Fast thresholded concordance probability for evolutionary optimization

Reference:
Ponnet Jolien, Raymaekers Jakob, Verdonck Tim.- Fast thresholded concordance probability for evolutionary optimization
Swarm and Evolutionary Computation - ISSN 2210-6502 - 78(2023), 101260
Full text (Publisher's DOI): https://doi.org/10.1016/J.SWEVO.2023.101260
To cite this reference: https://hdl.handle.net/10067/2013690151162165141

Fast thresholded concordance probability for evolutionary optimization

Jolien Ponnet[a], Jakob Raymaekers[b], Tim Verdonck[a,c]

[a]*Department of Mathematics, KU Leuven, Celestijnenlaan 200B, Leuven, 3001, Belgium*
[b]*Department of Quantitative Economics, Maastricht University, Tongersestraat 53, Maastricht, 6211 LM, The Netherlands*
[c]*Department of Mathematics, UAntwerp - imec, Middelheimlaan 1, Antwerp, 2020, Belgium*

Abstract

The concordance probability is an extension of the popular area under the curve (AUC) which is commonly used to measure the accuracy of a predictive model. It can be extended to the thresholded and weighted concordance probability which are more appropriate for some applications. The naive way of estimating this measure requires a quadratic computation time, which is prohibitive for large data sets. We propose a new algorithm that computes the weighted thresholded concordance probability in linearithmic time, which is proven and empirically confirmed. This unlocks the possibility of calculating the thresholded concordance probability in a big data world, and makes it possible to base the fitness function of a machine learning algorithm on the concordance probability. These applications are successfully illustrated by two real examples from the insurance sector. The first one focuses on feature selection based on the concordance probability using a binary particle swarm optimization. In the second application, we use a genetic algorithm to optimize a loss function based on the concordance probability. Since both of these applications require evaluating the concordance probability a very high number of times, a huge decrease in computation time is obtained using our fast algorithm. Moreover, it is shown that the neural network optimized for the concordance probability with the genetic algorithm outperforms the traditional benchmark methodology, i.e. a classical neural network optimized for the deviance. The applicability of our fast algorithm extends beyond these illustrations and unlocks various new uses of the thresholded and weighted concordance probability.

Keywords: c-index, quasilinear, mergesort, binary particle swarm optimization, neural network

1. Introduction

The concordance probability, also known as the C-index, is a popular measure for the discriminatory ability of a model. More specifically, it measures the probability that the

Email address: Tim.Verdonck@uantwerpen.be (Tim Verdonck)

order between a comparable pair of observations is kept, when comparing their predictions obtained by the model. In case we have a binary response variable Y, the concordance probability is equivalent to the probability that a random observation with response $Y = 0$ has a smaller prediction than a random observation with response $Y = 1$ [1]. Hence, in this binary setting, the concordance probability C can be formulated as:

$$C = \mathrm{P}\left(\pi(\boldsymbol{X}_i) > \pi(\boldsymbol{X}_j) \mid Y_i = 1,\ Y_j = 0\right), \tag{1}$$

where $\boldsymbol{X}_j$ represents the explanatory variables corresponding with observation j, such that its prediction $\pi(\boldsymbol{X}_j)$ equals $\mathrm{P}(Y_j = 1|\boldsymbol{X_j})$. This concordance probability is most commonly estimated by dividing the number of concordant pairs N_c by the number of comparable pairs N:

$$\widehat{C} = \frac{N_c}{N} = \frac{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} I\Big(\widehat{\pi}(\boldsymbol{x}_i) > \widehat{\pi}(\boldsymbol{x}_j),\ y_i = 1,\ y_j = 0\Big)}{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} I\Big(\widehat{\pi}(\boldsymbol{x}_i) \neq \widehat{\pi}(\boldsymbol{x}_j),\ y_i = 1,\ y_j = 0\Big)}, \tag{2}$$

with n the number of observations and $I(\cdot)$ the indicator function.

When the response variable is continuous, the concordance probability is easily extended as follows:

$$C = \mathrm{P}\left(\pi(\boldsymbol{X}_i) > \pi(\boldsymbol{X}_j) \mid Y_i > Y_j\right), \tag{3}$$

