**Confidentiality for Cox model**

The WebDISCO method for estimating a horizontally partitioned Cox model does not fully protect data confidentiality because predictors can potentially be identified by the central server. This vulnerability arises because data is sent for each event time. When only one event occurs during a specific time period, the predictors for that individual may become identifiable, compromising their anonymity.

To address this issue, it is crucial to ensure that there is never a single event associated with a single event time. One way to achieve this is by dividing the data into intervals where multiple patients are grouped together. For the WebDISCO method, to maintain confidentiality, an interval should either contain at least x events (five or more events, for example), or no events/censoring at all. Various methods can be used to achieve this grouping.

In the examples of the proposed methods below, data is grouped into pairs for simplicity, without considering the status (event or censoring). However, the actual implementation ensures that each group must contain at least five events (which may be changed by the user) or be entirely empty, in order to protect individual confidentiality.

**Averaging**

The data is ordered by time, and values are grouped and given a new time, which is the average of all values of time within a given group.

**Example.**

For the example, we assume we want to group data into groups of 2.

| *time* | $p_1$ | $p_2$ |
|---|---|---|
| *2* | 43 | 0 |
| *4* | 24 | 0 |
| *5* | 41 | 1 |
| *6* | 37 | 1 |
| *9* | 53 | 0 |
| *11* | 33 | 1 |
| *12* | 39 | 1 |
| *17* | 45 | 0 |

**Before**

| *time* | $p_1$ | $p_2$ |
|---|---|---|
| *3* | 43 | 0 |
| *3* | 24 | 0 |
| *5.5* | 41 | 1 |
| *5.5* | 37 | 1 |
| *10* | 53 | 0 |
| *10* | 33 | 1 |
| *14.5* | 39 | 1 |
| *14.5* | 45 | 0 |

**After**

Where $p_1$ and $p_2$ are predictors.

For this method, no communication is required between sites; all computations are performed locally in a single iteration. This method tends to generate the most intervals. A few points to note:

- Since values (event times and censored times) are grouped into sets of five in ascending order, two data points with the same time value might end up in different intervals. As a

result, they could be reassigned different time values, potentially losing their equality. This also means that the original time ordering in the .csv data file may affect the resulting groups.

- The positional order of values can be altered by this method. For example, in one site, a value might be assigned a lower position, while a nearby value at another site might be increased. This positional "switching" could introduce errors. However, with large datasets, the intervals tend to be quite small, so this error might not be significant.
- If the last interval is too small (not enough values), the last two intervals will be merged.

**Uniform Intervals (with cutoff)**

The period of the study is split into uniform intervals that contain at least x subjects with event times (not censored times) per interval. The last few values are excluded, as they can be spread far apart, and could affect negatively the interval size.

### Example

We assume we want to group data into groups of 2. For this example, we chose to exclude 0% of the data at the left of the distribution, and 15% at the right. This means the value 17 is excluded from the data.

The minimum value is 2, and the maximum value is 12. The intervals must fall within this range. If the intervals can be increased in increments of 0.05, the smallest possible interval size is 3.55.

| *time* | $p_1$ | $p_2$ |
|---|---|---|
| *2* | 43 | 0 |
| *4* | 24 | 0 |
| *5* | 41 | 1 |
| *6* | 37 | 1 |
| *9* | 53 | 0 |
| *11* | 33 | 1 |
| *12* | 39 | 1 |
| *17* | 45 | 0 |

**Before**

| *time* | $p_1$ | $p_2$ |
|---|---|---|
| *1* | 43 | 0 |
| *1* | 24 | 0 |
| *1* | 41 | 1 |
| *2* | 37 | 1 |
| *2* | 53 | 0 |
| *3* | 33 | 1 |
| *3* | 39 | 1 |
| - | - | - |

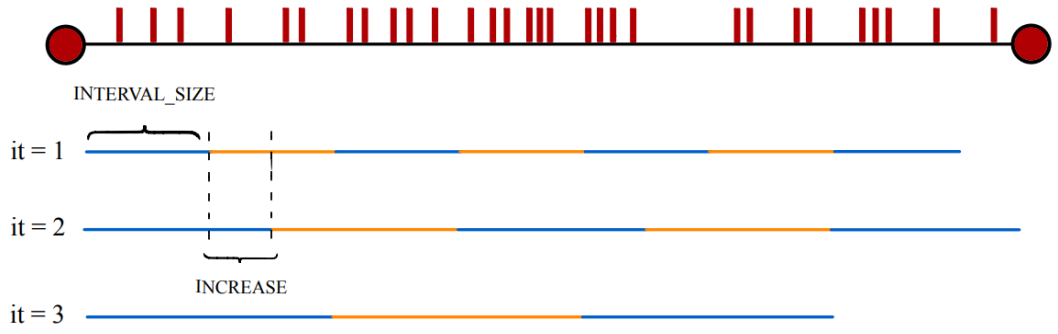
**After**

The intervals would be [2, 5.55[, [5.55, 9.1[, [9.1, 12.65[, but since that last interval goes over the maximum value, the last interval is not created. The values over 9.1 will be added to the previous interval.

