my $mytmp1 = "";
12 while (defined($_=getLine())) {
13   if (/^Size: ([0-9]+)/) {
14     $mytmp1 = $1;   ## remember the number $
15   }
16 }
17 $k{"Size"} = $mytmp1;
18 $k{"Traintime"} = stopCMD();
19 startCMD("mylearner -test -f $filestem -model $filestem.$la.model
   -p $predfile $args $testargs");
20 while (defined($_=getLine())) {
21   if (/^Error rate:\s+([0-9\.]+)/) {
22     $k{"Error"} = $1;
23   }
24 }
25 $k{"Testtime"} = stopCMD();
26 rmFile("$filestem.tmp");
27 endLA();
```

Figure 1: The `run_cla_TEMPLATE` file

Figure 1 shows the `run_cla_TEMPLATE` file with all comments removed and line numbers added. To adapt the file to some learning algorithm, copy it to a file `run_cla_xxx` (for a classification algorithm) where `xxx` is the name of the learning algorithm. Follow the advice given in the comments in the template file to program the interface file for your learning algorithm.

Here a few notes on adapting the template:

- Lines 1 to 8 should be kept untouched
- The LANAME in line 9 should be the same as the algorithm name portion of the interface file (i.e. the same as the `xxx` part in `run_cla_xxx`). VERSION should reflect the version of the interface file (if you want to diferenciate between different versions of the learning algorithm, include that version info in the LANAME, e.g. `myla2.1`)
- Line 10 should contain the command needed to call the training phase of the learning algorithm. The variable `$filestem` will contain the filestem of the

three files that are prepared before each invocation of the interface file (the data, test, and names file). The variables `$args` and `$trainargs` are the values passed via the options `-a` and `-at` respectively.

- Lines 12 to 15 show a loop that will process each of the lines that is output to stdandard output or standard error by the learning algorithm. Within the loop you can search the output for (numeric) information that you want to pass back to `run_exp` which will automatically record them in the `.results` file and calculate averages. NOTE: you must process all output lines in such a while loop, using the `defined($_=getLine())` condition, or the interface will not work properly! If you do not need any information from the output, simply remove lines 13 to 15.
- Line 17 shows how to pass back the information to the `run_exp` program: simply assign the value to a hash variable where the hash key is the desired name of the variable.
- Lines 18 should be kept unchanged: it shows how to finish the `startCMD` block that was opened in line 10 and record the time used up by the training phase of the learning algorithm.
- Line 19 starts the evaluation/prediction phase of the learning algorithm by calling a different program or the same program with the apropriate options for prediction. The variable `$testargs` will contain the value for the option `-ae`.
- Lines 20–24 show how to process the standard output of the algorithm. As for the training phase all lines must be processed that way. In addition, `run_exp` will only work correctly if you pass back the prediction error in the variable `Error` as shown. If your algorithm does not output the prediction error to standard output, you can work around the problem by passing back an arbitrary value and ignoring the errors in the `.result` file later, using the errors from the `.stats` file instead (this is recommended anyway, since the errors in the `.stats` file are more accurate)
- Line 25 finishes the `startCMD` block for the prediction phase and should be kept unchanged.
- Line 26: the `rmFile` function should be used to remove any temporary and working files the algorithm might have created.
- Line 27 must be kept unchanged.

6.2 Adding Preprocessing Algorithms

Preprocessing algorithms will change the database before the learning algorithm is applied. For each `run_exp` experimentation run, you can optionally specify one preprocessing algorithm. That algorithm will be called for each fold of the crossvalidation. In order for `run_exp` to be able to call the preprocessing algorithm, an interface script must be provided.

Preprocessing algorithms, like learning algorithms, have to process the training and testing sets for each fold separately. A typical interface script will contain two calls to the preprocessing program, one for the training file and one for the test file.

Note: The preprocessing algorithm should never use information from the class labels in the test set! The preprocessing algorithm should always carry out exactly the same preprocessing transformation on the test set as on the training set. If the preprocessing algorithm adapts itself to the input dataset you must take care that this does not happen when the test set is processed! For example, a class-aware discretization algorithm should discretize the numeric attributes in the test set in exactly the same way as it discretizes the attributes in the trainingset instead of calculating new discretization intervals, based on the specific information in the test set.

This is important, because of the practical reason that otherwise the content or format of the generated training and test files could be incompatible, but more importantly, because of the theoretical reason that anything else would be cheating – using information from the test set that should be regarded as completely unavailble for the estimation procedure.

