Comput Softw Big Sci (2017) 1:2
https://doi.org/10.1007/s41781-017-0002-8

CrossMark

ORIGINAL ARTICLE

# FastBDT: A Speed-Optimized Multivariate Classification Algorithm for the Belle II Experiment

**Thomas Keck**[1]

Received: 5 April 2017 / Accepted: 27 July 2017 / Published online: 29 September 2017
© The Author(s) 2017. This article is an open access publication

**Abstract** Stochastic gradient-boosted decision trees are widely employed for multivariate classification and regression tasks. This paper presents a speed-optimized and cache-friendly implementation for multivariate classification called FastBDT. The concepts used to optimize the execution time are discussed in detail in this paper. The key ideas include: an equal-frequency binning on the input data, which allows replacing expensive floating-point with integer operations, while at the same time increasing the quality of the classification; a cache-friendly linear access pattern to the input data, in contrast to usual implementations, which exhibit a random access pattern. FastBDT provides interfaces to C/C++, Python and TMVA. It is extensively used in the field of high energy physics (HEP) by the Belle II experiment.

**Keywords** Boosted decision trees · Multivariate classification · Equal-frequency binning · Cache-friendly · Belle · Hep

## Introduction

In multivariate classification one calculates the probability of a given data-point to be signal, characterized by a set of explanatory features $\mathbf{x} = \{x_1, \ldots, x_d\}$ and a class label $y$ (signal $y = 1$ and background $y = -1$). In supervised machine learning this involves a fitting-phase which uses training data-points with known labels and an application-phase, during which the fitted classifier is applied to new data-points with unknown labels. During the fitting-phase, the internal parameters (or model) of a multivariate classifier are adjusted, so that the classifier can statistically distinguish signal and background data-points. The model complexity plays an important role during the fitting-phase and can be controlled by the hyper-parameters of the model. If the model is too simple (too complex) it will be under-fitted (over-fitted) and perform poorly on test data-points with unknown labels.

Stochastic gradient-boosted decision trees [8] are widely employed in high energy physics for multivariate classification and regression tasks. The implementation presented in this paper was developed for the Belle II experiment [2], which is located at the SuperKEKB collider in Tsukuba, Japan. Multivariate classification is extensively used in the Belle II Analysis Software Framework (BASF2) [16], for instance during the reconstruction of particle tracks, as part of particle identification algorithms, and to suppress background processes in physics analyses. Often, a large amount of classifiers must be fitted due to hyper-parameter optimization, different background scenarios, to gain improved estimates on the importance of individual features, or to create networks of classifiers which feed into one another. Therefore, Belle II required a default multivariate classification algorithm which is: fast during fitting and application; robust enough to be trained in an automated environment; can be reliably used by non-experts; preferably generates an interpretable model and exhibits a good out-of-the box performance.

FastBDT satisfies those requirements and is the default multivariate classification algorithm in BASF2. On the other hand, BASF2 supports other popular multivariate analysis frameworks like TMVA [9], scikit-learn (SKLearn) [19], XGBoost [5] and Tensorflow [1] as well.

✉ Thomas Keck
thomas.keck2@kit.edu

1 Karlsruher Institut für Technologie, Campus Süd, Institut für Experimentelle Kernphysik, Wolfgang-Gaede-Str. 1, 76131 Karlsruhe, Germany

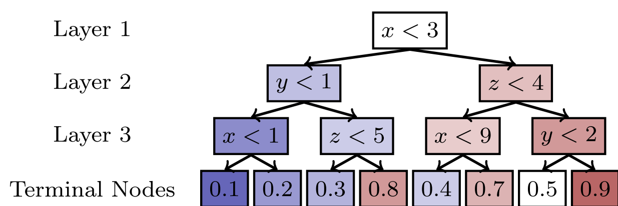


**Fig. 1** Three layer DT: a given test data-point (with unknown label) traverses the tree from top to bottom. At each node of the tree a binary decision is made until a terminal node is reached. The probability of the test data-point to be signal is the signal-fraction (number stated in terminal node layer) of all training data-points (with known label), which ended up in the same terminal node

In the following sections the stochastic gradient-boosted decision tree algorithm and its hyper-parameters are briefly described.

### Decision Tree (DT)

A DT performs a classification using a number of consecutive cuts (see Fig. 1). The maximum number of consecutive cuts is a hyper-parameter and is called the `depth of the tree` $D$.

The cuts are determined during the fitting-phase using a training sample with known labels. At each node only training data-points which passed the preceding cuts are considered. For each feature at each node a cumulative probability histogram (CPH) for signal and background is calculated, respectively. The histograms are used to determine the separation gain for a cut at each position in these histograms. The feature and cut-position (or bin) with the highest separation gain are used as the cut for the node. Hence each cut locally maximizes the separation gain between signal and background on the given training sample.

The predictions of a deep DT is often dominated by statistical fluctuations in the training data-points. In consequence, the classifier is over-fitted and performs poorly on new data-points. There are pruning algorithms which automatically remove cuts prone to over-fitting from the DT [15]. A detailed description of decision trees is available in [4].

### Boosted Decision Tree (BDT)

A BDT constructs a more robust classification model by sequentially constructing shallow DTs during the fitting-phase. The DTs are fitted so that the expectation value of a negative binomial log-likelihood loss-function is minimized. The depth of the individual DT is strongly limited to avoid over-fitting. Therefore a single DT separates signal and background only roughly and is a so-called weak-learner[1]. By using many weak-learners a well-regularized classifier with large separation power is constructed. The `number of trees` $N$ (or equivalently the number of boosting steps) and the `shrinkage` (or `learning rate`) $\eta$ are additional hyper-parameters of this model. The two parameters are anti-correlated , i.e. decreasing the value of $\eta$ increases the best value for $N$.