Here, $\pi(\boldsymbol{X}_i)$ still corresponds to the prediction of Y_i, e.g. in case of a linear model we have that $\pi(\boldsymbol{X}_i) = X_i\beta$ with β the model coefficients. An equivalent definition for C is the probability that a random comparable pair of responses with their predicted values is a concordant pair. Here we use the fact that two pairs $(\pi(\boldsymbol{X}_i), Y_i)$ and $(\pi(\boldsymbol{X}_j), Y_j)$ are concordant when $\mathrm{sgn}(\pi(\boldsymbol{X}_i) - \pi(\boldsymbol{X}_j)) = \mathrm{sgn}(Y_i - Y_j)$. It is clear that the closer the concordance probability is to 1, the better discriminatory ability of the model. The natural estimator is

$$\widehat{C} = \frac{N_c}{N} = \frac{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} I\Big(\widehat{\pi}(\boldsymbol{x}_i) > \widehat{\pi}(\boldsymbol{x}_j),\ y_i > y_j\Big)}{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} I\Big(\widehat{\pi}(\boldsymbol{x}_i) \neq \widehat{\pi}(\boldsymbol{x}_j),\ y_i > y_j\Big)}, \tag{4}$$

It is worth pointing out that ties in both the predictions and the responses are left out of the computation of the concordance probability. When there are no ties for the predictions, the concordance probability in the binary setting coincides with the well-known AUC. Note that the absense of ties in the predictions is necessary, since the AUC treats them as comparable pairs [2].

In this continuous setting, a thresholded concordance probability $C(\nu)$ can also be considered as introduced by Van Oirbeek et al. [3]. This measure does not consider nearly identical responses, by requiring that the responses of comparable observations differ at least ν from each other:

$$C(\nu) = \mathrm{P}\left(\pi(\boldsymbol{X}_i) > \pi(\boldsymbol{X}_j) \mid Y_i - Y_j > \nu\right). \tag{5}$$

This thresholded concordance probability is very useful in the context of severity models for insurance pricing as discussed by Ponnet et al. [4]. Such a severity model is used to predict the cost of an average claim. Hence, for evaluating such a severity model, we want to make sure that it can distinguish large from small risks. Moreover, distinguishing nearly identical claim sizes has little practical importance in this setting, which is taken into account by the thresholded concordance probability when $\nu > 0$. To estimate the concordance probability in continuous settings, we can use the natural estimator given by

$$\widehat{C}(\nu) = \frac{N_c}{N} = \frac{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} I\Big(\widehat{\pi}(\boldsymbol{x}_i) > \widehat{\pi}(\boldsymbol{x}_j),\ y_i - y_j > \nu\Big)}{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} I\Big(\widehat{\pi}(\boldsymbol{x}_i) \neq \widehat{\pi}(\boldsymbol{x}_j),\ y_i - y_j > \nu\Big)}, \tag{6}$$

Note that in this case, the number of concordant pairs N_c and the number of total pairs N only consider those pairs (i, j) for which $y_i - y_j > \nu$.

An extension of the thresholded concordance probability is obtained when each response-prediction pair i has a specific weight w_i. These weights can namely be considered in the weighted thresholded concordance probability, which is defined as follows:

$$\widehat{C}_w(\nu) = \frac{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} I\Big(\widehat{\pi}(\boldsymbol{x}_i) > \widehat{\pi}(\boldsymbol{x}_j),\ y_i - y_j > \nu\Big) w_i w_j}{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} I\Big(\widehat{\pi}(\boldsymbol{x}_i) \neq \widehat{\pi}(\boldsymbol{x}_j),\ y_i - y_j > \nu\Big) w_i w_j}. \tag{7}$$

The most straightforward way of calculating the different concordance probabilities of Equations (2), (4), (6) and (7), is by comparing all $n(n-1)/2$ pairs of observations, for example in a nested for-loop, and keep count of the concordant, discordant, and tied pairs. While this is easy to implement, it clearly comes with a computational complexity of $\mathcal{O}(n^2)$. This can become prohibitive for large data sets and when the concordance probability needs to be computed many times, for example when it is used in the fitness function of an evolutionary algorithm. In the currently available `R` packages, the estimation of the classical concordance probability is implemented in the naive way by a nested for-loop [5, 6, 7, 8, 9, 10]. In this paper we introduce an algorithm for computing the weighted thresholded concordance probability with time complexity $\mathcal{O}(n\log(n))$. We do this by adapting the well-known mergesort algorithm [11] for counting inversions in a vector. We take three steps. First, we make sure that ties in predictions are adequately dealt with. Second, we introduce an adjusted merge step of the algorithm to include a threshold on the responses of comparable pairs to compute the expression in Equation (6). Finally, we compute the expression in Equation (7) by introducing a second adjustment to the merge step to appropriately deal with weights.

A statistic related to the the concordance probability is Kendall's tau [12]. For Kendall's tau, the "naive" implementation also has a $\mathcal{O}(n^2)$ computational complexity. The `cor.fk` function of the `pcaPP` package [13] implements a faster algorithm which requires $\mathcal{O}(n\log(n))$ computational cost. The algorithm used for that function is similar in spirit to our algorithm in that it uses an adaptation of mergesort. However, it differs from ours in a number of ways. First, it cannot deal with custom weights. Second, it cannot deal with thresholds on the predictions. Finally, it counts ties in a different way and cannot readily be used to compute even the simplest version of the continuous concordance probability in case of ties.

