Journal of Statistical Software

July 2025, Volume 113, Issue 3. *doi: 10.18637/jss.v113.i03*

BoXHED2.0: Scalable Boosting of Dynamic Survival Analysis

Arash Pakbin
Texas A&M University

Xiaochen Wang
Yale University

Bobak J. Mortazavi
Texas A&M University

Donald K. K. Lee
Emory University

Abstract

Modern applications of survival analysis increasingly involve time-dependent covariates.The Python package **BoXHED2.0** (Boosted eXact Hazard Estimator with Dynamic covariates) is a tree-boosted hazard estimator that is fully nonparametric, and is applicable to survival settings far more general than right-censoring, including recurring events and competing risks. **BoXHED2.0** is also scalable to the point of being on the same order of speed as parametric boosted survival models, in part because its core is written in C++ and it also supports the use of GPUs and multicore CPUs. **BoXHED2.0** is available from PyPI and also from `https://github.com/BoXHED`.

Keywords: gradient boosting, nonparametric estimation, survival analysis, hazard estimation, survivor function, competing risks, time-varying covariates, Python.

1. Introduction

Survival analysis is concerned with analyzing the time T to an event of interest, and the fundamental quantity of interest is the hazard function $\lambda(t, X(t))$. This is informally the probability of $T \in [t, t + dt)$ given that the event has not yet occurred by t. Here, $X(t) \in \mathbb{R}^p$ denotes the predictable covariate process which can vary over time. We may think of the hazard as the survival analogue to the probability density function. For the special case where $X(t) = X$ is time-static, there also exists an analogue to the cumulative distribution called the survivor function $S(t|X) = \mathbb{P}(T > t|X)$, which can be derived from the hazard via

$$S(t|X) = \exp\left(-\int_0^t \lambda(u, X)du\right).$$

In general, if $X(t)$ is time-varying, the survivor function $S(t|X)$ is undefined because we do not know the future path of $\{X(u)\}_{0<u\leq t}$. Also, if the event of interest can recur (e.g., cancer relapse), the survivor function tells us nothing about events subsequent to the first. On the other hand, the hazard is well defined in both situations and represents the realtime risk of the event (re)occurring. For example, a patient's risk of stroke at a given point in time depends on factors such as heart rate and previous stroke history. The hazard as a function of these time-varying factors quantifies the patient's realtime risk of stroke.

Furthermore, in a competing risks setting where the occurrence of one type of event precludes the occurrence of the others (e.g., employee termination vs. resignation), the fundamental quantities that are identifiable are the cause-specific hazards (Andersen, Abildstrom, and Rosthøj 2002). The cause-specific hazard for each event type can be estimated by treating the occurrence of the other types as censoring, and then applying hazard estimation procedures such as the one introduced here. This again illustrates the unifying role played by the hazard function across a variety of survival settings.

This paper presents the Python library **BoXHED2.0**, a nonparametric hazard estimator that can handle high-dimensional, time-dependent covariates. It is a novel tree-based implementation of the gradient-boosted estimator proposed in (Lee, Chen, and Ishwaran 2021) for generic learners. The previous version **BoXHED1.0** (Wang, Pakbin, Mortazavi, Zhao, and Lee 2020) can only deal with right-censored data. **BoXHED2.0** is redesigned using a counting process framework to accommodate more general censoring schemes like those described above. It is also far more scalable than **BoXHED1.0** because its core engine is implemented in C++ and also because of a novel data preprocessing step. **BoXHED2.0** also provides built-in functionality for obtaining the survivor curve $S(t|X)$ from the estimated hazard function when the covariates are time-static.

The rest of the paper is organized as follows. Section 2 reviews related survival boosting packages. Section 3 describes the **BoXHED2.0** implementation. Section 4 introduces the **BoXHED2.0** software and its Python interface. The scalability of **BoXHED2.0** is studied in Section 5. Section 6 provides an example of a real life situation where the generality of the censoring schemes supported by **BoXHED2.0** is needed.

In Python, **BoXHED2.0** is available via **pip** and also via pre-compiled packages. Installation instructions and a tutorial for how to run **BoXHED2.0** can be found at `http://github.com/BoXHED`.

2. Related packages

There are a number of survival boosting packages for the case of time-static covariates. The R package **mboost** can be used to fit boosted parametric accelerated failure time (AFT) models (Schmid and Hothorn 2008; Hothorn, Bühlmann, Kneib, Schmid, and Hofner 2010). Boosted semiparametric Cox models can also be fit with **mboost** and also with the **gbm** package (Ridgeway 1999) in R. In Python this can be achieved with the **XGBoost** package (Chen and Guestrin 2016). For further flexibility, the R package **tbm** provides a boosting procedure for flexible transformation models of parametric families (Hothorn and Zeileis 2021; Hothorn 2020).

Machine learning work on the general time-dependent survival setting is much more recent. To our knowledge **BoXHED2.0** is the only nonparametric boosting package that extends beyond right-censored survival data. The closest package is **BoXHED1.0**, which is based on a special case of the **BoXHED2.0** implementation that deals only with right-censored data. **BoXHED1.0** is entirely implemented in Python, whereas the core engine of **BoXHED2.0** is written in C++ and supports multicore CPU and GPU training. **BoXHED2.0** employs a novel data preprocessing step that removes the need for explicit integral evaluations, which is a major bottleneck in **BoXHED1.0**. These innovations make **BoXHED2.0** orders of magnitude more scalable than **BoXHED1.0**.

3. Implementation details

3.1. Overview of survival setting

BoXHED2.0 adopts the Aalen intensity model (Aalen 1978) which encompasses a vast range of survival settings beyond right-censoring. Under the setting considered, the probability of an event occurring in $[t, t+dt)$ is

$$\lambda(t, X(t))Y(t)dt, \tag{1}$$

where $Y(t) \in \{0, 1\}$ is a predictable process indicating whether the subject is at-risk of experiencing an event during $[t, t+dt)$. Further, let $N(t)$ be the cumulative number of events that has occurred by t, and $N(t-)$ the count that excludes any potential event occurring exactly at time t. Some special cases of the setting (1) include:

- Right-censored and non-recurring events: $Y(t) = I(t \leq C, N(t-) < 1)$. Here, C is the right-censoring time.
  - Cause-specific hazards in the competing risks setting can be estimated as a further special case of this (Andersen *et al.* 2002).
- Recurring events (up to $N_{\max}$ times): $Y(t) = I(N(t-) < N_{\max})$.
- Left-truncation and right-censoring: The likelihood for this coincides with the one obtained from setting $Y(t) = I(L \leq t \leq C, N(t-) < 1)$, where L is the left-truncation time. Hence, the same computational procedure applies.
- Cancer relapse: $Y(t) = 1$ whenever a patient is in remission and hence is at risk of relapse, and $Y(t) = 0$ if the patient is in relapse.

The event history of a subject up to time τ is captured by the functional data point $\{X(t), Y(t), N(t)\}_{t \leq \tau}$. For recurring events, $X(t)$ might include variables like time since the last event and/or the number of past events $N(t-)$.[1]

Under (1), the process generating the first event time can be described as follows: Conditional on the first event having not occurred by t, the probability that it happens in $[t, t+dt)$ equals $\lambda(t, X(t))Y(t)dt$. Thus the likelihood of observing the first event at $T_{(1)}$ follows a sequence of

[1] The choice of $N(t-)$ rather than $N(t)$ ensures that $X(t)$ remains a predictable process.

coin flips at $t = 0, dt, 2dt, \ldots$, i.e.,

$$\begin{aligned}
&\{1 - \lambda(0, X(0))Y(0)dt\} \times \{1 - \lambda(dt, X(dt))Y(dt)dt\} \times \ldots \times \lambda(T_{(1)}, X(T_{(1)})) \\
&\xrightarrow[dt\downarrow 0]{} \exp\left\{-\int_0^{T_{(1)}} Y(t)\lambda(t, X(t))dt\right\} \lambda(T_{(1)}, X(T_{(1)})) \\
&= \exp\left\{-\int_0^{T_{(1)}} Y(t)\lambda(t, X(t))dt + \log \lambda(T_{(1)}, X(T_{(1)}))\right\},
\end{aligned}$$

where the limit can be understood as a product integral. Continuing this line of argument for potential subsequent events leads to the following likelihood for all observed events $T_{(1)}, T_{(2)}, \ldots$ up to time τ:

$$\begin{aligned}
&\exp\left\{-\int_0^{\tau} Y(t)\lambda(t, X(t))dt + \sum_k \log \lambda(T_{(k)}, X(T_{(k)}))\right\} \\
&= \exp\left\{-\int Y(t)\lambda(t, X(t))dt + \int \{\log \lambda(t, X(t))\} dN(t)\right\},
\end{aligned}$$

where we drop mention of the integral ranges since we can set $Y(t) = 0$ and $dN(t) = 0$ for $t > \tau$. Note that the likelihood also captures the case where censoring is present. For example, if the subject was censored at time C before any events have occurred, we have $Y(t) = I(t \leq C)$ and $N(t)_{t \leq C} = 0$. The likelihood then reduces to the more familiar $\exp\left\{-\int_0^C \lambda(t, X(t))dt\right\}$. Thus letting $F(t, x) := \log \lambda(t, x)$ be the log-hazard function and given n functional data samples

$$\{X_i(t), Y_i(t), N_i(t)\}_t, \tag{2}$$

we define the *likelihood risk* as the negative log-likelihood

$$R_n(F) = \sum_{i=1}^{n} \left\{\int Y_i(t) e^{F(t, X_i(t))} dt - \int F(t, X_i(t)) dN_i(t)\right\}. \tag{3}$$

By the likelihood principle, a function $\hat{F}$ that minimizes $R(F)$ provides a candidate for the log-hazard estimator, with $\hat{\lambda} = e^{\hat{F}}$ being the corresponding hazard estimator. The consistency for such a hazard estimator obtained from gradient boosting is formally established in Lee *et al.* (2021) for arbitrary learners.[2] The **BoXHED2.0** estimator is a novel implementation that employs tree learners in particular.