### Gradient Boosted Decision Tree (GBDT)

A GBDT uses gradient-decent in each boosting step to reweight the original training sample. In consequence, data-points which are hard to classify (often located near the optimal separation hyper-plane) gain influence during the training. A boost-weight calculated for each terminal node during the fitting-phase is used as output of each DT instead of the signal-fraction. The probability of a test data-point (with unknown label) to be signal is the sigmoid-transformed sum of the outputted boost-weights of each tree. The complete algorithm is derived and discussed in detail by [7].

### Stochastic Gradient Boosted Decision Tree (SGBDT)

A SGBDT uses a randomly drawn (without replacement) sub-sample instead of the full training sample in each boosting step during the fitting phase. This approach further increases the robustness against over-fitting, because the statistical fluctuations in the training sample are averaged out in the sum over all trees. The incorporation of randomization into the procedure was extensively studied by [8]. The fraction of samples used in each boosting step is another hyper-parameter called the `sampling rate` $\alpha$.

### Related Work

In general there are two approaches to increase the execution speed of an algorithm: modify the algorithm itself or optimize its implementation.

It is easy to see that the first approach has large potential and there are several authors which investigated this approach for the SGBDT algorithm: the original paper [7] on GBDTs showed that 90–95 % of the training data-points can be removed from the fitting-phase after some boosting steps without sacrificing classifier accuracy. Another approach was presented in [3] where a subset of the training data was used to prune underachieving features early during the fitting phase without affecting the final performance. Traditional boosting as discussed above treats the tree learner as a black box, however, it is possible to exploit the underlying tree structure to achieve higher accuracy and smaller (hence faster) models [20].

[1] A simple model with few parameters.

FastBDT uses a complementary approach and gains an order of magnitude in execution time, compared to other popular SGBDT implementations such as those provided by TMVA (ROOT v6.08/06) and SKLearn (v0.18.2), by (mainly) optimizing the implementation of the algorithm. After the first release of FastBDT [12], the XGBoost project (development version 22.06.2017) implemented a histogram binning[2] based on the ideas introduced by FastBDT and LightGBM[3], and observed similar speed-ups during the fitting phase in their recent version. Furthermore, the classification quality of FastBDT equals or exceeds the previously mentioned implementations in typical HEP-specific classifications tasks, which often include simulation mismodeling, measurement uncertainties and missing values. A comparison between the different implementations is presented in Sect. 4.

## Implementation

On modern hardware it is difficult to predict the execution time, e.g. in terms of spent CPU cycles, because there are many mechanisms built into modern CPUs to exploit parallelisable code execution and memory access patterns. In consequence it is important to benchmark all performance optimizations. In this work `perf` [6], `valgrind` [17] and `std::chrono::high_resolution_clock` [13] were used to benchmark the execution time and identify critical code sections. The most time consuming code section in the SGBDT algorithm is the calculation of the cumulative probability histograms (CPH), which are required in order to calculate the best-cut at each node of the tree. The main concepts used in the implementation of the fitting-phase of FastBDT are described in the following sections.

### Memory Access Patterns

At first we consider the memory layout of the training data. Each training data-point consists of $d$ continuous features, one boolean label (signal or background) and an optional continuous weight. In total there are $M$ training data-points. There are two commonly used memory layouts in this situation: `array of structs` and `struct of arrays` (see Fig. 2).

CPU caches assume *spatial locality* [13], which means that if the values in memory are accessed in linear order there is a high probability that they are already cached. In addition, CPU caches assume *temporal locality* [13], which means that frequently accessed values in memory are cached

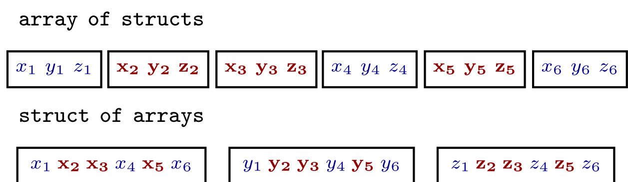


**Fig. 2** Possible memory layouts of the input features. Shown are three features $x$, $y$ and $z$ and six data-points. The *bold-red* and *blue* coloring will take on different meanings as we go through the concepts described in Sect. 2

as well. Finally, the CPU assumes *branch locality*, meaning only a few conditional jumps occur and therefore instructions can be efficiently prefetched and pipelined. The key idea is to use spatial locality (i.e., a linear memory access pattern) for the caching of the large feature input data, temporal locality for the caching of the small cumulative probability histograms (CPH), while at the same time avoiding conditional jumps as much as possible to ensure branch locality.

FastBDT uses an `array of structs` memory layout and determines the cuts for all features and all nodes in the same layer of the tree simultaneously. Several sources of conditional jumps and non-uniform memory access are considered:

*Signal and Background* The CPHs are calculated separately for signal and background (imagine the red data-points are signal and the blue data-points are background in Fig. 2). This leads to additional conditional jumps in both memory layouts. Hence FastBDT stores and processes signal and background data-points separately. Additionally this reduces the amount of CPH data which has to be cached by a factor of two, and saves one conditional jump during the filling of the CPHs.

*Multiple Nodes per Layer* In each layer of the decision tree the algorithm has to calculate the optimal cuts for multiple nodes, where each node is optimized with respect to a distinct subset of the training data-points (imagine the blue data-points belong to node A and the red data-points belong to node B in Fig. 2). In both memory layouts, consecutive memory access is only possible if the CPHs are calculated for all nodes in the current layer in parallel. Therefore FastBDT calculates the CPHs in different nodes at the same time (the CPH data is likely to be cached due to temporal locality), which allows accessing the values in direct succession (hence the input data is likely to be cached due to spatial locality). In contrast, other popular implementations determine the cuts node-after-node. Hence the data-points which have to be accessed at the current node depend on the preceding nodes and therefore this commonly used approach exhibits random jumps during the memory access regardless of the memory layout.