Other packages having a fast implementation of Kendall's tau, such as the `copula` package [14], seem to rely on the implementation of `pcaPP`. In summary, there seem to be no fast implementations of comparable statistics allowing for the computation of the weighted and thresholded concordance probability.

In Section 2, we discuss how the (weighted) thresholded concordance probability can be computed in quasilinear time. The results of this section are confirmed by a simulation study which is the topic of Section 3. We apply the proposed algorithms on two real data applications in Section 4. Finally, the conclusion is given in Section 5.

2. Computing the concordance probability

In this section, we describe algorithms to compute the concordance probability in quasilinear time. We will start with the case of a binary response variable. Then, we will treat the concordance probability for a continuous response variable. Finally, we will show that the extension to the weighted thresholded concordance probability can also be incorporated into our algorithm.

2.1. Binary response

We have pointed out the equivalence between the popular AUC and C in case of a binary response variable. The AUC is often approximated based on the trapezium rule with $\mathcal{O}(n\log(n))$ computation time. For a binary response variable, it can also be computed exactly using the following procedure. First, the observations are ordered according to the values of the corresponding predictions (in ascending order), which takes $\mathcal{O}(n\log(n))$ time. Next, the crucial step is to see that every observation for which $Y = 0$ creates a number of concordant pairs, which is equal to the number of one observations on its right. Similarly, each observation for which $Y = 1$ creates a number of discordant pairs, which is equal to the number of zero observations on its right. The latter can be computed in $\mathcal{O}(n)$ time, such that we can conclude that the exact calculation of the AUC can be done in $\mathcal{O}(n\log(n))$ time. From here on, we call this the *binary algorithm*.

When there are ties however, the AUC and the C-index are no longer equivalent, yet we would still like to compute the latter very quickly. That is why several approximations for the concordance probability were introduced by Van Van Oirbeek et al. [3], both for the binary and the continuous setting. However, these are still approximations, and when using the concordance probability to compare the discriminatory ability of two models, one prefers the absence of a bias.

We now show that the exact procedure for calculating the AUC in the binary setting, can be adjusted to appropriately deal with ties to calculate $\hat{C}$ exactly and quickly. We first compute the AUC in $\mathcal{O}(n\log(n))$ time. Then, we adjust the resulting number for potential ties in the predictions as follows. We first count the number of unique ties in the predictions, in $\mathcal{O}(n)$ time. Next, on each tie in the predictions together with its corresponding response, the binary algorithm is applied again to obtain the number of concordant and discordant pairs that contain ties in the predictions. Proposition 1 below shows that this can once again be done in $\mathcal{O}(n\log(n))$ time. Once these numbers are known, the AUC can be corrected for these initially incorrect numbers of concordant and discordant pairs to obtain the concordance probability $\hat{C}$. Ties in the responses are not considered here, since this is not relevant in the binary setting.

Proposition 1. *Let $\widehat{\boldsymbol{\pi}} = \big(\widehat{\pi}(x_1), \ldots, \widehat{\pi}(x_n)\big)$ be a vector of predictions and $y = (y_1, \ldots, y_n)$ a vector of binary responses. We can can compute the concordance probability of Equation* (2) *in $\mathcal{O}(n \log(n))$ time by adjusting the AUC for the presence of ties.*

Proof. We start of by calculating the AUC in $\mathcal{O}(n \log(n))$ time using the sorting procedure outlined above. This yields the number of concordant and discordant pairs but does not discard the pairs for which the predictions are tied.

Suppose we have k ties in the vector of n predictions $\widehat{\boldsymbol{\pi}}$, and each tie j appears n_j times. Furthermore, we consider unique observations as ties of multiplicity 1, such that $n = \sum_{j=1}^{k} n_j$. Additionally, denote $c_j = \frac{n_j}{n}$ so that. $\sum_{j=1}^{k} c_j = 1$ and $c_j > 0$ for all $j = 1, \ldots, n$. Consider now a tie j in the predictions of the observations with indices $i_1, i_2, \ldots, i_{n_j}$, i.e. $\widehat{\pi}(x_{i_1}) = \widehat{\pi}(x_{i_2}) = \ldots = \widehat{\pi}(x_{i_{n_j}})$. In order to correct the computed AUC for this tie, we recompute the number of concordant and discordant pairs on these observations and subtract them from the total numbers computed on $\widehat{\boldsymbol{\pi}}$. We repeat this procedure for all ties j for which $n_j > 1$.
This means that for each tie j, we have to perform an algorithm of $\mathcal{O}(n_j \log(n_j))$ time, such that the total computation time has a time complexity $\mathcal{O}(\sum_{j=1}^{k} n_j \log(n_j))$. The total complexity to correct for all ties is thus given by:

$$\begin{aligned}
\mathcal{O}\left(\sum_{j=1}^{k} n_j \log(n_j)\right) &= \mathcal{O}\left(\sum_{j=1}^{k} c_j n \log(c_j n)\right) \\
&= \mathcal{O}\left(\sum_{j=1}^{k} c_j n \big(\log(c_j) + \log(n)\big)\right) \\
&\leq \mathcal{O}\left(\sum_{j=1}^{k} c_j n \log(n)\right) \\
&= \mathcal{O}\left(n \log(n) \sum_{j=1}^{k} c_j\right) \\
&= \mathcal{O}\left(n \log(n)\right)
\end{aligned}$$