3.2. BoXHED2.0

We aim to construct a tree ensemble for the log-hazard estimator

$$F_M(t, x) = F_0 + \nu \sum_{m=0}^{M-1} g_m(t, x)$$

that iteratively reduces (3), i.e., the boosted nonparametric maximum likelihood estimator (MLE). Here, $g_0(t, x), \ldots, g_{M-1}(t, x)$ are tree learners of limited depth. The initial guess

[2] Under mild identifiability conditions the estimator converges in probability to the tree ensemble that best approximates the true hazard.

$F_0 = \log \frac{\sum_i N_i(\infty)}{\sum_i \int Y_i(t)dt}$ is the best minimizing constant for (3), M is the number of boosting iterations, and $\nu \ll 1$ is the learning rate. The **BoXHED2.0** estimator is given by

$$\hat{\lambda}(t,x) = e^{F_M(t,x)}. \tag{4}$$

In traditional gradient boosting (Friedman 2001), at the m-th iteration the tree g_m is constructed to approximate the negative gradient function of the risk at $F = F_m$, the direction of steepest descent. More recent implementations of boosting such as **XGBoost** constructs g_m in a more targeted manner, growing it to directly minimize the second order Taylor approximation to the risk. **BoXHED2.0** takes this one step further by growing trees to directly minimize the exact form of (3), resulting in even more targeted risk reduction:[3] Starting with a tree learner $g_{m,0}$ of depth zero (the root node being the only leaf node), we split each leaf node to maximally reduce $R_n(F)$ and repeat the process. Thus the intermediate tree of depth l is[4]

$$g_{m,l}(t,x) = \sum_{\ell=1}^{2^l} c_{m,\ell} I_{B_{m,\ell}}(t,x),$$

where $B_{m,\ell}$ represents the time-covariate region for the ℓ-th leaf node of the form

$$B_\ell = \left\{ (t,x) : \begin{pmatrix} t^{(\ell_0)} < t \le t^{(\ell_0+1)} \\ x^{(1,\ell_1)} < x^{(1)} \le x^{(1,\ell_1+1)} \\ \vdots \\ x^{(p,\ell_p)} < x^{(p)} \le x^{(p,\ell_p+1)} \end{pmatrix} \right\}, \tag{5}$$

and $c_{m,\ell}$ is the value of the tree function in that region. Here, $x^{(k)}$ denotes the k-th covariate. To obtain $g_{m,l+1}(t,x)$ from $g_{m,l}(t,x)$, we split each leaf region $B_{m,\ell}$ in $g_{m,l}$ into left and right daughter regions A_L and A_R by either splitting on time t or on one of the covariates $x^{(1)}, \ldots, x^{(p)}$. Since the leaf node regions are disjoint, the restriction of $g_{m,l+1}$ to $(t,x) \in B_{m,\ell}$ is

$$g_{m,l+1}(t,x)|_{B_{m,\ell}} = \gamma_L I_{A_L}(t,x) + \gamma_R I_{A_R}(t,x).$$

The variable or time axis to split on, the location of the split, and also the values of (γ_L, γ_R) are all chosen to directly minimize $R_n(F_m + g_{m,l+1})$. Since the values γ_L, γ_R only apply to the subregions A_L, A_R that are inside $B_{m,\ell}$, $R(F_m + g_{m,l+1})$ equals

$$\begin{aligned} &\sum_{i=1}^n \sum_{k=L,R} \left\{ \int_{(t,X_i(t)) \in A_k} Y_i(t) e^{F_m(t,X_i(t))+\gamma_k} dt - \int_{(t,X_i(t)) \in A_k} \left[F_m(t,X_i(t)) + \gamma_k\right] dN_i(t) \right\} + C \\ =&\sum_{i=1}^n \sum_{k=L,R} \left\{ e^{\gamma_k} \int_{(t,X_i(t)) \in A_k} Y_i(t) e^{F_m(t,X_i(t))} dt - \gamma_k \int_{(t,X_i(t)) \in A_k} dN_i(t) \right\} + C' \\ =& \sum_{k=L,R} (e^{\gamma_k} U_k - \gamma_k V_k) + C'. \end{aligned}$$

[3] If $R_n(F)$ can be reduced by moving in the direction of a tree learner g, then g is necessarily aligned to the negative gradient function of $R_n(F)$ because the risk is convex (Lee *et al.* 2021). As shown in Lee *et al.* (2021), alignment is needed for consistency, and directly fitting the learner to the negative gradient is just one of many possible ways to achieve this.

[4] Similar to **XGBoost**, to obtain a tree of depth l, our implementation splits every node above it to obtain 2^l terminal nodes.

where C, C' do not depend on γ_L or γ_R, and

$$U_k = \sum_{i=1}^{n} \int_{(t,X_i(t))\in A_k} Y_i(t) e^{F_m(t,X_i(t))} dt, \qquad V_k = \sum_{i=1}^{n} \int_{(t,X_i(t))\in A_k} dN_i(t). \tag{6}$$

Hence $R_n(F_m + g_{m,l+1})$ is minimized by $\gamma_k = \log(V_k/U_k)$. Adopting the convention that $0 \times \infty = 0$, the drop in risk $R_n(F_m + g_{m,l}) - R_n(F_m + g_{m,l+1})$ from splitting $B_{m,\ell}$ into (A_L, A_R) is

$$\Pi = (V_L + V_R)\log\left(\frac{U_L + U_R}{V_L + V_R}\right) - V_L \log\left(\frac{U_L}{V_L}\right) - V_R \log\left(\frac{U_R}{V_R}\right). \tag{7}$$

Therefore the best split (A_L, A_R) of $B_{m,\ell}$ is that which maximizes Π. Since Π does not depend on the other disjoint leaf regions, all 2^l leaf nodes of $g_{m,l}$ can be split in parallel.

The quantities (6) and (7) are calculated for every possible split in order to determine the best split for each $B_{m,\ell}$. Note that V_k is the total number of events observed in the region A_k for the n subjects, and may include more than one event per subject due to the possibility of recurring events.

3.3. Speedup via data preprocessing

Three factors contribute to **BoXHED2.0**'s massive scalability. The first is the use of C++ for the core calculations, based in part on the **XGBoost** codebase (Chen and Guestrin 2016), a highly efficient boosting package for non-functional data. The second is explicit parallelization via multicore CPUs and GPUs. The third is a novel data preprocessing step: Recall that the integral U_k in (6) must be calculated for every possible split in order to identify the one that maximizes the score (7). The preprocessing step in **BoXHED2.0** transforms the functional survival data (2) in such a way that the required numerical integration comes for free as part of training. Details can be found in the appendix. It is worth noting that a training set only needs to be preprocessed once, rather than for each time the **BoXHED2.0** estimator is fit with a particular set of hyperparameters during the tuning step.

3.4. Split search using multicore CPUs and GPUs

For a continuous covariate, traditional boosting implementations typically place candidate split points at every observed covariate value. This takes $\mathcal{O}(n)$ trials to search through all possible splits for one covariate. On the other hand, picking a pre-specified set of quantiles (e.g., every percentile) of the observed values as candidate splits reduces the search time to $\mathcal{O}(1)$. The default option in **BoXHED2.0** is to use 256 quantiles as candidate split points for time and for each covariate.[5] Users may choose to use a different number, or even supply their own candidate split points, but in either case the number of candidate splits may not exceed 256. **BoXHED2.0** offers two flavours of quantiles: Raw and time-weighted.

Raw quantiles. The set of unique values for time and for each covariate are collected, and the quantiles are obtained from this.

[5] For GPU training of **BoXHED2.0** models, the resource bottleneck is typically GPU memory. Using 256 candidate splits allows the covariate values to be stored as a byte rather than as a double (Lou, Caruana, Gehrke, and Hooker 2013).

Weighted quantiles. In **XGBoost** the risk function is approximated by its second order Taylor expansion, and the Hessian is used as the weight for quantile sketch. For the time-dependent survival setting we propose a much more natural weight, i.e., time. To illustrate, imagine a sample with one subject ($n = 1$) whose covariate value is $x = 1.3$ for $t \in (0, 2]$ and $x = 2$ for $t \in (2, 3]$. Under the raw quantile setting, $x = 1.3$ and $x = 2$ are each given a weight of $1/2$. However, since twice as much time was spent at $x = 1.3$ than at $x = 2$, in a weighted setting we give $x = 1.3$ a weight of $2/3$, and $1/3$ for $x = 2$.

3.5. Missing covariate values

In practice, it is possible for some of the covariates in the data to be missing. If the k-th covariate $x^{(k)}$ in question is categorical, then ‘missingness’ can be treated as an additional factor level. Otherwise, **BoXHED2.0** implements left and right splits of the form

$$\left\{\left\{x^{(k)} \le \chi \text{ or } x^{(k)} \text{ missing}\right\}, \left\{x^{(k)} > \chi\right\}\right\} \quad \text{or} \quad \left\{\left\{x^{(k)} \le \chi\right\}, \left\{x^{(k)} > \chi \text{ or } x^{(k)} \text{ missing}\right\}\right\}.$$

3.6. Variable importance

The variable importance measure for the k-th variable (the zeroth one being time t) is

$$\mathcal{I}_k = \sum_{m=0}^{M-1} \mathcal{I}_k(g_m) = \sum_{m=0}^{M-1} \sum_{\ell=1}^{L} \Pi_{m,\ell} I(v(m, \ell) = k) \ge 0, \tag{8}$$

where for tree g_m with L internal nodes, $\Pi_{m,\ell}$ is the split score (7) at the ℓ-th internal node, and $v(m, \ell)$ is the variable used for the partition. Hence the inner sum on the right represents the total reduction in likelihood risk due to splits on the k-th variable in the m-th tree, and $\mathcal{I}_k$ is the total reduction across the M trees. In other words, the variable importance quantifies a variable’s contribution to reducing the likelihood risk. This is more natural than the traditional variable importance measure in (Friedman 2001), which is defined as the reduction in mean squared error between the tree learners and the gradients of the risk at each boosting iteration. To convert $\mathcal{I}_k$ into a measure of relative importance between 0 and 1, it is scaled by $\max_k \mathcal{I}_k$, where a larger value confers higher importance.

4. Using BoXHED2.0

This section employs a synthetic dataset to walk readers through the use of **BoXHED2.0**. A detailed Jupyter notebook tutorial called `BoXHED2_tutorial.ipynb` is also provided on the GitHub page for **BoXHED2.0**.

4.1. Structure of training data

Input data on the event histories of study subjects are provided to **BoXHED2.0** as a **pandas** dataframe (The **pandas** Development Team 2020). The **pandas** dataframe follows the same convention as a Cox analysis of time-dependent covariates:

- `ID`: Subject ID.
- `t_start`: The start time of an epoch for the subject.
- `t_end`: The end time of the epoch.

- X$_i$: Value of the i-th covariate between t_start and t_end.
- delta: Event label, which is 1 if an event occurred at t_end; 0 otherwise.

An illustrative example of the entries of the dataframe is:

```
   ID  t_start   t_end       X0  ...     X10  delta
0   1   0.0100  0.0747   0.2655       0.2059      0
1   1   0.0747  0.1072   0.7829       0.4380      0
2   1   0.1072  0.1526   0.7570       0.7789      0
```

Each row of the dataframe corresponds to an *epoch* of a subject's history. The start and end times of an epoch are given by t_start and t_end, and the values of the subject's covariates (X0 to X10 in this example) are constant inside an epoch. For each row we must have $0 \leq$ t_start $<$ t_end. Also, epochs cannot overlap. In other words, the beginning of an epoch cannot start earlier than the end of the prior epoch. Any column whose name is not in the set {ID, t_start, t_end, delta} is treated as a covariate.

4.2. Importing BoXHED and defining an instance

From the Python package **BoXHED** the class boxhed can be imported

```
>>> from boxhed.boxhed import boxhed
```

and from the imported class, an instance can be defined:

```
>>> boxhed_ = boxhed()
```

4.3. Data preprocessing

As explained in Section 3.3, **BoXHED2.0** preprocesses the training data to speed up training. This step only needs to be run once per training set. **BoXHED2.0** models are fit directly to the preprocessed data.

The user may specify the number of candidate split points (≤ 256) for time and for each non-categorical covariate (argument "num_quantiles"). The locations of such splits are based on the marginal quantiles of the training data. Alternatively, the user may specify no more than 256 custom candidate split points for time and/or a subset of non-categorical covariates (argument "split_vals"). For example, if the third line in the code below is uncommented, candidate splits would be limited to four locations on time and three locations on the X_2 variable, while the other non-categorical variables will each be endowed with 256 candidate split points:

```
>>> X_post = boxhed_.preprocess(data = train_data,
...   #split_vals = {"t": [0.2, 0.4, 0.6, 0.8], "X_2": [0, 0.4, 0.9]},
...   num_quantiles = 256, nthread = 20)
```

The preprocessor returns a dictionary which contains the preprocessed data, which is used for training and hyperparameter tuning.

4.4. Hyperparameter tuning

BoXHED2.0 enables both manual selection of hyperparameters and also hyperparameter tuning through K-fold cross-validation. The primary **BoXHED** hyperparameters that need to be tuned are:

- `max_depth` the maximum depth of each boosted tree.
- `n_estimators` the number of trees.

The hyperparameter grid provided to cross validation is specified as a dictionary:

```
>>> param_grid = {
...   'max_depth':    [1, 2, 3, 4, 5],
...   'n_estimators': [50, 100, 150, 200, 250, 300]
... }
```

This hyperparameter grid amounts to trying trees of depth up to 5 (2^5 leaf nodes) and up to 300 trees. The folds are split by subject ID, so that all of the subject's epochs belong to either the training split or the validation split. To perform K-fold cross-validation we first import the function `cv()`:

```
>>> from boxhed.model_selection import cv
```

The `cv()` function may be called on the preprocessed data. The number K in K-fold cross-validation is set as the variable `num_folds`. The user can specify whether to use GPUs or CPUs in the `cv()` function. Here we use CPUs by setting `gpu_list =[-1]`. The parameter `nthread` is set to 1 by default, while setting it to -1 corresponds to using all CPU threads. For instructions on how to use GPU, refer to the tutorial on GitHub.

The user can specify the argument `seed` to fix the seed of the random number generator used to produce the cross validation splits. Here, we choose a value of 6 for replication purposes:

```
>>> cv_rslts = cv(param_grid, X_post, num_folds = 5, seed = 6, ID = ID,
... gpu_list = [-1], nthread = -1)
```

The function `cv()` returns a dictionary containing the possible hyperparameter combinations along with their K-fold means and standard errors of log-likelihood values. They are all in form of **NumPy** vectors (Harris *et al.* 2020). The means can be inspected as follows:

```
>>> import numpy as np
>>> nrow = len(param_grid['max_depth'])
>>> ncol = len(param_grid['n_estimators'])
>>> print(np.around(cv_rslts['score_mean'].reshape(nrow, ncol), 2))
```

```
    [[ -518.84 -500.75 -496.83 -495.81 -495.93 -495.97]
     [ -499.32 -498.62 -500.91 -502.83 -505.1  -507.48]
     [ -500.69 -508.07 -513.77 -520.41 -526.88 -533.33]
     [ -507.09 -518.29 -531.18 -545.01 -555.32 -569.18]
     [ -516.2  -533.55 -555.09 -572.67 -593.68 -614.24]]
```

For standard errors:

```
>>> print(np.around(cv_rslts['score_ste'].reshape(nrow, ncol), 2))
```

```
[[ 5.72 5.05 4.86 4.72 4.77 4.73]
 [ 4.7  4.91 4.99 5.24 5.49 5.38]
 [ 4.73 5.   5.04 4.91 4.73 4.5 ]
 [ 5.73 6.15 5.92 6.03 5.69 6.48]
 [ 6.37 5.89 6.06 7.07 7.   7.51]]
```

The numbers above can be expressed in the tabular form (standard errors in parentheses):

```
n_estimators     50             100             150             200             250             300
max_depth
1     -518.84(5.72)   -500.75(5.05)   -496.83(4.86)   -495.81(4.72)   -495.93(4.77)   -495.97(4.73)
2     -499.32(4.70)   -498.62(4.91)   -500.91(4.99)   -502.83(5.24)   -505.10(5.49)   -507.48(5.38)
3     -500.69(4.73)   -508.07(5.00)   -513.77(5.04)   -520.41(4.91)   -526.88(4.73)   -533.33(4.50)
4     -507.09(5.73)   -518.29(6.15)   -531.18(5.92)   -545.01(6.03)   -555.32(5.69)   -569.18(6.48)
5     -516.20(6.37)   -533.55(5.89)   -555.09(6.06)   -572.67(7.07)   -593.68(7.00)   -614.24(7.51)
```

The mean log-likelihood is maximized at {`max_depth=1`, `n_estimators=200`} with value -495.81. However, the *one-standard-error rule* (Chapter 7.10 in Hastie, Tibshirani, Friedman, and Friedman 2009) suggests choosing the 'simplest model' whose mean log-likelihood is no less than 1 standard error of the maximum. In this example, this means choosing the simplest model whose mean log-likelihood is no less than $-495.81 - 4.72 = -500.53$. The hyperparameters with mean log-likelihood larger than this are:

```
n_estimators     50              100             150             200             250             300
max_depth
1                                           -496.83         -495.81         -495.93         -495.97
2           -499.32         -498.62
3
4
5
```

There is no generally accepted way to compare the complexities of these choices against one another. One heuristic is to define the complexity of a hyperparameter combination as

$$\log_2\left(\texttt{n\_estimators} \times 2^{\texttt{max\_depth}}\right) = \log_2(\texttt{n\_estimators}) + \texttt{max\_depth},$$

which is $\log_2$(total number of leaf nodes in tree ensemble). Under this criterion, {`max_depth=2`, `n_estimators=50`} has the lowest complexity, and is hence the most parsimonious choice. However, if we only consider hyperparameters with `max_depth` ≤ 1 and `n_estimators` ≤ 200, then {`max_depth=1`, `n_estimators=150`} is the most parsimonious. This restriction ensures that the chosen combination is no less parsimonious than the log-likelihood maximizer under any sound definition of complexity.