The intervals are: [2, 5.55[, [5.55, ∞[

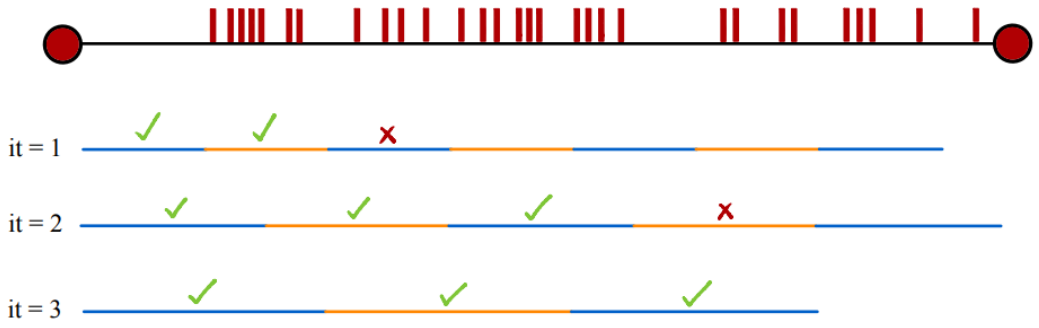
The excluded values, here 17, will not be used.

To be able to choose the smallest possible interval size, each site must compute a matrix locally, which contains information on all possible interval site for that site. Here is a simple example:



In this example, *interval_size* is the initial interval size (and thus the smallest interval size considered), and *increase* is the value by which the size of the interval increases each iteration.

One iteration at a time, each interval is checked to see if they respect all the conditions. To be valid, an interval can contain either x values or more (5 in the picture below), or zero values (neither event times nor censored times). This figure illustrates this.




Once there is an interval that is not valid in one iteration, the rest do not need to be checked, as the interval size is not valid somewhere. The matrix produced here would be:

| It | Binary_ outcome |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 1 |

Only the last iteration could be valid here, as it is the only one where all intervals respect the conditions. The last few values not in an interval will be added to the last interval, even if there were enough values for another interval (which may happen whenever the next interval would end up outside the range of the data).[1]

This method requires communication between sites and a global server to ensure all sites have the same intervals. The communication process is as follows:

1. Each site sends its cutoff values (min and max) to the global server according to the percentage of excluded values determined by the global server. The cutoff value is the last time value included after excluding the specified percentage of values.
2. The global server selects the lowest minimum cutoff value and the highest maximum cutoff value received from all sites and sends it back to each site.
3. Knowing both cutoff values, each site calculates a matrix, which contains information about which interval size respects all conditions, and then sends this matrix to the global server.
4. The global server selects the smallest interval size that works for all sites and generates a list with all interval's borders.
5. Each site can now aggregate data locally.

[1] While this may seem strange at first, it is not possible to simply create a new interval, since all sites would need to agree to it, and this would cost an additional communication.

**Non-uniform Intervals**

The period of the study is split into non-uniform intervals. These intervals are the smallest possible size that contains x subjects with event times (not censored times).

### Example

We assume we want to group data into groups of 2.

Each interval should contain the minimum number of values possible. Ideally, each interval will contain exactly 2 values.

| *time* | $p_1$ | $p_2$ |
|---|---|---|
| *2* | 43 | 0 |
| *4* | 24 | 0 |
| *5* | 41 | 1 |
| *6* | 37 | 1 |
| *9* | 53 | 0 |
| *11* | 33 | 1 |
| *12* | 39 | 1 |
| *17* | 45 | 0 |

**Before**

| *time* | $p_1$ | $p_2$ |
|---|---|---|
| *1* | 43 | 0 |
| *1* | 24 | 0 |
| *2* | 41 | 1 |
| *2* | 37 | 1 |
| *3* | 53 | 0 |
| *3* | 33 | 1 |
| *4* | 39 | 1 |
| *4* | 45 | 0 |

**After**

Intervals: [2, 5[, [5, 9[, [9, 12[, [12, ∞[.

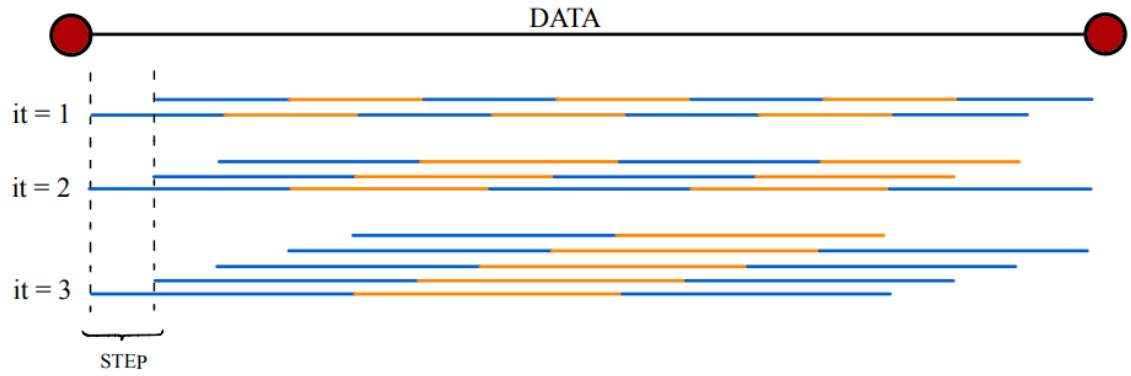
However, all the sites must agree on these intervals (see next section).