As with the interface scripts for learning algorithms, use one of the scripts included in the package as a template.

7 Running Experiments

First make sure the data is in standard format (see Section 4). The main experimentation program by default does a quick check, but you should use the checking program `check_database.pl` on the full database. Depending on the format your data is originally in, the steps to convert it into METAL-format might be very different.

Here are some hints what kind of conversion might be necessary:

- It might be necessary to convert fiels from DOS to UNIX format
- The database should be available in a format that is as close as possible to "CSV" (comma-separated values) format. Many programs that export CSV format will put non-numeric values in quotes; these have to be removed for METAL format.
- Be careful that removing special characters originally used for non-numeric values but not allowed in the METAL format will not cause several different values to get mapped to one value!
- Missing values are often coded as "empty strings". Missing values must be coded as question marks for METAL format, both for numeric and non-numeric fields.

METAL–MLEE lets you choose rather freely how to run the necessary experiments: run different algorithms on different machines, run different databases on different machines, run different algorithms on the same machine but at different times etc.

You should consider the following points when planning the experiments:

- For each experiment you should have a separate output directory. If you run different algorithms for the same database at different times on the same machine, you can simply reuse the output directory: the new target/prediction files will be added to the directory, and the `.results` and `.log` files will be appended with the new data (unless the option `-o` for `run_exp` is specified, which will overwrite the olde `.results` and `.log` files.

- If you run experiments on the same file system, take care that different experiments will not use identical files to prevent data loss. `run_exp` uses temporary file names for some files to prevent this, but output files might still be identical.

- If you run some algorithms for a database on machine A and other algorithms on machine B it is advisable to use different output directories for these runs and then merge the created files. Results must be merged by copying together the generated `.pred`, `.target`, `.dct` files and concatening together all `.results` files to the final `.results` file and all `.log` files to the final `.log` file. The script `exp_append_results` will do this for a source and a target directory: the source directory must contain a subdirectory for each filestem for which an experiment was run. The destination directory will contain a subdirectory for each filestem. Copying together is done by repeatedly compying partly results for several filestems from different source directories to the same destination directory.

- Running different algorithms on different machines will it make harder to compare CPU time measurements even for the same file stem.

- The simplest way to carry out an experiment is to run all algorithms for a filestem on the same machine in a single run of `run_exp`.

8 Structure and Organization of Output Data

8.1 The log file

For each experiment a log file with the name `filename_seed.log` is created. The log file contains the log of what `run_exp` has been doing. If `run_exp` is invoked several times for the same filestem and seed in the same output directory, each new log will be added to the end of any existing one, unless the option `-o` (overwrite) has been given to the `run_exp` command. The log will contain more information from the `run_exp` command if the `-d` (debug) option was given and will also include debuggin information from the interface scripts called if the option `-lad` was given.

8.2 The `results` file

The `:results` file contains a group of variables that describe the experiment and database, and another group of varibales that contain information for each combination of algorithm, fold, and repetition.

`File`: The full path and filestem to the database processed.

`Filestem`: The filestem without any path as used in the output files. If a suffix is added to the output file names (e.g. when a preprocessing algorithm is used), that suffix will be included here. In other words, this is the part of the filestem that will be used in the output files before the `_<seed>` part.

`InFilestem`: The filestem without any path as specified for the `-f` command line option of `run_exp`. This will never contain any suffixes.

`ModelType`: Either `classification` or `regression`

`Start`: The start date and time of the experiment, in standard UNIX date format.

`User`: The login name of the user running the experiment

`Host`: The (short) hostname of the machine on which the experiment was run.

`OS`: The (short) name of the operating system.

`System`: More detailled information about the operating system, version and architecture.

`CPUlimit`: The CPU time limit specified for this experiment; the value 0 means no limit.

`Seed`: The random seed used for the randomization of the crossvalidation folds.

`Version run_exp`: The program version of `run_exp`

`Samplespec: n/n`: The values given (or the default values) for the `-samp` and `-hsamp` options of `run_exp`.

`Preprocessing`: The name of the preprocessing algorithm, or empty

`DBSize`: The number of records in the `.data` file of the input database.

`DBdataMD5`: The MD5 key of the `.data` file. This can be used to check if exactly the same file has been used for different experiments.

`DBnameMD5`: The MD5 key of the `.names` file.