The function `best_param_1se_rule()` automates the described process of finding the most parsimonious combination. It first needs to be imported:

```
>>> from boxhed.model_selection import best_param_1se_rule
```

The user needs to then supply a model complexity measure:

```
>>> def model_complexity(max_depth, n_estimators):
...     from math import log2
...     return log2(n_estimators) + max_depth
```

And finally run the `best_param_1se_rule()` function:

```
>>> best_params, _ = best_param_1se_rule(cv_rslts, model_complexity,
...   bounded_search = True)
```

Setting `bounded_search = True` forces the search to only consider hyperparameter combinations satisfying `max_depth` ≤ 1 and `n_estimators` ≤ 200 (the hyperparameter combination maximizing the log-likelihood).

Having found the best hyperparameter combination, the **BoXHED** instance can be set to this hyperparameter combination:

```
>>> boxhed_.set_params(**best_params)
```

Alternatively, the user can manually supply the hyperparameter combination as follows:

```
>>> boxhed_.set_params(**{'max_depth':1, 'n_estimators':150})
```

4.5. Fitting BoXHED

BoXHED can be fit using multicore CPUs or GPUs. GPUs are normally accessed through an integer identifier which is in the range {0, 1, . . . , # GPUs -1}. To use a GPU, you may specify its number. For example, to use the first GPU:

```
>>> boxhed_.set_params(**{'gpu_id'=0})
```

The default value for the parameter `gpu_id` is -1 which corresponds to using CPUs only. When using CPUs, the number of CPU threads can be set by:

```
>>> boxhed_.set_params(**{'nthread'=20})
```

Following the same convention as the function `cv()`b, setting the parameter `nthread` to -1 corresponds to using all CPU threads. Finally, to fit **BoXHED**, execute the following to pass the preprocessed data `X_post` to the `fit()` function:

```
>>> boxhed_.fit(X_post['X'], X_post['delta'], X_post['w'])
```

BoXHED stores the variable importance measures, as defined in Section 3.6, in a class variable `VarImps`. It is a dictionary of variables with their corresponding importances. It can be printed:

```
>>> print (boxhed_.VarImps)
```

```
{ 'X_0':  1971.2105061100006,
  'time': 1558.9728342899996,
  'X_4':  21.197394600000003,
  'X_9':  9.797206169999999,
  'X_1':  6.96407747,
  'X_10': 6.74315118,
  'X_6':  2.85510993,
  'X_7':  5.40960407 }
```

The set of time values where tree splits are made can be accessed by: %

```
>>> print (boxhed_.time_splits)
```

```
[0.02809964 0.03668491 0.04400259 0.05081875 0.07196306 0.08450372
 0.10639625 0.11365116 0.14458408 0.15542936 0.19222417 0.20236476
 0.21580724 0.23802629 0.3294313  0.36373213 0.42403865 0.69717562
 0.70240098 0.72262114 0.82380444 0.83513868 0.84786463 0.89849269
 0.93779886 0.97168624]
```

4.6. Hazard estimation

We can extract the value of the estimated hazard function $\hat{\lambda}(t, x_0, x_1, \ldots)$ evaluated at a given time t with covariate values $X(t) = (x_0, x_1, \ldots)$. For example, given the following **pandas** dataframe `test_X`:

```
     t          X_0  ...  X_10
0    0.000000   0.0       0.508183
1    0.010101   0.0       0.414983
2    0.020202   0.0       0.407774
3    0.030303   0.0       0.589770
4    0.040404   0.0       0.840509
```

We can retrieve the estimated value $\hat{\lambda}(t, x_0, \ldots, x_{10})$ for each row of `test_X` using the `hazard()` function:

```
>>> haz_vals = boxhed_.hazard(test_X)
```

Plotting the estimated hazard values as a function of time and one of the covariates, X_0, yields the following plot:

4.7. Survivor curve estimation for time-static covariates

Recall from Section 1 that the survivor curve $S(t|X) = \mathbb{P}(T > t|X)$ is not meaningful when the covariates $X(t)$ change over time. However, for time-static covariates $X(t) = X$, $S(t|X)$ can be obtained from the estimated hazard function.

As an example, suppose we want to obtain the survivor curve conditional on the covariate vector $(X_0 = 0.20202, \ldots, X_{10} = 0.128217)$. We wish to evaluate the curve at $t = 0, 0.01, 0.02, \ldots, 0.99$. We use the code below to generate the **pandas** dataframe:

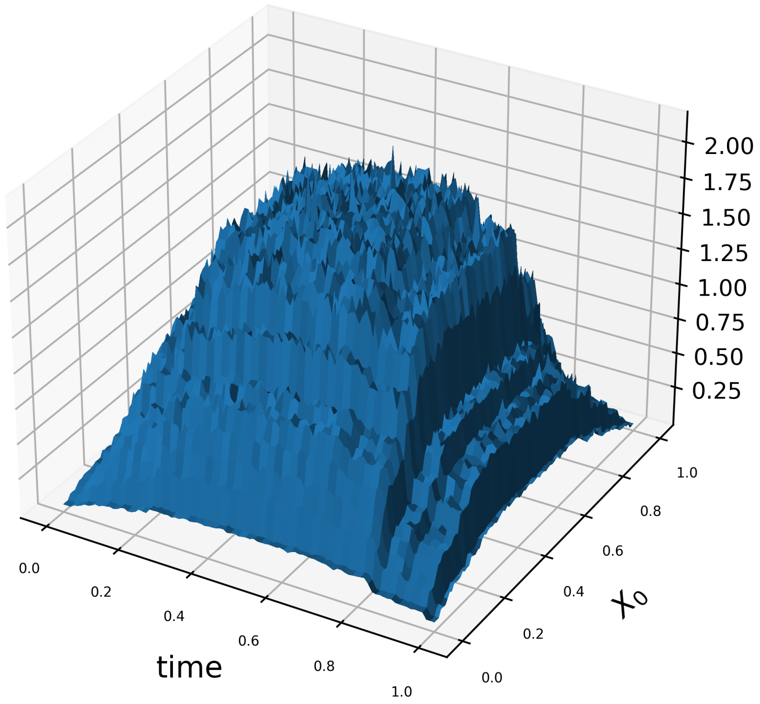

Figure 1: The estimated hazard values as a function of time and one of the covariates, X_0.

```
>>> t = [t/100 for t in range(0, 100)]
>>> df_surv = pd.concat([test_X.loc[2000].to_frame().T]*len(t))
>>> df_surv = df_surv.reset_index(drop=True)
>>> df_surv['t'] = t
```

```
        t     X_0      ...  X_10
0    0.00  0.20202         0.128217
1    0.01  0.20202         0.128217
2    0.02  0.20202         0.128217
..    ...      ...         ...
98   0.98  0.20202         0.128217
99   0.99  0.20202         0.128217
```

To estimate the value of the survivor curve for each row, run:

```
>>> surv_vals = boxhed_.survivor(df_surv)
```

4.8. Saving and loading the fitted BoXHED model

Use `dump_model()` to save the fitted **BoXHED** instance to disk, and use `load_model()` to retrieve it.

Saving a **BoXHED** model:

```
>>> boxhed_.dump_model("./boxhed.pkl")
```

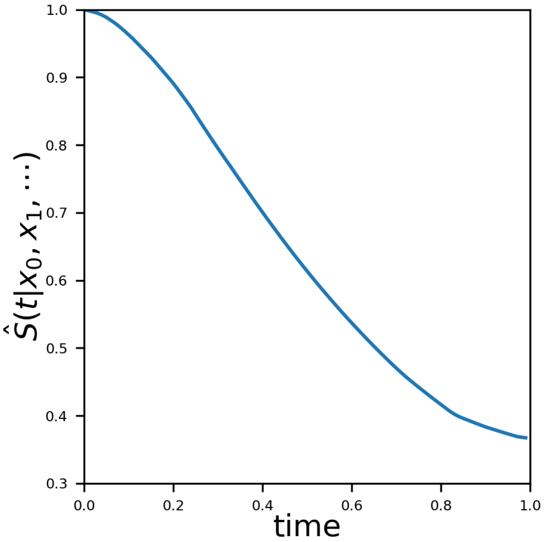

Figure 2: The estimated survivor curve as a function of time.

Retrieving a **BoXHED** model:

```
>>> boxhed_2 = boxhed()
>>> boxhed_2.load_model("./boxhed.pkl")
```

5. Scalability analysis

To benchmark how **BoXHED2.0** scales with data, we perform runtime comparisons in three settings: 1) Comparing **BoXHED2.0** (CPU and GPU) against **BoXHED1.0** as the number of rows in the dataset is increased; 2) Comparing **BoXHED2.0** against the boosted parametric survival models in Hothorn *et al.* (2010) as the number of rows in the dataset is increased; and 3) Assessing how **BoXHED2.0** scales with the number of covariates in the model.

To create a synthetic dataset of a given size, we used one of the experiments in Wang *et al.* (2020). The dataset consists of 41 covariates and up to 10 million rows, details are provided in the appendix. Computations were performed on a server with two Intel Xeon CPU E5-2650 v4 2.20GHz processors with 512 GB of RAM, and a GeForce GTX 1080 Ti GPU with 11GB of RAM. Raw quantiles were used to select 256 splits. 20 CPU threads were utilized for **BoXHED2.0** CPU.

BoXHED2.0 vs. BoXHED1.0: Scaling the number of data rows. Figure 3 depicts how **BoXHED2.0** and **BoXHED1.0** scale as the number of data rows is increased. The hyperparameters chosen for **BoXHED** are {`max_depth`=1, `n_estimators`=250}. On the other hand, the hyperparameters for **BoXHED1.0** are set as {`max_depth`=1, `n_estimators`=1} and each covariate (including time) are given 10 candidate splits. In other words, the **BoXHED1.0** runtimes measure how long it takes **BoXHED1.0** to decide on the first split from among 10

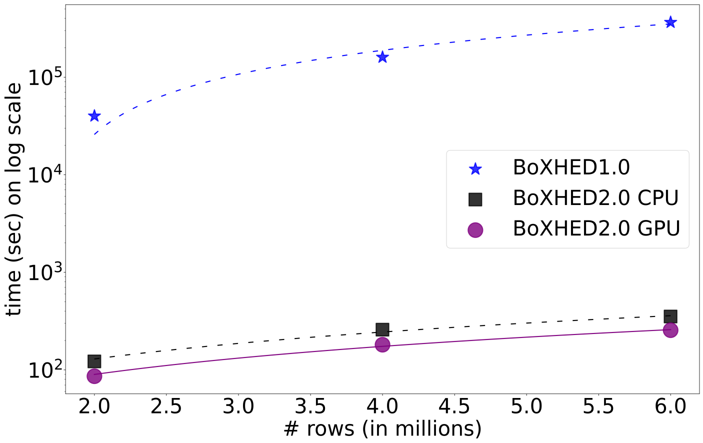

Figure 3: Runtimes of **BoXHED2.0** vs. **BoXHED1.0** as the number of data rows increase.

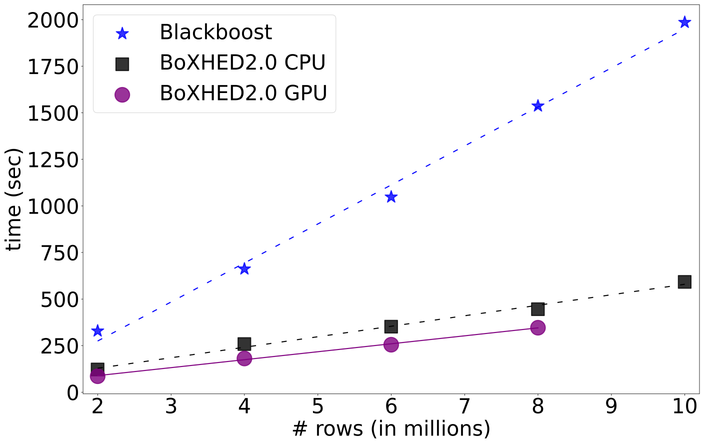

Figure 4: Scalability analysis of **BoXHED2.0** vs. the boosted parametric survival model **mboost** (log-normal family).

candidate splits rather than from 256. Even after giving **BoXHED1.0** such a substantial leg up, **BoXHED2.0** is still on average 900 times faster.

BoXHED2.0 vs. boosted parametric survival models: Scaling the number of data rows. Figure 4 compares **BoXHED2.0** run times against those for the boosted parametric survival model **mboost** in R (Hothorn *et al.* 2010). The hyperparameters chosen for all estimators are {`max_depth`=1, `n_estimators`=250}. For **BoXHED2.0** GPU, only the results for up to 8 million rows of data are included, as larger datasets did not fit into the memory of the GPU we used for testing. We see from the figure that all methods scale linearly with the number of rows. Remarkably though, the nonparametric **BoXHED2.0**'s speed exceeds that of the boosted parametric model by a large margin. This is even more remarkable given that **mboost** only applies to time-static covariates, so it does not have to contend with the extra computations required for time-dependent covariates.

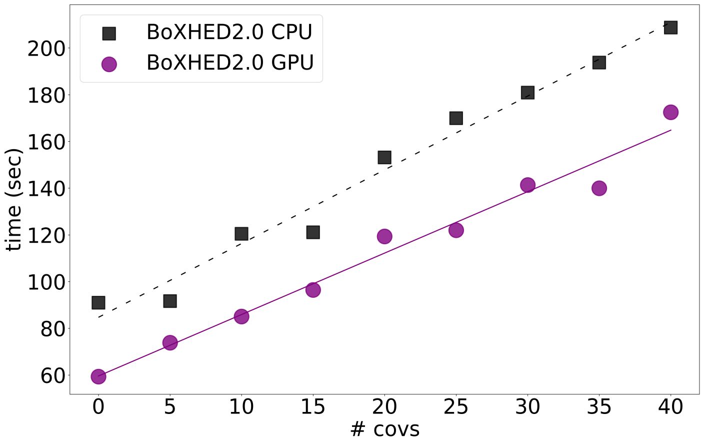

Figure 5: Scalability analysis of **BoXHED2.0** for varying number of covariates.

BoXHED2.0: Scaling the number of covariates. Figure 5 reveals how the runtime of **BoXHED2.0** scales with the number of covariates. The number of rows in the dataset is fixed at 4 million rows. The chosen hyperparameters are {`max_depth`=1, `n_estimators`=250}. We see that **BoXHED2.0** runtimes scale linearly with the number of covariates, with **BoXHED2.0** GPU being around 20% faster than **BoXHED2.0** CPU.

6. BoXHED2.0 as a realtime risk monitor

We demonstrate **BoXHED2.0** in practice by assessing the realtime risk of experiencing invasive-ventilation events in the intensive care unit (ICU). This involves survival data that go beyond the right-censoring setting prevalent in survival analysis: Over the course of an ICU admission, a patient may experience multiple episodes of invasive ventilation (iV), which is when a tube is inserted into the trachea to assist with breathing. The onset of iV thus constitutes a recurrent event. Moreover, while undergoing an episode of iV, the patient by definition cannot be at-risk of experiencing a concurrent episode of iV, so the at-risk indicator $Y(t)$ must be zero during an episode.

Patients undergoing iV have increased risk of severe adverse events and mortality, with the importance of proper and timely ventilation being punctuated during the COVID-19 pandemic (Yu *et al.* 2021). Thus having a realtime risk measure for the onset of iV may facilitate early intervention, potentially via non-invasive means. To our knowledge there is no other nonparametric machine learning solution besides **BoXHED2.0** for handling this type of survival data. We demonstrate this application on the MIMIC-IV dataset (Medical Information Mart for Intensive Care, version IV, Johnson *et al.* 2021), identifying informative covariates and providing real-time risk monitoring. For details on the data extraction process (cohort and variables) see the appendix.

After using **BoXHED2.0** to estimate the hazard rate for iV, the ten most informative covariates, along with their relative importances, are displayed in Figure 6.

As expected, the most important features represent the patient's current (and degrading) respiratory state. The estimated hazard values serve as a realtime risk indicator. As a patient's ICU stay progresses, their vitals evolve and hence so does the patient's risk. An

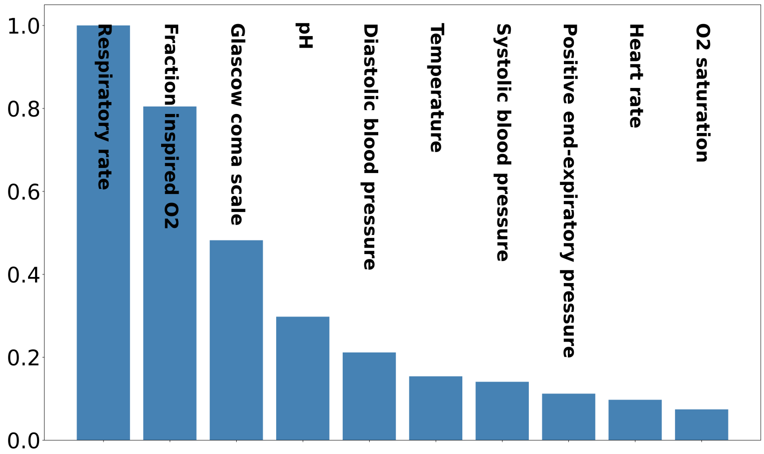

Figure 6: Relative variable importances for the onset of iV.

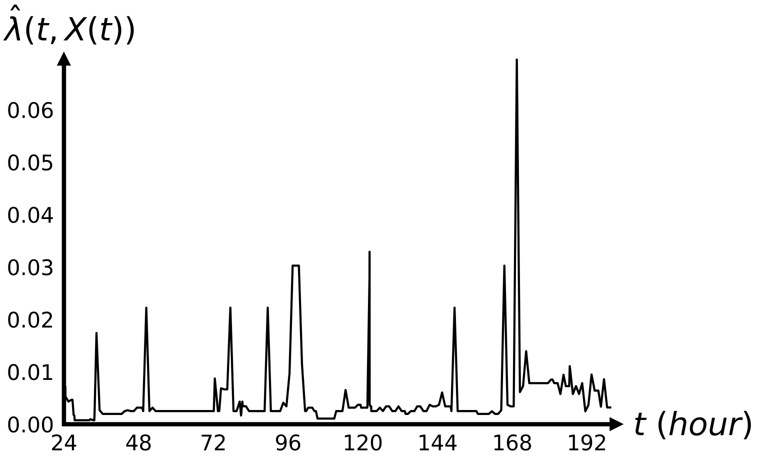

Figure 7: **BoXHED2.0** can be used as a real-time risk indicator.

example of how the iV risk evolves for a randomly selected patient is shown in Figure 7.

7. Discussion

With its core written in C++, **BoXHED2.0** is a Python package which is the only nonparametric boosting implementation for time-dependent survival settings beyond the classic right-censoring setup. It is based on the theoretically justified procedure in Lee *et al.* (2021), and runs at speeds that match even those of boosted parametric models. **BoXHED2.0** supports the use of GPUs and multicore CPUs, and is available from `http://github.com/BoXHED`.

8. Acknowledgments

AP and BJM are supported in part by National Institutes of Health grant R21-EB028486. DKKL is supported in part by National Institutes of Health grant R01-HL164405.

References

Aalen OO (1978). “Nonparametric Inference for a Family of Counting Processes.” *The Annals of Statistics*, **6**(4), 701–726. doi:10.1214/aos/1176344247.

Andersen PK, Abildstrom SZ, Rosthøj S (2002). “Competing Risks as a Multi-State Model.” *Statistical Methods in Medical Research*, **11**(2), 203–215. doi:10.1191/0962280202sm281ra.

Chen T, Guestrin C (2016). “**XGBoost**: A Scalable Tree Boosting System.” In *Proceedings of the 22nd ACM SIGKDD Int’l Conference on Knowledge Discovery & Data Mining*, pp. 785–794.

Clark PA, Lettieri CJ (2013). “Clinical Model for Predicting Prolonged Mechanical Ventilation.” *Journal of Critical Care*, **28**(5), 880–e1. doi:10.1016/j.jcrc.2013.03.013.

Friedman JH (2001). “Greedy Function Approximation: A Gradient Boosting Machine.” *The Annals of Statistics*, **29**, 1189–1232. doi:10.1214/aos/1013203451.

Harris CR, Millman KJ, Van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, Kern R, Picus M, Hoyer S, van Kerkwijk MH, Brett M, Haldane A, del Río JF, Wiebe M, Peterson P, Gérard-Marchant P, Sheppard K, Reddy T, Weckesser W, Abbasi H, Gohlke C, Oliphant TE (2020). “Array Programming with **NumPy**.” *Nature*, **585**(7825), 357–362. doi:10.1038/s41586-020-2649-2.

Harutyunyan H, Khachatrian H, Kale DC, Ver Steeg G, Galstyan A (2019). “Multitask Learning and Benchmarking with Clinical Time Series Data.” *Scientific Data*, **6**(1), 1–18. doi:10.1038/s41597-019-0103-9.

Hastie T, Tibshirani R, Friedman JH, Friedman JH (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, volume 2. Springer-Verlag.

Hothorn T (2020). “Transformation Boosting Machines.” *Statistics and Computing*, **30**(1), 141–152. doi:10.1007/s11222-019-09870-4.

Hothorn T, Bühlmann P, Kneib T, Schmid M, Hofner B (2010). “Model-Based Boosting 2.0.” *Journal of Machine Learning Research*, **11**, 2109–2113.

Hothorn T, Zeileis A (2021). “Predictive Distribution Modeling Using Transformation Forests.” *Journal of Computational and Graphical Statistics*, **30**(4), 1181–1196. doi:10.1080/10618600.2021.1872581.

Johnson A, Bulgarelli L, Pollard T, Horng S, Celi LA, Mark R (2021). “MIMIC-IV.”

Lee DKK, Chen N, Ishwaran H (2021). “Boosted Nonparametric Hazards with Time-Dependent Covariates.” *The Annals of Statistics*, **49**(4), 2101–2128. doi:10.1214/20-aos2028.

Lou Y, Caruana R, Gehrke J, Hooker G (2013). “Accurate Intelligible Models with Pairwise Interactions.” In *Proceedings of the 19th ACM SIGKDD Int’ Conference on Knowledge Discovery & Data Mining*, pp. 623–631.

Ridgeway G (1999). "The State of Boosting." *Computing Science and Statistics*, **31**, 172–181. doi:10.1007/978-1-4612-1538-7_9.

Schmid M, Hothorn T (2008). "Flexible Boosting of Accelerated Failure Time Models." *BMC Bioinformatics*, **9**, 269. doi:10.1186/1471-2105-9-269.

The **pandas** Development Team (2020). *pandas-dev/pandas:* ***pandas***. doi:10.5281/zenodo.3509134.

Wang X, Pakbin A, Mortazavi BJ, Zhao H, Lee DKK (2020). "**BoXHED**: Boosted eXact Hazard Estimator with Dynamic covariates." In *International Conference on Machine Learning*, pp. 9973–9982. PMLR.

Yu L, Halalau A, Dalal B, Abbas AE, Ivascu F, Amin M, Nair GB (2021). "Machine Learning Methods to Predict Mechanical Ventilation and Mortality in Patients with COVID-19." *PLOS One*, **16**(4), e0249285. doi:10.1371/journal.pone.0249285.

A. Data preprocessing

Following the standard treatment of time-dependent survival data in the Cox model, the functional data sample $\{X_i(t), Y_i(t), N_i(t)\}_t$ for subject i is fed into **BoXHED2.0** in the tabular form

$$\begin{pmatrix} \underline{\tau}_{i1} & \overline{\tau}_{i1} & \chi_{i1} & \Delta_{i1} \\ \underline{\tau}_{i2} & \overline{\tau}_{i2} & \chi_{i2} & \Delta_{i2} \\ & & \vdots & \\ \underline{\tau}_{iJ_i} & \overline{\tau}_{iJ_i} & \chi_{iJ_i} & \Delta_{iJ_i} \end{pmatrix}, \tag{9}$$

where the j-th epoch (row) represents a time interval $(\underline{\tau}_{ij}, \overline{\tau}_{ij}]$ over which the subject is at-risk, i.e., $Y_i(t) = 1$. Note that the first epoch need not start at $\underline{\tau}_{i1} = 0$, and the endpoint $\overline{\tau}_{ij}$ of the j-th epoch need not equal the beginning $\underline{\tau}_{i,j+1}$ of the $(j+1)$-th epoch. The value of the covariate in the j-th epoch is χ_{ij}, and $\Delta_{ij} = 1$ if the subject experienced an event at $\overline{\tau}_{ij}$, and is zero otherwise. The goal is to preprocess the data so that the quantities (6),

$$U_k = \sum_{i=1}^{n} \int_{(t,X_i(t)) \in A_k} Y_i(t) e^{F_m(t,X_i(t))} dt, \qquad V_k = \sum_{i=1}^{n} \int_{(t,X_i(t)) \in A_k} dN_i(t),$$

defined for the left and right daughter regions A_L and A_R can be computed efficiently without numerical integration.

The insight behind the preprocessing step stems from the observation that the tree learners $g_0(t,x), g_1(t,x), \ldots$ in $F_m(t,x) = F_0 - \nu \sum_{q=0}^{m-1} g_q(t,x)$ are piecewise-constant over the disjoint time-covariate regions (5):

$$B_\ell = \left\{ (t,x) : \begin{pmatrix} t^{(\ell_0)} < t \le t^{(\ell_0+1)} \\ x^{(1,\ell_1)} < x^{(1)} \le x^{(1,\ell_1+1)} \\ \vdots \\ x^{(p,\ell_p)} < x^{(p)} \le x^{(p,\ell_p+1)} \end{pmatrix} \right\},$$

where $\{t^{(\ell_0)}\}_{\ell_0}$ is the set of candidate split points for time t, and $\{x^{(k,\ell_k)}\}_{\ell_k}$ is the set of candidate splits for the k-th covariate $x^{(k)}$. It follows that for a fixed value of x, $F_m(t,x)$ is a constant function of time between any two consecutive candidate split points $t^{(\ell_0)}$ and $t^{(\ell_0+1)}$. Suppose for a moment that each epoch $(\underline{\tau}_{ij}, \overline{\tau}_{ij}]$ in the input data is completely contained within some interval $(t^{(\ell_0)}, t^{(\ell_0+1)}]$. Since each daughter region $A_{k \in \{L,R\}}$ is the union of some subset of the regions (5), U_k and V_k reduce to the weighted sums

$$U_k = \sum_{(\underline{\tau}_{ij}+,\chi_{ij}) \in A_k} w_{ij} e^{F_m(\underline{\tau}_{ij}+,\chi_{ij})}, \qquad V_k = \sum_{(\underline{\tau}_{ij}+,\chi_{ij}) \in A_k} \Delta_{ij}, \tag{10}$$

where $w_{ij} = \overline{\tau}_{ij} - \underline{\tau}_{ij}$. The choice of $\underline{\tau}_{ij}+$ in (10) is due to the fact that $F_m(t,x)$ is constant over half-open intervals of the form $(t^{(\ell_0)}, t^{(\ell_0+1)}]$.

Similarly, the likelihood risk (3) evaluated at F_m, $R_n(F_m)$, can be computed as the sum

$$\frac{1}{n} \sum_{ij} \Big\{ w_{ij} e^{F_m(\underline{\tau}_{ij}+,\chi_{ij})} - \Delta_{ij} F_m(\underline{\tau}_{ij}+, \chi_{ij}) \Big\}. \tag{11}$$

Thus the 'sufficient statistics' for **BoXHED2.0** are $\{\underline{\tau}_{ij}, w_{ij}, \chi_{ij}, \Delta_{ij}\}_{ij}$, and (10)–(11) can be easily computed from the tree ensemble predictions for $F_m(t,x)$.

To excise explicit numerical integration from **BoXHED2.0**, we need to put the data (9) in a form where each epoch is completely contained within some interval $(t^{(\ell_0)}, t^{(\ell_0+1)}]$. This is done by applying four operations to the rows of (9). The first two are:

1. Any epoch $(\underline{\tau}_{ij}, \overline{\tau}_{ij}]$ that contains one of the candidate split points $t^{(\ell_0)}$ is split into $(\underline{\tau}_{ij}, t^{(\ell_0)}]$ and $(t^{(\ell_0)}, \overline{\tau}_{ij}]$. If an epoch spans multiple split points then it is split into a number of shorter epochs.

2. Transform the (newly created) rows $\{\underline{\tau}_{ij}, \overline{\tau}_{ij}, \chi_{ij}, \Delta_{ij}\}_{ij}$ into $\{\underline{\tau}_{ij}, w_{ij}, \chi_{ij}, \Delta_{ij}\}_{ij}$.

The time complexity for achieving the above is $\mathcal{O}(n \log |\{t^{(\ell_0)}\}_{\ell_0}|)$. Figure 8 illustrates the operations on a simple numerical example. The left table describes the event histories for two subjects. The first subject experienced an event at the end of their first epoch ($t = 0.13$), returns to the sample at $t = 0.15$, and becomes lost to follow up at $t = 0.25$. A similar story can be told for the second subject.

The second set of required operations stem from the fact that most boosting implementations, including **XGBoost**, use tree splits of the form '$<$' for the left daughter and '$\geq$' for the right. In particular, this implies time splits of the form $[\cdot, \cdot)$. This is incompatible with the epochs for s urvival data, which are intrinsically of the form $(\cdot, \cdot]$. To see why, consider for example a subject who experiences an event at a candidate split point $t^{(\ell_0)}$. Since the candidate split points are not sampled from an absolutely continuous distribution, but are instead chosen from the observed data, the probability of this happening is not zero. Under the splitting convention $[\cdot, \cdot)$, the observed event will be counted towards some region $[t^{(\ell_0)}, t^{(\ell_0')})$, even though the subject was never at-risk there. The hazard MLE for that region would then be one (number of events in region) divided by zero (time spent at-risk in region).[6] For mathematical consistency, the lower end of the time interval must be open and the upper end closed. For notational consistency we will also apply the same convention to each covariate.

Fortunately, through additional processing, we can still use **XGBoost**'s splitting convention to fit trees with leaf nodes of the form $(t^{(\ell_0)}, t^{(\ell_0')}] \times (x^{(1,\ell_1)}, x^{(1,\ell_1')}] \times \ldots \times (x^{(p,\ell_p)}, x^{(p,\ell_p')}]$. Let us first discuss the processing of covariates. Suppose that a leaf region $B_{m,\ell}$ contains an epoch whose k-th covariate value $\chi_{ij}^{(k)}$ coincides with one of the candidate split points $x^{(k,\ell_k)}$. A **XGBoost** cut at that location would assign the epoch to the right daughter $B_{m,\ell} \cap \{x^{(k)} \geq x^{(k,\ell_k)}\}$. To ensure that it will be assigned to the left daughter $B_{m,\ell} \cap \{x^{(k)} < x^{(k,\ell_k)}\}$ instead, we apply the map

3. $\chi_{ij}^{(k)} \mapsto x^{(k,\ell_k-1)}$ for $\chi_{ij}^{(k)} \in (x^{(k,\ell_k-1)}, x^{(k,\ell_k)}]$.

Note that the covariate values in the interior of $(x^{(k,\ell_k-1)}, x^{(k,\ell_k)}]$ are also mapped to the candidate split point $x^{(k,\ell_k-1)}$. This caps the number of unique covariate values in the data to the maximum number of candidate split points set by the user. If fewer than 256 candidates are used, the covariate values can be stored as a byte, which is useful for GPU training, given memory scarcity.

With the application of Step 3, the fitted value for a region $[x^{(k,\ell_k)}, x^{(k,\ell_k')})$ in fact represents the fitted value for $(x^{(k,\ell_k)}, x^{(k,\ell_k')}]$ in the **BoXHED2.0** hazard estimator. Accordingly, the

[6] The format for the **BoXHED1.0** data allowed for similar situations where $V_k > 0$ while $U_k = 0$. An ad-hoc imputation was used to remove the pathology (Chapter 4.1.2 in Wang *et al.* 2020). The **BoXHED2.0** data format is designed to rule out this possibility from the start.

$$\left(\begin{array}{ccccc} i & \underline{\tau} & \overline{\tau} & \chi & \Delta \\ \hline 1 & 0.01 & 0.13 & 0.27 & 1 \\ 1 & 0.15 & 0.25 & 0.51 & 0 \\ 2 & 0.06 & 0.10 & 0.81 & 1 \\ 2 & 0.13 & 0.25 & 0.92 & 0 \end{array}\right) \to \left(\begin{array}{ccccc} i & \underline{\tau} & \overline{\tau} & \chi & \Delta \\ \hline 1 & 0.01 & \mathbf{0.10} & 0.27 & \mathbf{0} \\ \mathbf{1} & \mathbf{0.10} & \mathbf{0.13} & \mathbf{0.27} & \mathbf{1} \\ 1 & 0.15 & 0.25 & 0.51 & 0 \\ 2 & 0.06 & 0.10 & 0.81 & 1 \\ 2 & 0.13 & \mathbf{0.15} & 0.92 & 0 \\ \mathbf{2} & \mathbf{0.15} & \mathbf{0.25} & \mathbf{0.92} & \mathbf{0} \end{array}\right) \to \left(\begin{array}{ccccc} i & \underline{\tau} & w & \chi & \Delta \\ \hline 1 & 0.01 & \mathbf{0.09} & 0.27 & 0 \\ 1 & 0.10 & \mathbf{0.03} & 0.27 & 1 \\ 1 & 0.15 & \mathbf{0.10} & 0.51 & 0 \\ 2 & 0.06 & \mathbf{0.04} & 0.81 & 1 \\ 2 & 0.13 & \mathbf{0.02} & 0.92 & 0 \\ 2 & 0.15 & \mathbf{0.10} & 0.92 & 0 \end{array}\right)$$

Figure 8: Numerical illustration of Steps 1 and 2. The candidate split points $\{t^{(\ell_0)}\}_{\ell_0}$ are set as $\{0,\ 0.10,\ 0.15\}$, where 0 is always included by default (but will not be split on). The first epoch in the original data (left) is split into the first two rows in the middle table, and the last epoch in the original data is split into the last two rows in the middle table.

$$\left(\begin{array}{ccccc} i & \underline{\tau} & w & \chi & \Delta \\ \hline 1 & 0.01 & 0.09 & 0.27 & 0 \\ 1 & 0.10 & 0.03 & 0.27 & 1 \\ 1 & 0.15 & 0.10 & 0.51 & 0 \\ 2 & 0.06 & 0.04 & 0.81 & 1 \\ 2 & 0.13 & 0.02 & 0.92 & 0 \\ 2 & 0.15 & 0.10 & 0.92 & 0 \end{array}\right) \longrightarrow \left(\begin{array}{ccccc} i & \underline{\tau} & w & \chi & \Delta \\ \hline 1 & \mathbf{0} & 0.09 & 0.27 & 0 \\ 1 & 0.10 & 0.03 & 0.27 & 1 \\ 1 & 0.15 & 0.10 & \mathbf{0.27} & 0 \\ 2 & \mathbf{0} & 0.04 & \mathbf{0.51} & 1 \\ 2 & \mathbf{0.10} & 0.02 & \mathbf{0.81} & 0 \\ 2 & 0.15 & 0.10 & \mathbf{0.81} & 0 \end{array}\right)$$

Figure 9: Steps 3 and 4 of processing the last table in Figure 8. The candidate split points $\{x^{(1,\ell_1)}\}_{\ell_1}$ for the covariate are set as $\{0.51,\ 0.81\}$ in this example. Notice that the covariate value for the third row, 0.51, coincides with the smallest candidate split point. In the table on the right, we set it to the smallest value (0.27) observed in the data. This is always possible because under **XGBoost**'s splitting convention, the smallest value will never be chosen as a split point.