To minimize the amount of communication between sites, an algorithm was implemented that requires only one back and forth with the central server.

First, the local data is split into intervals:


Where *interval size* is the initial size of the interval (and thus the smallest interval size considered), and *increase* is the amount that the interval increases every iteration. Every site can also exclude a percentage of data on the left and right side of the distribution, as these values can be spread far apart.

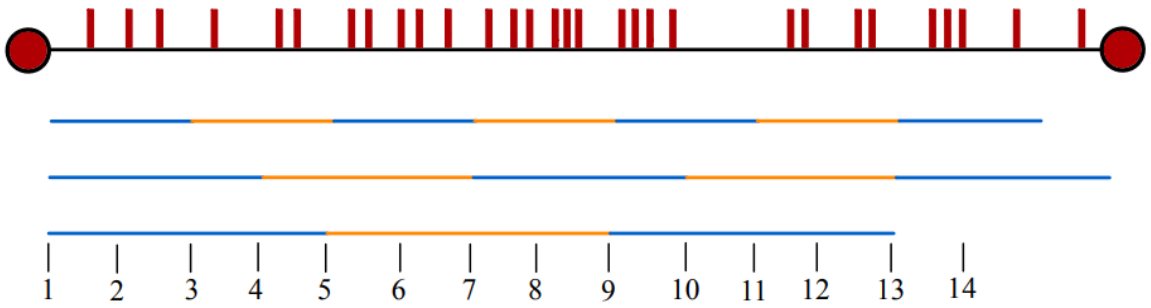
To find the smallest interval size possible, it can be interesting to have overlapping intervals:




The value *step* determines the shift between the start of the intervals. In the first example, *step* was equal to *interval size*. To make sure that all values are contained within intervals of decent size, *step* should always be equal or smaller than the initial *interval size*, or values may fall between two intervals. The best practice, when possible, is for *interval size* and *step* to be equal to the smallest difference between two values in the dataset.

Once the intervals parameters have been determined and the intervals have been computed, the number of values (event times) in each interval is checked. If the interval contains either x values or more, or zero values (neither event times nor censored times), the value 1 is put at associated position in the binary outcome matrix.

For this simple example:




If we want at least 5 values in each interval, the binary output matrix obtained from what we can see in the previous picture would be:

| | | **Interval start position** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
| **Interval length** | **2** | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | **3** | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| | **4** | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

For example, consider intervals that start at position 1. Only an interval of size 4 contains at least 5 values. The interval of size 2 contains only 3 values, and the interval of size 3 contains 4 values.

Another example is at position 5. Here, the interval of size 2 contains 5 values, and the interval of size 4 contains 11 values. No interval of size 3 starts at position 5, which explains the value of 0.

And finally, the interval size can be chosen, by selecting the smallest interval size possible.

| | | Interval start position | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| Interval length | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 3 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| | 4 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Each site generates their own binary output matrix. The central server will sum all these matrices. If there are 3 sites, the global output matrix may look something like this:

| | | Interval start position | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| Interval length | 2 | 0 | 0 | 1 | 0 | 3 | 0 | 3 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| | 3 | 1 | 1 | 2 | 1 | 3 | 1 | 3 | 0 | 2 | 0 | 0 | 0 | 3 | 0 |
| | 4 | 3 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 |

Instead of selecting the smallest interval size where there is a 1, the interval size selected should be the one where the value is equal to the number of sites (in this case 3). This guarantees the interval size meets our criteria in all of the sites.

The algorithm used for interval size selection is this one:

**At interval position 1**: interval size 4 is selected. The next position to check is $1 + 4 = 5$.

**At interval position 5**: interval size 2 is selected. The next position to check is $5 + 2 = 7$.

**At interval position 7**: interval size 2 is selected. The next position to check is $7 + 2 = 9$.

**At interval position 9**: interval size 4 is selected. The next position to check is $9 + 4 = 13$.

**At interval position 13**: interval size 3 is selected. It is the last interval.

It should be noted that it is only possible to place a 1 if there are enough values remaining to create subsequent intervals, or if there are no values left. This means that if an interval could be created (containing 5 values) but leaves only 3 values outside the interval, the interval will not be created. In this case, the only possible interval is the one containing 8 values, as it leaves no values behind.

For this example, this is what the algorithm would give: