YaDT: Yet another Decision Tree builder

http://pages.di.unipi.it/ruggieri/software

August 2017, version 2.0.1

Salvatore Ruggieri

Università di Pisa, Italy
ruggieri@di.unipi.it

Abstract. This manual describes the functionality of the YaDT system for decision tree learning.

1 Introduction

Decision trees are among the top used data mining and machine learning models of supervised learning [12]. The C4.5 algorithm is a key reference for decision tree learning. The original Quinlan's implementation of C4.5 was developed in ANSI C [6]. Several optimizations were proposed in the last two decades, addressing both the data structures holding the training set and the computation of the quality measures of continuous attributes, which is the most computationally expensive procedure [2,3,7]. *Yet another Decision Tree builder* (YaDT) [8] is a from scratch C++ implementation of C4.5 (Release 8, the latest), implementing and enhancing the optimisations proposed in the cited papers. Moreover, it offers advanced tree semplification [9], features selection [10], and meta-classifiers (random forests and bagging). YaDT is a quite paradigmatic example of sequential complex code, adopting extreme sequential optimizations both for time and memory occupation, for which the effort in designing further improvements would result in a minimal impact on the overall performances. YaDT implements then task parallel strategies at various steps:

– at tree building, a nested parallelization of sub-tree construction and attribute quality calculation (called node-attribute parallelism in [1]);
– at feature selection, a parallelization of tree construction for each tree built from different feature subsets. Notice that each tree building adopts, in turn, the node-attribute parallelism, amounting at a triple nested parallelism;
– at random forests, a trivial parallelization of tree construction for each tree in the forest. Again, nested parallelism occurs at each tree construction.

Parallelism is implemented through the Intel Threading Building Block (TBB) library[1]. This differs from the implementation in [1], where the FastFlow library[2] was used instead.

2 Distributions

YaDT distribution contains executables for 64-bit Windows and Linux OS. Distributions prior to 2.0.1 included also APIs, but, they are discontinued now. Content of a distribution:

– (Windows) the executable `dTcmd.exe` and the TBB dynamic library `tbb.dll`,
– (Linux) the executable `dTcmd` and the TBB dynamic library `libtbb.so.2`,
– a directory `data` with some datasets from the UCI Machine Learning repository [4]. Data files are in CSV format. Metadata files are in the format required by YaDT (see later).

3 Command line options

YaDT is launched using:

```
dTcmd [options...]
```

where a number of options types are provided regarding the input datasets, tree building, tree pruning, feature selection, validation, and outputs.

3.1 Input dataset options

There are two mandatory input files and two optional ones. Mandatory input files can be provided in CSV or in binary format.

[1] www.threadingbuildingblocks.org
[2] calvados.di.unipi.it

Metadata file: `-fm <file>` The mandatory metadata input file `<file>`. If the filename ends with `.gz`, it will be automatically uncompressed by YaDT. Pipe filenames can be provided. The metadata file, which typically has extension `.names`, specify the format of the other input files, and it includes three columns (without header), which in order represent:

- *feature name* can be any text.
- *feature data type* can be one of:
  `null` no data type for the values (require feature type to be `ignore`).
  `string` any string. No text qualifier character is accepted. Order of values is lexicographic.
  `ostring;<v1>;<v2>;...;<vN>` any string among `<v1>`, `<v2>`, ..., `<vN>`. No text qualifier character is accepted. Order of values is as specified.
  `integer` any 64-bit integer.
  `uinteger` any 64-bit unsigned integer.
  `float` any 32-bit float number.
  `double` any 64-bit double number.
- *feature type*, which can be
  `ignore` don't use the feature in tree building.
  `discrete` a discrete predictive feature. Splits on discrete features generate one child for each domain value.
  `continuous` a continuous predictive feature. Splits on continuous feature generate two childs, one for values lower or equal than a threshold, and the other for value higher than that.
  `weights` instance weight (a non-negative float number). No missing values admitted.
  `class` the class attribute. No missing values admitted.