`Type_data`=, `N_continuous_attr`, `N_discrete_attr`, `N_total_discrete_vals`, `Avg_discrete_vals`, `Log_discrete_combinations`, `Avg_discrete_combinations`, `N_classes`: These values are the output of the `parse_names` program and are explained in Section 5.7.1.

`Learner`: For each learning algorithm there is one line with this key, giving the name of the learning algorithm.

`Learner_Parameters <learner>`: for each learner a line giving all the parameters as specified on the `run_exp` command line.

`DCT_Totaltime`: The total CPU time measured for the DCT algorithm, if it was run.

`Evalmethod`: The evaluation method used – one of `xval`, `holdout`, `cstho`, or `loov`.

`Evalparms`: The parameters used for the method, separated by commas. In addition, for each method, there is a special set of keywords that individually give the values for the evaluation parameters, e.g. for `xval`: `XVAL_folds` and `XVAL_repeat`.

`DBSizeTrain <r> <f>`: The actual size of the training data for repetition `<r>` and fold `<f>`.

`DBSizeTest <r> <f>`: The actual test size per repetition/fold.

`Error <r> <f> <alg>`: The holdout error (error of the learned model on the test set) for algorithm `<alg>` for that repeition/fold. This is the error as reported from the interface script, not as measured by the `run_stats` script from the target/predictions files.

`Resubsterror <r> <f> <alg>`: The resubstitition error (error of the model on the training set), if reported by the interface script.

`Size <r> <f> <alg>`: The model size, ifand as reported by the interface script.

`Testtime <r> <f> <alg>`: The time needed for the testing step in CPU seconds, as reported by the interface script.

`Traintime <r> <f> <alg>`: The time needed for the training step in CPU seconds, as reported by the interface script.

`Totaltime <r> <f> <alg>`: The time needed for both the training and testing step, in CPU seconds, as reported by the interface script. For some learning algorithms it might not possible easily to get individual training and testing times since they carry out both steps in one program run. For this only the `Totaltime` value will be different from the missing value indicator.

`Status <alg>`: The final status of the experiment for this algorithm. This is estimated from the output of the interface scripts. Either `ok` if everything worked well, `timeout` if the CPU time limit was ???, `unknown` if the status could not be determined, and `nok` if something went wrong (e.g. the algorithm crashed).

`Error <alg>`: The final average error as calculated from the individuals error reported by the interface script.

`Resubsterror <alg>`: The final average Resubstitution error.

`Size <alg>`: The final average model size.

`Testtime <alg>`: The final average testing time.

`Traintime <alg>`: The final average training time.

`Totaltime <alg>`: The final average total time.

`Stop`: The date and time when the experiment was finished, in standard UNIX date format.

8.3 The `.stats` file

The stats file contains all the measures that get calculated from the `.pred` and `.targets` files by the `run_stats` program (the `run_stats` program gets called automatically at the end of `run_exp` unless explicitly supressed).

The variables in the `.stats` file for classification–type experiments:

`Error <alg> <rep> <fold>`: The error of the model learned by algorithm `<alg>` from the training set and evaluated on the testset for repition `<rep>` and fold `<fold>`.

`Error <alg>`: The error averaged over all classifications from all folds and repitions. Note that this will be different from the average of the fold/repitions Errors above, if fold sizes are not the same for all folds.

`StdDevOfError <alg>`: The standard deviation of the errors for all folds/repitions.

`StdErrOfError <alg>`: The standard error of the errors for all folds/repitions (i.e. the standard deviation divided by the squareroot of the number of errors)

`Correct-Wrong <alg1> <alg2>`: The number of cases where the classification was correct for algorithm `<alg1>` and wrong for algorithm `<alg2>`.

`Wrong-Correct <alg1> <alg2>`: The number of cases where the classification was wrong for algorithm `<alg1>` and correct for algorithm `<alg2>`.

`pvalMcNemar <alg1> <alg2>`: The p-value of the McNemar test for identical distributions of wrong/correct and correct/wrong counts.

`pvalPairedTTest <alg1> <alg2>`: The p-value of a paired t-test for the errors.

`p-val_McNemar <alg1> <alg2>`: OBSOLETE and only kept for backward compatibility!

The variables in the `.stats` file for regression–type experiments:

`ErrorSSE`: Sum of squared errors

`ErrorMSE`: Mean squared error

`ErrorRMSE`: Root mean squared error

`ErrorNMSE`: Normalized mean squared error

`ErrorMAD`: Mean absolute differences

`ErrorNMAD`: Normalized mean absolute deviation
`RSquare`: correlation coefficient between targets and predictions
`p-MeanDiffZero <alg1> <alg2>`: p value for the test for equal means

8.4 The `.dct` file

The DCT program and its output are documented in [DCT doc].

8.5 The targets files

For each fold of the crossvalidation, a file containing only the targets of the test file for this fold gets stored in the results directory. The name of this file is of the form `<filestem>_<seed>_<fold>.targets`.

These files are necessary for the `run_stats` program to calculate error estimates and similar measures.

8.6 The prediction files

For each combination of learning algorithm and fold of the crossvalidation, a file containing only the predictions of this learning algorithm for the test file gets stored in the results directory. The name of this file is of the form `<filestem>_<seed>_<fold>_<alg>.pred`.

These files are necessary for the `run_stats` program to calculate error estimates and similar measures.

9 Solutions to frequent problems

The experiment fails and I cannot figure out why? If something goes wrong, always carefully look into the log file. If there is no hint what went wrong, repeat the experiment with the added options `-d` and `-lad`. This will create a *huge* logfile, since all the output of all learning algorithms will be included, but usually contains the crucial information about what went wrong. Some things to check for: are all programs that are needed from the interface scripts in the binary search path? The helper programs too? Is the directory used to store temporary files on a device that has enough free space to hold all the temporary files? Are there leftover temporary files from earlier runs that clobber up space?

A CPU timeout was specified, but the algorithms run much longer? On some systems – mainly Windows – the CPU limitation mechanism does not work. Unfortunately there is no solution for this as of now.

A CPU timeout was specified, but the algorithm seems to never stop? Apart from the cause given in the previous problem, it is also possible that the learning algorithm or some other algorithm that gets called (indirectly via interface scripts) from `run_exp` is waiting for input from the standard input stream. In that case, the algorithm is halted, does not consume CPU time and thus, never stops. One reason for this behaviour could be that a licensed learning algorithm is requesting a license code.

It seems for some of the folds there are empty train/test files? If the database is very small and there are many different class labels, the `shuffle` program will not be able to do stratification correctly without leaving some of the files empty. In that case, simply turn off stratification (option `-start 0`).

10 Glossary of Frequently Used and Exotic Terms

advisor : → *data mining advisor*

base database : a database that is used for experimentation to obtain → *metadata*

data characteristics : the collection of measurements obtained for a base database by the → *DCT* program and → *landmarkers*. A subset of these characteristics are used as meta–data.

database measurements : → *Data characteristics* obtained by the → *DCT* program.

DCT program : A program that calculates many different database measurements from a database. For more information on that program see [DCT doc].

data file : A comma separated variables (CSV) file that contains the actual data for a database. Each line contains one database entry as a comma separated list of ASCII values. See Section 4 for details.

data mining advisor : A web-based application that uses meta–data obtained with METAL–MLEE to build a model that will give algorithm ranking recommendations for new databases.

experiment : The process of carrying out a complete run of evaluation steps for all learning algorithms for one base database.

filestem : the common part of the the two files (the "data–" and the "names" – file) that together are used to describe a database. This is the filename without the file extension. See Section 4.

interface script : A script that makes a learning algorithm program usable with the main METAL–MLEE experimentation program, `run_exp`.

landmark : A → *database characteristic* calculated by running a fast learning algorithm on the database.

meta–data : a collection of data describing → *base databases* and the performance of learning algorithms on these base databases.

meta–database : a collection of meta–data that is used for → *meta–learning*

ranking : A recommendation of the → *data mining advisor* is given as a ranking: a ranked list of algorithms - the most recommended first, the least recommended last.

results files : The collection of all files that are generated as a result of an experiment: the `.stats` file, the `.results` file, the `.dct` file, and others (see Section 8).

.result file : One of the files that gets generated during an experiment. The file name extension of that file is `.results`, hence the name.

References

[Petrak 2002a] Petrak J.: The Machine Learning Experimentation Environment (MLEE) - User's Guide and Manual. In preparation.

[Quinlan 1993] Quinlan J.R.: C4.5: Programs for Machine Learning, Morgan Kaufmann, Los Altos/Palo Alto/San Francisco, 1993.

[DCT doc] The dct documentation.