`hazard` function in **BoXHED2.0** maps any new covariate value that coincides with a candidate split point $x^{(k,\ell_k)}$ back to $x^{(k,\ell_k-1)}$, before running it through the fitted trees to obtain the estimated hazard value.

For the processing of time, this was already partly accomplished by our indexing of the epoch $(\underline{\tau}_{ij}, \overline{\tau}_{ij}]$ with $\underline{\tau}_{ij}$ rather than $\overline{\tau}_{ij}$ in (10) and (11). Therefore, if a leaf region $B_{m,\ell}$ is split on time into daughters $B_{m,\ell} \cap \{t < \underline{\tau}_{ij}\}$ and $B_{m,\ell} \cap \{t \geq \underline{\tau}_{ij}\}$, the epoch will be correctly assigned to the latter. Thus even if $\underline{\tau}_{ij}$ coincides with a candidate split point $t^{(\ell_0)}$, there is no need to re-map. However, given the GPU memory discussion above, we still need to perform the following:

4. $\underline{\tau}_{ij} \mapsto t^{(\ell_0-1)}$ for $\underline{\tau}_{ij} \in (t^{(\ell_0-1)}, t^{(\ell_0)})$.

Steps 3 and 4 are illustrated in Figure 9.

Similar to the covariate regions, the fitted value for $[t^{(\ell_0)}, t^{(\ell_0')})$ represents the estimated hazard value for $(t^{(\ell_0)}, t^{(\ell_0')}]$ in the **BoXHED2.0** estimator. Hence the `hazard` function in **BoXHED2.0** also maps any new time value that coincides with a candidate split point $t^{(\ell_0)}$ back to $t^{(\ell_0-1)}$.

B. Preprocessing for invasive ventilation data in MIMIC-IV

We begin by applying the MIMIC-III preprocessing pipeline to MIMIC-IV in order to extract participants and a commonly available set of variables (Harutyunyan, Khachatrian, Kale, Ver Steeg, and Galstyan 2019). Given the severity of a surgical tracheotomy, a patient is censored at the time of receiving one should this happen. Following the literature on forecasting iV, we commence risk estimation after 24 hours (Clark and Lettieri 2013), resulting in a cohort of 29,108 patients (36,068 ICU stays). In addition to the features extracted by the pipeline (demographics and vital measurements), we add respiratory related measurements that are pertinent to iV, namely, inspiratory/expiratory, oxygen flow and consumption, and respiratory rate. Due to the functional nature of the data, we carry the last observed value forward for missing value imputation.

The complete list of the extracted variables are as follows:

Variable	MIMIC-IV table	Type
Capillary refill rate	chartevents	categorical
Ethnicity	chartevents	categorical
Gender	chartevents	categorical
Diastolic blood pressure	chartevents	continuous
Fraction inspired oxygen	chartevents	continuous
Glascow coma scale total	chartevents	continuous
Glucose	chartevents, labevents	continuous
Heart Rate	chartevents	continuous
Mean blood pressure	chartevents	continuous
Oxygen saturation	chartevents, labevents	continuous
Systolic blood pressure	chartevents	continuous
Temperature	chartevents	continuous
pH	chartevents, labevents	continuous
Age	chartevents	continuous
Height	chartevents	continuous
Weight	chartevents	continuous
Inspired O2 Fraction	chartevents	continuous
Respiratory Rate	chartevents	continuous
O2 saturation pulseoxymetry	chartevents	continuous
PEEP set	chartevents	continuous
Inspired Gas Temp.	chartevents	continuous
Paw High	chartevents	continuous
Vti High	chartevents	continuous
Fspn High	chartevents	continuous
Apnea Interval	chartevents	continuous
Tidal Volume (set)	chartevents	continuous
Tidal Volume (observed)	chartevents	continuous
Minute Volume	chartevents	continuous
Respiratory Rate (Set)	chartevents	continuous
Respiratory Rate (spontaneous)	chartevents	continuous

Continued on next page

Table 1 – continued from previous page

Variable	MIMIC-IV table	Type
Respiratory Rate (Total)	chartevents	continuous
Peak Insp. Pressure	chartevents	continuous
Plateau Pressure	chartevents	continuous
Mean Airway Pressure	chartevents	continuous
Total PEEP Level	chartevents	continuous
Inspiratory Time	chartevents	continuous
Expiratory Ratio	chartevents	continuous
Inspiratory Ratio	chartevents	continuous
Ventilator Tank #1	chartevents	continuous
Ventilator Tank #2	chartevents	continuous
Tidal Volume (spontaneous)	chartevents	continuous
PSV Level	chartevents	continuous
O2 Flow	chartevents	continuous
O2 Flow (additional cannula)	chartevents	continuous
Flow Rate (L/min)	chartevents	continuous
CO2 production	chartevents	continuous
Cuff Pressure	chartevents	continuous
ETT Position Change	chartevents	continuous
ETT Re-taped	chartevents	continuous
Spont Vt	chartevents	continuous
Spont RR	chartevents	continuous
MDI #1 Puff	chartevents	continuous
Cuff Volume (mL)	chartevents	continuous
Trach Care	chartevents	continuous
MDI #2 Puff	chartevents	continuous
MDI #3 Puff	chartevents	continuous
Negative Insp. Force	chartevents	continuous
Vital Cap	chartevents	continuous
BiPap O2 Flow	chartevents	continuous
PCV Level	chartevents	continuous
BiPap EPAP	chartevents	continuous
BiPap IPAP	chartevents	continuous
Pinsp (Draeger only)	chartevents	continuous
Recruitment Duration	chartevents	continuous
PeCO2	chartevents	continuous
Recruitment Press	chartevents	continuous
Nitric Oxide	chartevents	continuous
Nitric Oxide Tank Pressure	chartevents	continuous
Transpulmonary Pressure (Exp. Hold)	chartevents	continuous
Transpulmonary Pressure (Insp. Hold)	chartevents	continuous
P High (APRV)	chartevents	continuous
P Low (APRV)	chartevents	continuous
T High (APRV)	chartevents	continuous

Continued on next page

Table 1 – continued from previous page

Variable	MIMIC-IV table	Type
T Low (APRV)	chartevents	continuous
Small Volume Neb Dose #2	chartevents	continuous
ATC %	chartevents	continuous
BiPap bpm (S/T -Back up)	chartevents	continuous
Peak Exp Flow Rate	chartevents	continuous
Vd/Vt Ratio	chartevents	continuous

Table 1: The selected clinical variables for iV risk estimation in MIMIC-IV.

To track iV recurrence, we also add the cumulative number of past iVs as well as time since last iV (if any).

We randomly split this patient cohort into training and test sets using the approach in Harutyunyan *et al.* (2019) (with modification for MIMIC-IV), resulting in 24,764 patients (30,716 ICU stays) in the training set and 4,344 patients (5,352 ICU stays) in the test set. A 5-fold cross validation is used on the training set to select **BoXHED2.0** hyperparameters. The tuned hyperparameters for **BoXHED2.0** are {100 trees, depth 3}.

C. Synthetic dataset for scalability analysis

Letting $X(t)$ be a piecewise-constant covariate with values drawn from $U(0,1]$, we simulate event times from the following hazard function:

$$\lambda(t, X(t)) = B(t, 2, 2) \times B(X(t), 2, 2), t \in (0, 1],$$

where $B(\cdot, a, a)$ is the PDF of the Beta distribution (with shape and scale a). This means that λ takes the form of Beta PDFs. If the event has not occurred by $t = 1$ the subject is administratively censored at that point. In addition to $X(t)$, we add up to 40 irrelevant covariates to the dataset. The event histories for 5,000 subjects are drawn for training following Wang *et al.* (2020). For benchmarking against boosted parametric models, we draw up to 10 million rows of data.

Affiliation:

Arash Pakbin, Bobak J. Mortazavi
Department of Computer Science & Engineering
Texas A&M University
College Station TX, United States of America
E-mail: a.pakbin@tamu.edu, bobakm@tamu.edu

Xiaochen Wang
Department of Biostatistics
Yale University
New Haven CT, United States of America
E-mail: xcwang11@gmail.com

Donald K. K. Lee
Goizueta Business School and Department of Biostatistics & Bioinformatics
Emory University
Atlanta GA, United States of America
E-mail: donald.lee@emory.edu

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
July 2025, Volume 113, Issue 3 *Submitted:* 2023-01-13
doi:10.18637/jss.v113.i03 *Accepted:* 2023-11-15