For instance,the file `golf.names` contains:

```
outlook,string,discrete
temperature,integer,continuous
humidity,integer,continuous
windy,string,discrete
goodPlaying,float,weights
toPlay,string,class
```

They specify training/test data consisting of the following columns: *outlook*,which contains strings interpreted as discrete values; *temperature*,which contains integers interpreted as continuous values; *humidity*,which contains integers interpreted as continuous values; *windy*,which contains strings interpreted as discrete values; *goodPlaying*, which contains floats interpreted as weight values; *toPlay*,which contains strings interpreted as class values.

There is no specific requirement on the positions of the features, e.g., the class attribute can be in any column, not necessarily in the rightmost one.

Training data file: `-fm <file>` The mandatory training data input file `<file>`. The format of the file must be consistent with the metadata specification in the metadata file. Missing values are written as `?`. If the filename ends with `.gz`, it will be automatically uncompressed by YaDT. Pipe file names can be provided.

Metadata and training data CSV file: `-f <file>` A shorthand for `-fm <file>.names -fd <file>.data`. Sample datasets are already in files with extensions `.names` for meta-data and `.data` for data.

Metadata and training data binary file: `-bd <file>` Replaces `-fm <file>` and `-fd <file>` options, by loading metadata and data from a previously saved internal binary format (see later option `-db <file>`). **Important**: binary output/input is guarranteed to work only if `<file>` is saved and loaded using the same version of YaDT and on the same machine/OS. It does not work across YaDT versions, OS systems, or different machine architectures.

Test data file: `-ft <file>` The optional test data input file `<file>`. The format of the file must be consistent with the metadata specification in the metadata file. Missing values are written as `?`. If the file name ends with `.gz`, it will be automatically uncompressed by YaDT. Pipe file names can be provided.

Score data file: `-fs <file>` The optional score data input file `<file>`. The format of the file must be consistent with the metadata specification in the metadata file, with the exception that the class column must not be present. Missing values are written as `?`. If the filename ends with `.gz`, it will be automatically uncompressed by YaDT. Pipe file names can be provided.

CSV separator `-sep <c>` The character separator used in all input/output CSV files.

3.2 Tree building options

These options regards the construction of each single decision tree.

Split criterion: `-gain` **or** `-grpure` The default split criterion is the Gain Ratio of C4.5 [6]. It can be changed to Information Gain using `-gain` or to Pure Gain Ratio using `-grpure`. Gain Ratio normalizes Information Gain (IG) over the Split Information (SI) of an attribute, i.e., GR = IG/SI. This definition does not work well for attributes which are (almost) constants over the cases C, i.e., when SI ≈ 0. [5] proposed the heuristics of restricting the evaluation of GR only to attributes with above *average* IG. This is the default in YaDT as well. Pure Gain Ration consists of restricting the evaluation of GR only for attributes with IG higher than a *minimum* threshold [10].

Stop criterion: `-m <num>` Tree building is stopped accordingly to the C4.5 criteria when: (1) the IG of all features is below a minimum threshold; or, (2) there are less than 4 cases associated to a node, i.e., in an hypothetical binary split one of the child nodes would have less than 2 cases associated. Criterior (2) is parameterized using the `-m <num>` option, with `-m 2` as the default.the parameter).

Tree load from binary file: `-bt <file>` This options prevents building a decision tree at all. The tree is loaded from a binary file where has been previously saved (see option `-tb <file>`).

3.3 Tree pruning options

C4.5 has two pruning mechanisms. One is called *pre-pruning*, and it consists of pruning sub-trees if the root of the sub-tree has a lower or equal misclassification error rate[3] of the sub-tree when turned into a leaf node. The other is Error-Based Pruning with grafting. YaDT offers several other simplification strategies, which are described and experimented in detail in [9].

Only pre-pruning: `-np` performs only pre-pruning.

No pruning: `-npp` performs no form of pruning, not even pre-pruning.