[2] https://github.com/dmlc/xgboost/issues/1950.

[3] https://github.com/Microsoft/LightGBM.

*Stochastic sub-sampling* Only a fraction of the training data-points is used during the fitting of each tree (imagine the blue data-points are used and the red data-points are disabled in Fig. 2). The `array of structs` allows consecutive memory access within a data-point without conditional jumps. As a consequence FastBDT uses this memory layout, but has to calculate all the CPHs for the features in parallel as well. In the `struct of arrays` layout the conditional jumps cannot be avoided without re-arranging the data in the memory.

### Preprocessing

Continuous input features are represented by floating point numbers. However, DTs only use the order statistic of the features (in contrast to, e.g. artificial neural networks). Hence the algorithm only compares the values to one another and does not use the values themselves. Therefore FastBDT performs an equal-frequency binning on the input features and maps them to integers. This has several advantages: integer operations can usually be performed faster by the CPU; the integers can directly be used as indices for the CPHs during the calculation of the best-cuts; and the quality of the separation is often improved because the shape of the input feature distribution (which may contain sharp peaks or heavy tails) are mapped to a uniform distribution (Fig. 3).

This preprocessing is only done during the fitting-phase. Once all cuts are determined the inverse transformation is used to map the integers used in the cuts back to the original floating point numbers. Therefore there is no runtime-overhead during the application-phase due to this preprocessing.

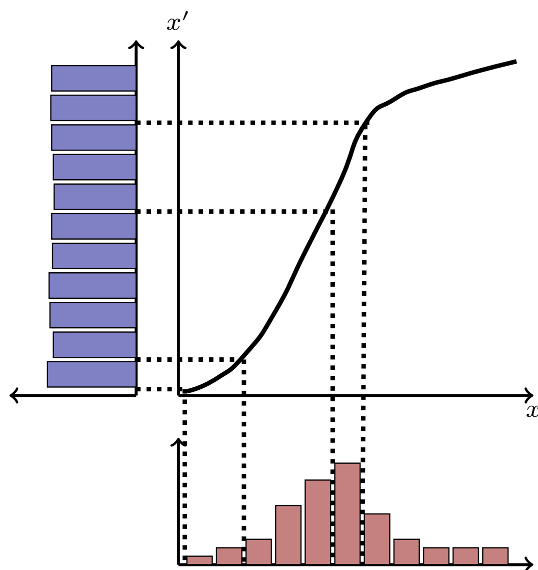


**Fig. 3** Equal-frequency binning. The feature $x$ is binned, so that in each bin there is roughly the same number of data-points. The bin boundaries are indicated by the *dotted lines*

As consequence of the equal-frequency binning is that the best-cut found at each decision tree branch is only an approximation compared to a search performed on the original dataset. The author observed an improved performance caused by this approximation.

### Parallelism

Usually one can distinguish between two types of parallelism: task-level parallelism (multiple processors, multiple cores per processor, multiple threads per core) and instruction-level parallelism (instruction pipelining, multiple execution units and ports, vectorization). It is possible to use task-level parallelism to reduce the execution time during the fitting-phase of the SGBDT algorithm. However FastBDT was not designed to do so, since our use-cases typically require fitting many classifiers in parallel and therefore already exploit task-level parallelism effectively, or use a shared infrastructure, where we are only interested in minimizing the total CPU time. Other implementations like XGBoost do take advantage of this type of parallelism.

The application-phase is embarrassingly parallel and task-level parallelism can be used by all implementations even if not directly supported[4].

On the other hand instruction-level parallelism is difficult to exploit in the SGBDT algorithm (in the fitting-phase and the application-phase) due to the large number of conditional jumps intrinsic to the algorithm, which strongly limits the use of instruction pipelining and vectorization. However, due to the chosen memory layout and preprocessing many conditional jumps in the algorithm can be replaced by index operations. Figure 4 shows an example for such an optimization (branch locality).

## Advanced Features

FastBDT offers more advanced capabilities, which are briefly described in the following sections.

### Support for Negative Weights

The boosting algorithm assigns a weight to each data-point in the training dataset. FastBDT supports an additional weight per data-point provided by the user. These individual weights are processed separately from the boosting

[4] E.g. by running the loop over the data-points in parallel using OpenMPI [18] `#pragma omp parallel for` as done in XGBoost after calling its predict function.

```
int a = 0;
int b = 0;
for(int i=0; i<1e9; ++i) {
 if(rand()%2 == 0) a++;
 else              b++;
}
cout<<a<<" "<<b<<endl;
```

**(a)**

```
int a[] = {0,0};

for(int i=0; i<1e9; ++i) {
  a[rand()%2]++;

}
cout<<a[0]<<" "<<a[1]<<endl;
```

**(b)**

**Fig. 4** Branch locality optimisation example which increments two counters randomly. Both codes were compiled with the highest optimization level (-Ofast) available using g++ 4.8.4. The optimised version is 30% faster. For comparison: incrementing a single variable by one (by a random number) without branching has a execution time of 0.5 s (6.6 s). **a** Straight-forward implementation—execution time 10.1 s. **b** If statement replaced by array lookup—execution time 6.9 s

weights. In particular FastBDT allows for negative individual weights, which are commonly used in data-driven techniques to statistically separate signal and background using a discriminating variable [14]. Other frameworks like TMVA, SKLearn and XGBoost support negative weights as well.

### Support for Missing Values

There are at least two different kinds of missing values in a dataset. Firstly, missing values which can carry usable information about the target e.g. in particle classification in HEP a feature provided by a detector can be absent because the detector was not activated by the particle. Secondly, missing values which should not be used to infer the target e.g. a feature provided by a detector is absent due to technical reasons.

FastBDT supports both types of missing values. The first type can be passed as negative or positive infinity; FastBDT will put these values in its underflow or overflow bin. In consequence, a cut can be applied separating the missing from the finite values, and the method can use the information provided by the presence of a missing value. The second type should be passed as NaN (Not a Number) floating point value according to the IEEE 754 floating point standard [10]. These values are ignored during the fitting and application phase of FastBDT. If the tree tries to cut on a feature, which is NaN, the current node behaves as if it is a terminal-node for the corresponding data-point.

### Purity Transformation

FastBDT implements further preprocessing steps based on equal-frequency binning, which calculate the signal-purity in each bin and sort the bins by increasing signal-purity.

The resulting transformed features lose their original meaning, but the individual cuts are more efficient because the features are linearly correlated to the signal-probability by construction. In contrast to the sole equal-frequency binning, this so-called purity transformation has to be performed during the application-phase as well.

### Boosting to Uniformity

Physics analyses often employ MVA classifiers to improve the signal to noise ratio, and afterwards extract physical observables by fitting a signal and background model to one or more *fit-variables*. A non-uniform selection efficiency of the classifier in the *fit-variables* leads to artificial peaking backgrounds and increased sensitivity on Monte Carlo simulation. In the past, this effect was mitigated by removing features from the training which are dependent on the *fit-variables*.

FastBDT implements the uniform gradient boosting algorithm first described by [21], which enforces a uniform selection efficiency in the *fit-variables* for signal and background, separately. Therefore no features have to be removed from the training, and the algorithm can take advantage of the complete dataset.

### Feature Importance Estimation

Many multivariate classification methods offer the possibility to estimate the feature importance, i.e. influence of the features on the decision. For BDTs the usual approach to calculate the global feature importance is to sum up the separation gain of each feature, by looping over all trees and nodes. The individual importance for a single data-point can be calculated similarly by summing up all separation gains along the path of the event through the trees.

This approach suffers from the possible correlation and non-linear dependencies between the features, as can be seen by the simple example in Table 1. In a single decision

**Table 1** Truth table of `exclusive or`. Both input features $x$ and $y$ cannot classify the target on their own

| $x$ | $y$ | Target |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

tree, one of the features will have a separation gain of zero, although both carry exactly the same amount of information.

Due to the fast fitting phase of FastBDT it is viable to use another popular approach: measuring the decrease in the performance (e.g. the area under the curve (AUC) of the receiver operating characteristic) of the method if a feature is left out. This requires $N$ fit operations, where $N$ is the number of features. The accuracy of the method can be further improved by recursively eliminating the most-important feature, which requires $\frac{1}{2}N(N+1)$ fit operations.

Although this approach is independent of the employed classification method, it only becomes a viable option with a fast and robust method.

## Comparison

FastBDT (Version 5.0) was compared against other SGBDT implementations frequently used in high energy physics: TMVA [9] (ROOT version 6.08/06), scikit-learn (SKLearn) [19] (version 0.18.2), and XGBoost [5] (development version 22.06.2017). The used benchmark dataset contains 1.6 million data-points, 40 features and a binary target. It was split into equal parts, used during the fitting and application phase, respectively. The dataset was produced using Monte Carlo simulation of the decay of a $D^0$ meson into one charged pion, one neutral pion and one kaon, a common classification problem in high energy physics. However, the nature of the data has no influence on the execution time of the SGBDT algorithm in the considered implementations, since no optimizations which prune features using their estimated separation power, as described in [3], are employed.

In a first **preprocessing** step the dataset is converted into the preferred data-format of each implementation: ROOT `TTree` for TMVA, numpy `ndarray` for SKLearn, `DMatrix` for XGBoost, and `EventSample` for FastBDT. Afterwards, the **fitting** and **application** steps are performed for each implementation. Each step is measured using std::chrono::high_resolution_clock. The preprocessing time is small with respect to the fitting time for most of the investigated hyper-parameter configurations. All results stated below are valid as well if one considers the preprocessing as part of the fitting-phase.

The wall-clock execution time[5] of each implementation is measured five times for each considered set of hyper-parameters. If not stated otherwise the following default hyper-parameters are chosen: `depth of the trees` = 3, `number of trees` = 100, `number of features` = 40, `number of training data-points` = 800000, `sampling-rate` = 0.5, `shrinkage` = 0.1. The optimal hyper-parameters for this problem depend on the implementation. FastBDT achieves its optimal performance with `number of trees` ≈ 400.

Furthermore an approximate cut-search based on a binned data distribution with `number of bins` = 256 was chosen for FastBDT, TMVA and XGBoost. SKLearn does not support approximate cut-search. All implementations have additional hyper-parameters which are not shared by the other implementations. The respective default values are used in these cases.

All measurements are performed on an Intel(R) Core (TM) i7-4770 CPU (@ 3.40GHz) with a main memory of 32 GigaByte. The code used to perform the measurements can be found in the FastBDT repository.

### Fitting-Phase

In general it is expected that the fitting-phase runtime of the algorithms scales linearly in `depth of the trees`, `number of trees`, `number of features`, `number of training data-points` and `sampling-rate`. As can be seen from Figure 5 these expectations are not always fulfilled. For each varied hyper-parameter the slope and intercept was fitted using an ordinary linear regression.

FastBDT outperforms all other implementations during the fitting phase, except for XGBoost, which implemented the histogram-based binning and caching techniques presented in this paper in their recent version.

SKLearn is the slowest contestant and also violates the expected linear runtime behaviour for the `depth of the trees` (see 5b) and the `number of training data-points` (see 5c). It is not clear why this is the case.

TMVA has a constant drop in the linear increase of the runtime as a function of the `number of features`, because at this point TMVA stops filling pairwise histograms for all features, which are used for evaluation and plotting purposes only. FastBDT and XGBoost have a linear runtime behaviour in all considered hyper-parameters, as expected.

FastBDT is the fastest contestant for small models (`depth of the trees` < 5 and `number of trees` < 300), whereas XGBoost has a slightly better scaling behaviour for large models, i.e. smaller slope, because its regularisation method includes the tree structure and effectively shrinks large models.

[5] The wall-clock time, that is the elapsed real time, is used for all measurements. In the used setup the wall-clock time is approximately equal to the CPU time.

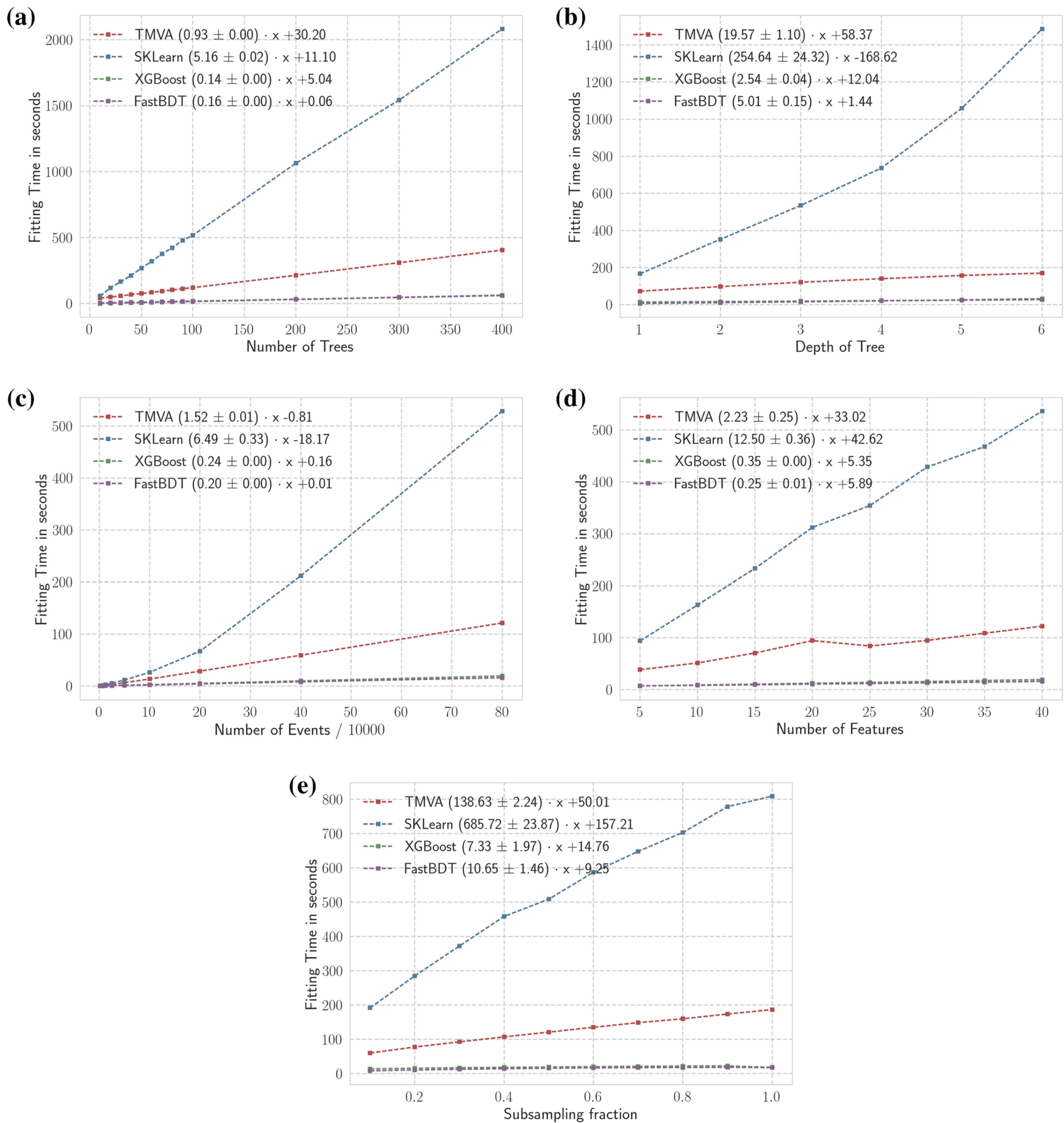


**Fig. 5** Wall-clock time for the fitting-phase as a function of different hyper-parameters. The $y$ axes show the fitting time in seconds. The $x$ axes show the varied hyper-parameter. Each configuration was measured five times: the measured minimal values are shown as *boxes* and the scaling is indicated by the *dotted line* in between the measured points. A linear regression $y = a \cdot x + c$ was performed using the minimal values of each configuration. The measured values of FastBDT and XGBoost are overlapping, and cannot be distinguished by the naked eye. **a** number of trees, **b** depth of the trees, **c** number of training data-points, **d** number of features, **e** sampling rate

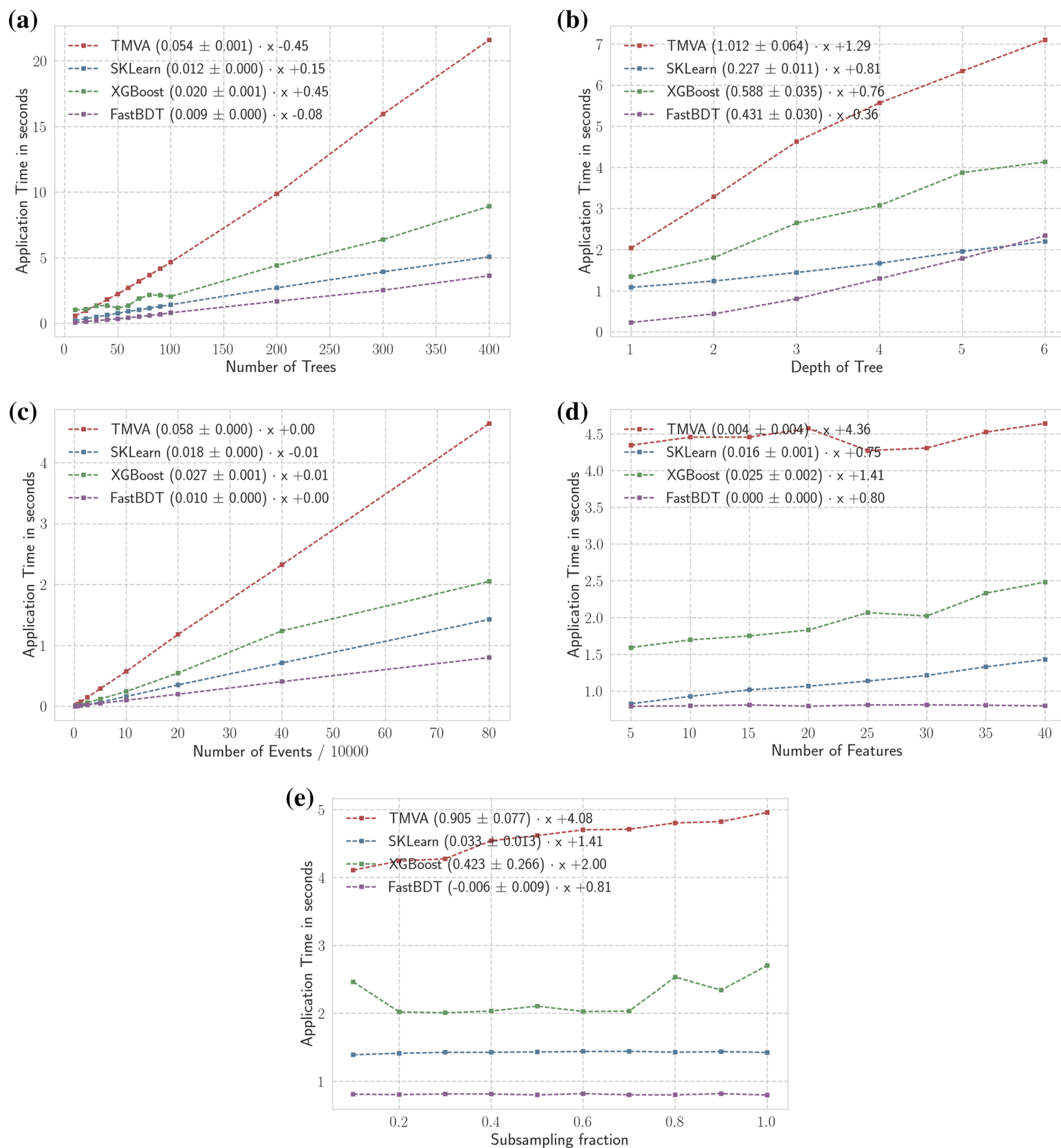


**Fig. 6** Wall-clock time for the application-phase as a function of different hyper-parameters. The $y$ axes show the application time in seconds. The $x$ axes show the varied hyper-parameter. Each configuration was measured five times: the measured minimal values are shown as *boxes* and the scaling is indicated by the *dotted line* in between the measured points. A linear regression $y = a \cdot x + c$ was performed using the minimal values of each configuration. **a** number of trees, **b** depth of the trees, **c** number of test data-points, **d** number of features, **e** sampling rate

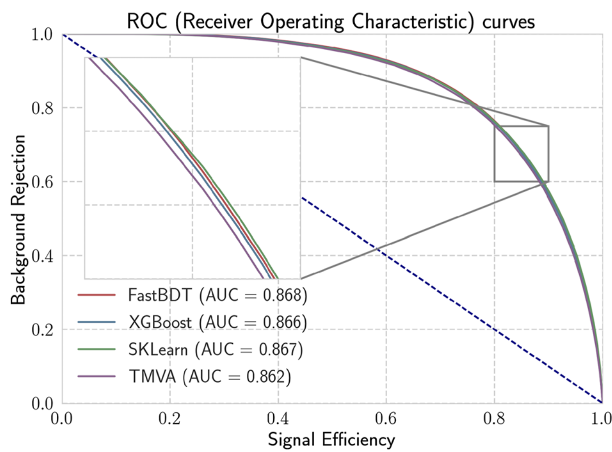


**Fig. 7** Receiver operating characteristic curves for different implementations of stochastic gradient-boosted decision trees

### Application-Phase

During the application-phase the runtime should scale linearly in `depth of the trees`, `number of trees` and `number of test data-points`. The runtime should be independent of the `sampling rate` and `number of features`. The results are summarized in Fig. 6.

The runtime behaviour in the `number of trees` and `number of test data-points` is as expected. FastBDT violates the expected linear scaling for `depth of the trees` $> 4$. In this region, the pre-fetching of the necessary cuts from the memory does not work optimally, i.e. the larger the tree the harder it is to predict the memory address which has to be accessed next.

XGBoost violates the expected linear scaling for `depth of trees` and `sampling rate`. This can be explained by its regularisation method, which takes into account the tree structure and available statistics.

TMVA seems to be faster compared to its average runtime for small values of the `sampling rate`, this can be explained by the limit on the minimum number of data-points per node. TMVA includes the possibility to write out the complete trained BDT into standalone C++ code. This can improve the application speed, however this option was not investigated in this work.

Overall, FastBDT outperforms all other implementations during the application phase.

### Classification Quality

The quality of the classification was evaluated using the area under the receiver operating characteristic curve (AUC ROC). This quality indicator is independent of the chosen working-point (i.e., the desired signal efficiency or background rejection) and is widely used to compare the separation power of different algorithms to one another. The higher the value the better the separation between signal and background.

The classification quality depends strongly on the concrete classification problem at hand. Figure 7 shows the ROC curves of all contestants on the benchmark dataset with the default configuration. Figure 8 shows the AUC ROC values for different hyper-parameter configurations.

The main difference between the implementation arises due to the different regularisation methods. FastBDT uses the equal-frequency binning to prevent over-fitted cuts; XGBoost employs a modified separation gain, which includes the structure of the current tree; and TMVA limits the minimum number of data-points per event to 5%.

FastBDT performance equally well or better compared to other implementations in most situations, except for extreme cases: shallow models with `depth of the trees` $= 1$ or `number of trees` $< 50$, where the exact cut-search of SKLearn yields better results; and complex models `depth of the trees` $>= 7$ or `number of trees` $>= 400$, where the regularisation method of XGBoost is superior.

The performance of FastBDT, SKLearn and XGBoost is nearly independent of the used sampling rate, whereas TMVA depends strongly on it. Hence it is likely that TMVA over-fits in regions with small sampling rate (and therefore low statistic).

Overall, FastBDT, SKLearn and XGBoost yield similar AUC ROC values over a wide range of hyper-parameters.

## Conclusion

FastBDT implements the widely employed Stochastic Gradient-Boosted Decision Tree algorithm [8], which exhibits a good out-of-the-box performance and generates an interpretable model. Often, multivariate classifiers are trained once and are applied on big data-sets afterwards. FastBDT performs well in this use-case and offers support for missing values, an equal-frequency preprocessing of the features, purity transformation, boosting to uniformity, and a fast application-phase. Another advantage is the fast (in terms of CPU time) fitting-phase. Possible use-cases are: real-time learning applications; frequent re-fitting of classifiers; fitting a large number of classifiers; boosting to flatness [22] and measuring the dependence of many features to one another.

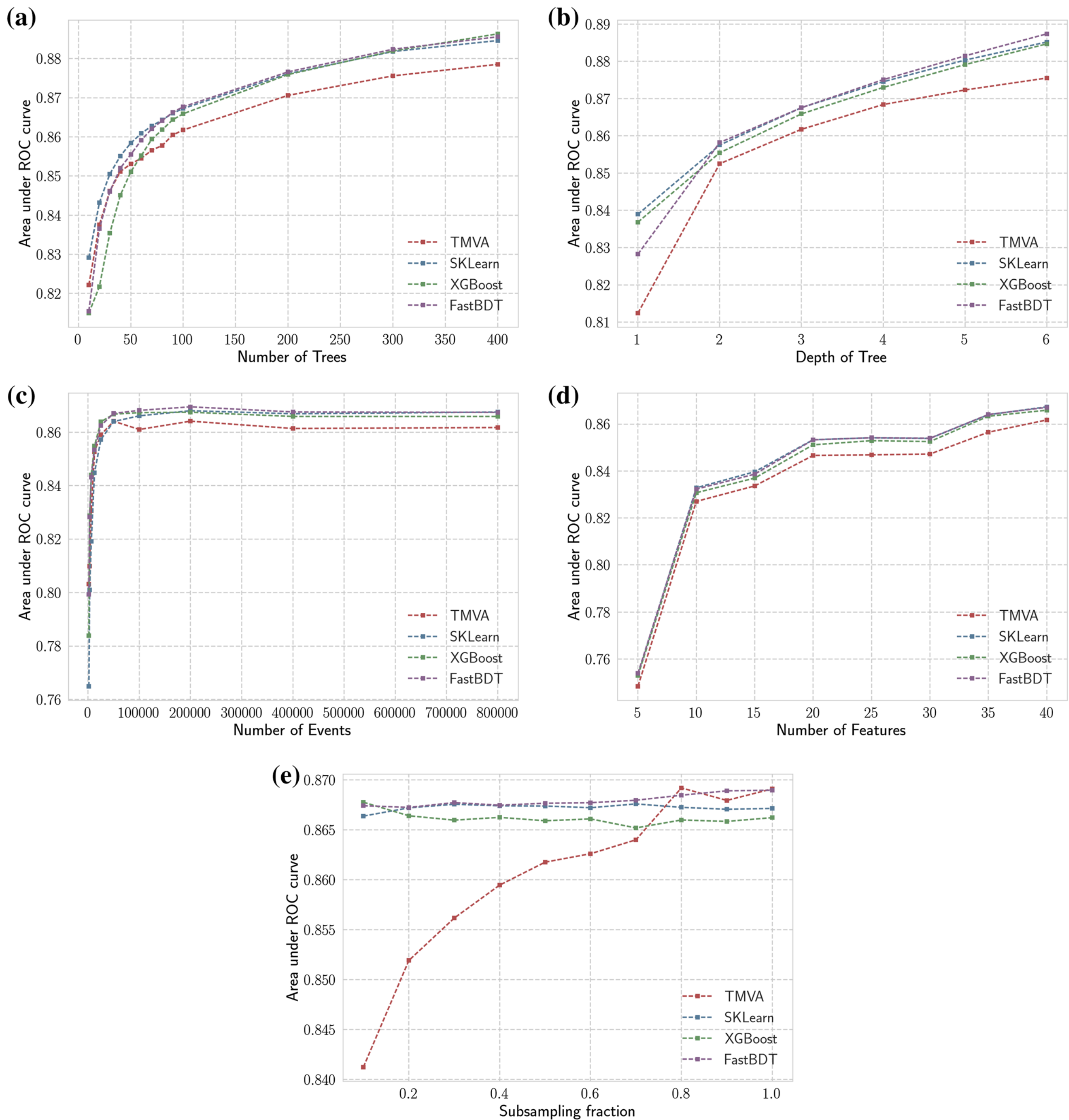


**Fig. 8** Behaviour of the classification quality with different hyper-parameter configurations. The *y* axes show the area under the receiver operating characteristic (ROC) curve. The *x* axes show the varied hyper-parameter. Each configuration was measured five times: The measured mean values are shown as boxes connected by dotted lines for better visibility. **a** number of trees, **b** depth of the trees, **c** number of training data-points, **d** number of features, **e** sampling rate

The techniques employed by FastBDT can be (and have been) migrated to other BDT implementations. The equal-frequency binning works especially well with boosted-decision trees.

FastBDT is licensed under the GPLv3 license and available on github https://github.com/thomaskeck/FastBDT.

**Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

## References


1. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X (2016) Tensorflow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf
2. Abe T et al (2010) Belle ii technical design report. *arXiv*. https://arxiv.org/abs/1011.0352
3. Appel R, Fuchs T, Dollar P, Perona P (2013) Quickly boosting decision trees—pruning underachieving features early. JMLR Workshop and Conference Proceedings, vol. 28, pp 594–602. http://jmlr.org/proceedings/papers/v28/appel13.pdf
4. Breiman L, Friedman JH, Olshen RA, Stone CJ (1984) Classification and regression trees. Statistics/probability series
5. Chen T, Guestrin C (2016) Xgboost: a scalable tree boosting system. https://arxiv.org/abs/1603.02754
6. de Melo AC (2010) The new linux 'perf' tools. kernel.org. http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf
7. Friedman JH (2000) Greedy function approximation: a gradient boosting machine. Ann Stat 29:1189–1232
8. Friedman JH (2002) Stochastic gradient boosting. Computational Statistics and Data Analysis 38(4):367–378 (Nonlinear Methods and Data Mining). doi:10.1016/S0167-9473(01)00065-2. http://www.sciencedirect.com/science/article/pii/S0167947301000652
9. Hoecker A, Speckmayer P, Stelzer J, Therhaag J, von Toerne E, Voss H (2007) TMVA: Toolkit for multivariate data analysis. PoS, ACAT:040. https://arxiv.org/abs/physics/0703039
10. IEEE (1985) IEEE standard for binary floating-point arithmetic. Institute of Electrical and Electronics Engineers
11. ISO (2012) ISO/IEC 14882:2011 Information technology—Programming languages—C++. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372
12. Keck T (2016) Fastbdt: a speed-optimized and cache-friendly implementation of stochastic gradient-boosted decision trees for multivariate classification. http://arxiv.org/abs/1609.06119
13. Kowarschik M, Weiß C (2003) Algorithms for memory hierarchies: advanced lectures. Chapter: An overview of cache optimization techniques and cache-aware numerical algorithms, pp 213–232. doi:10.1007/3-540-36574-5_10. http://dx.doi.org/10.1007/3-540-36574-5_10
14. Martschei D, Feindt M, Honc S, Wagner-Kuhr J (2012) Advanced event reweighting using multivariate analysis. J Phys Conf Ser 368:012028. doi:10.1088/1742-6596/368/1/012028. http://www-ekp.physik.uni-karlsruhe.de/~jwagner/www/publications/AdvancedReweighting_MVA_ACAT2011.pdf
15. Mingers J (1989) An empirical comparison of pruning methods for decision tree induction. Mach Learn 4(2):227–243. doi:10.1023/A:1022604100933
16. Moll A (2011) The software framework of the belle ii experiment. J Phys Conf Ser 331(3). http://stacks.iop.org/1742-6596/331/i=3/a=032024
17. Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6):89–100. doi:10.1145/1273442.1250746. http://doi.acm.org/10.1145/1273442.1250746
18. OpenMP Architecture Review Board (2015) OpenMP application program interface version 4.5. http://www.openmp.org/mp-documents/openmp-4.5.pdf
19. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830
20. Zhang T, Johnson R (2014) Learning nonlinear functions using regularized greedy forest. IEEE Trans Pattern Anal Mach Intell 36(5):942–954. doi:10.1109/TPAMI.2013.159. http://arxiv.org/abs/1109.0887
21. Rogozhnikov A, Bukva A, Gligorov VV, Ustyuzhanin A, Williams M (2015) New approaches for boosting to uniformity. JINST 10(03). doi:10.1088/1748-0221/10/03/T03002. https://arxiv.org/abs/1410.4140
22. Stevens J, Williams M (2013) uboost: a boosting method for producing uniform selection efficiencies from multivariate classifiers. J Instrum 8(12). http://stacks.iop.org/1748-0221/8/i=12/a=P12013