Error-based pruning: `-ebp[g|a]` **and** `-c <num>` performs, in addition to pre-pruning, error-based pruning (`-ebp`), with grafting the largest child (`-ebpg`), or with grafting all childs (`-ebpa`). The C4.5 default is the `-ebpg` option. The YaDT deafult is the `-ebp` option. Error-based pruning requires a confidence level parameter `-c <num>`, where `<num>` must in (0, 1]. Default value is `-c 0.25`.

Minimum-error pruning: `-mep[g|a]` performs, in addition to pre-pruning, minimum-error pruning (`-mep`), with grafting the largest child (`-mepg`), or with grafting all childs (`-mepa`). Minimum-error pruning internally splits the training set into a stratified building set (70% of training), for tree building, and a stratified pruning set (30% of training), for tree simplification.

Pessimistic-error pruning: `-pep[g|a]` performs, in addition to pre-pruning, pessimistic-error pruning (`-pep`), with grafting the largest child (`-pepg`), or with grafting all childs (`-pepa`).

[3] Misclassification errors are computed by YaDT using the C4.5's distribution imputation method [11] for instances that contain missing values.

Reduced-error pruning: `-rep[g|a]` performs, in addition to pre-pruning, reduced-error pruning (`-rep`), with grafting the largest child (`-repg`), or with grafting all childs (`-repa`). Reduced-error pruning internally splits the training set into a stratified building set (70% of training), for tree building, and a stratified pruning set (30% of training), for tree simplification.

3.4 Feature selection options

Feature selection is a pre-processing step occurring before building a decision tree or decision forest. It selects a subset of the available predictive features to be used in actual training. YaDT implements a wrapper approach by splitting the training set into a building set, used to build trees for a given feature subset, and a search set, used to evaluate the built trees. Splitting is done with stratified sampling, using 70% of training data for tree building and 30% for evaluation. Once features are selected, the actual decision tree or decision forest is built on the whole training data.

Sequential forward selection: `-sfs` runs a greedy procedure that starts from zero features and it adds one feature at a time while the misclassification error increases. The feature added is the one that minimize misclassification error. The trees built are built and pruned with the same parameters specified in Sections 3.2 and 3.3

Sequential backward elimination: `-sbe` runs a greedy procedure that starts from all features and it eliminates one feature at a time while the misclassification error of the tree built does not increase. The feature added is the one that minimize misclassification error. The trees built are simplified according to tree pruning options (see Sect. 3.3).

Pruned sequential backward elimination: `-psbe` runs a greedy procedure that starts from all features and it eliminates one feature at a time while the misclassification error of the tree built does not increase. The feature added is the one that minimize misclassification error. The trees built are simplified according to tree pruning options (see Sect. 3.3). The trees built during feature selection are not simplified, i.e., the option `-npp` holds during their construction (see Sect. 3.3). If the `-gain` or `-grpure` options are not given, i.e., the default split criterion of Gain Ratio holds, the trees built during feature selection use the Pure Gain Ratio split criterion, i.e., the option `-grpure` holds during feature selection.

Acceptable feature subset: `-as <b>` using the complete search method described in [10], it finds a subset of features that is guarranteed to have misclassification error exceeding at most `<b>` the misclassification error of the best feature subset. The trees built during feature selection are not simplified, i.e., the option

`-npp` holds during their construction (see Sect. 3.3). If the `-gain` or `-grpure` options are not given, i.e., the default split criterion of Gain Ratio holds, the trees built during feature selection use the Pure Gain Ratio split criterion, i.e., the option `-grpure` holds during feature selection.

3.5 Ensemble options

Ensembles are collections of decision trees, whose predictions are made by a majority voting mechanism. Each tree in the ensemble is built and pruned with the same parameters specified in Sections 3.2 and 3.3, and using the features selected as specified in Section 3.4.

Bagging: `-ba <n>` compute a bagging of `<n>` decision trees.

Bagging: `-fo <n>` compute a forest of `<n>` decision trees. At each decision node, candidate split attributes are chosen from a random subset of $log_2\ k$ features from the k available predictive features.

3.6 Validation options

These options set the data for testing the predictive performance of decision trees. Misclassification errors are computed by YaDT using the C4.5's distribution imputation method [11] for instances that contain missing values.

Holdout random: `-h <num>` hold method: random `<num>`% is used as training and rest as test set. Here `<num>` is a float number in the range $(0, 100)$.

Holdout deterministic: `-hf <num>` hold method: first `<num>`% is used as training and rest as test set. Here `<num>` is a float number in the range $(0, 100)$. This option switches dataset loading to be single-threaded.

Holdout stratified: `-hs <num>` hold method: stratified random `<num>`% is used as training and rest as test set. Here `<num>` is a float number in the range $(0, 100)$.

Holdout deterministic stratified: `-hd <num>` hold method: deterministic stratified random `<num>`% is used as training and rest as test set. Here `<num>` is a float number in the range $(0, 100)$. This option switches dataset loading to be single-threaded.

Cross-validation: `-cv <r> <num>` cross-validation method using `<r>` runs of `<num>` folds each.

3.7 Output options

These options set the desired outputs regarding data, trees, and logs.

Binary data: `-db <file>` output dataset in binary format to `<file>`. The binary data can be loaded using the `-bd <file>` option. Binary input is much faster than loading from CSV or compressed files.

Binary tree: `-tb <file>` output tree(s) in binary format to `<file>`. The binary tree(s) can be loaded using the `-bt <file>` option.

Execution log: `-l <file>` output logs of execution to `<file>`.

Execution log: `-nl` does not output logs of execution to standard output.

Text tree: `-t <file>` output tree(s) in textual format to `<file>`.

Text tree: `-tstd` output tree(s) in textual format to standard output.

Text tree: `-oerr` in textual tree output, include evaluation measures of ordinal classification. This makes sense if the class is ordinal, and its data type in the metadata is `ostring;<v1>;<v2>;...;<vN>` where `<v1>,<v2>,...,<vN>` are the ordered class values. **Important:** at the moment, YaDT does not include specific algorithms for ordinal classification.

PMML tree: `-x <file>` output tree(s) in PMML[4] format to `<file>`.

PMML tree: `-xstd` output tree(s) in PMML format to standard output.

DOT tree: `-d <file>` output tree(s) in DOT format to `<file>`.

DOT tree: `-dstd` output tree(s) in DOT format to standard output.

Score data: `-s <file>` out predictions on the input score dataset (see option `-fs <file>`) to `<file>`.

3.8 Multi-core options

Maximum number of threads: `-tt <n>` sets the maximum number of concurrent threads to `<n>`. The default value is the number of cores as provided by the OS.

[4] http://dmg.org

4 Contacts

Please send any request to ruggieri@di.unipi.it.

References

1. M. Aldinucci, S. Ruggieri, and M. Torquati. Decision tree building on multi-core using FastFlow. *Concurrency and Computation: Practice and Experience*, 26(3):800–820, 2014.
2. U. M. Fayyad and K. B. Irani. On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, 8:87–102, 1992.
3. J. E. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest — A framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2/4):127–162, 2000.
4. M. Lichman. UCI machine learning repository, 2013. http://archive.ics.uci.edu/ml.
5. J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
6. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.
7. S. Ruggieri. Efficient C4.5. *IEEE Transactions on Knowledge and Data Engineering*, 14:438–444, 2002.
8. S. Ruggieri. YaDT: Yet another Decision tree Builder. In *Proc. of Int. Conf. on Tools with Artificial Intelligence (ICTAI 2004)*, pages 260–265. IEEE, 2004.
9. S. Ruggieri. Subtree replacement in decision tree simplification. In *Proc. of the SIAM Conference on Data Mining (SDM 2012)*, pages 379–390. SIAM, 2012.
10. S. Ruggieri. Enumerating distinct decision trees. In *Proc. of the Int. Conf. on Machine Learning (ICML 2017)*, volume 70, pages 2960–2968. JMLR Workshop and Conference Proceedings, 2017.
11. M. Saar-Tsechansky and F. Provost. Handling missing values when applying classification models. *Journal of Machine Learning Research*, 8:1625–1657, 2007.
12. X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, J. Motoda, G. J. McLachlan, A. F. M. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.