Hence, the overall time complexity remains $\mathcal{O}(n \log(n))$. □

2.2. Continuous response

For a continuous response variable, we no longer have two groups in the observations, which is why the AUC and the previously discussed algorithm for $\hat{C}$ cannot be used. We propose a new algorithm to calculate $\hat{C}$ exactly and quickly in this setting. Once again, the first step consists of ordering the observations based on their predictions (from small to large). Next, the number of discordant pairs necessary for the concordance probability, equals the number of inversions in the responses. An inversion in a vector a is defined as a pair of elements $a[i] > a[j]$ for which $i < j$. Similarly, the number of concordant pairs necessary for the concordance probability, equals the number of non-inversions in the responses. A non-inversion in a vector a is defined as a pair of elements $a[i] < a[j]$ for which $i < j$. Counting the number of (non)-inversions in an array, is something that can be done in $\mathcal{O}(n \log(n))$, based on the mergesort algorithm [11, 15, 16].

Proposition 2. *Let $\widehat{\boldsymbol{\pi}} = \big(\widehat{\pi}(x_1), \ldots, \widehat{\pi}(x_n)\big)$ be a vector of predictions and $y = (y_1, \ldots, y_n)$ a vector of continuous responses. Using the merge procedure of Algorithm 1, we can can compute the concordance probability of Equation* (4) *in $\mathcal{O}(n \log(n))$ time.*

Proof. The classical mergesort algorithm is a standard example of a divide-and-conquer recurrence. The divide step divides the sequence to be sorted in two subsequences of equal length. The conquer step then uses mergesort on each of the two subsequences. Finally, the combine step merges the sorted subsequences in a single sorted sequence. This algorithm requires $\mathcal{O}(n \log(n))$ computation time as a result of the merge step taking $\mathcal{O}(n)$ time. More precisely, assume w.l.o.g. that $n = 2^k$. The computation time $T(n)$ of the classical mergesort algorithm is given by the recurrence relation

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

where the first term results from the splitting of the sample in 2 and the latter from the merging step. More generally, we have that

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

implies $T(n) = n \log(n)$ through the master theorem for divide-and-conquer recurrences.

Using this algorithm, counting the number of concordant and discordant pairs can also be done simultaneously, once the observations are ordered based on their predictions (from small to large). To this end, only the merge function of the mergesort algorithm needs to be adapted, as can be seen in the pseudo-code of Algorithm 1 in Appendix A, where we suppose a 0-based indexing. More specifically, suppose we want to merge the sorted left array x of length n with the sorted right array y of length m in a sorted way. Therefore, we are comparing the i^{th} element of x with the j^{th} element of y. When $x[i]$ is smaller than $y[j]$, we know that it will also be smaller than all other elements of y on the right of $y[j]$, since y is a sorted array. Hence, because of x being the left array and y the right array, we know that $x[i]$ creates $m - j$ concordant pairs. On the other hand, when $y[j]$ is smaller than $x[i]$, we know that it will also be smaller than all other elements of x on the right of $x[i]$, since x is a sorted array. Hence, because of y being the right array and x the left array, we know that $y[j]$ creates $n - i$ discordant pairs. Finally, when $x[i]$ is equal to $y[j]$, we observed a tie in the responses. In that case, we have to find the smallest index l, larger than j, such that $y[l]$ is strictly larger than $y[j]$. If this index l does not exist, we know that $x[i]$ creates $m - j$ ties. Otherwise, it creates $l - j$ ties and $m - l$ concordant pairs.

As a result, while merging the left and right array in the mergesort algorithm, we can keep track of the number of ties in the responses, as well as the number of concordant and discordant pairs without changing the time complexity of $\mathcal{O}(n \log(n))$.

One way of correcting for the presence of ties in the predictions is using the same procedure as that of Proposition 1. This does not change the time complexity of the overall algorithm, which remains $\mathcal{O}(n \log(n))$. □

2.3. The weighted thresholded concordance probability

We start by addressing the thresholded concordance probability and include the weighted version afterwards.

The thresholded concordance probability only considers observations with responses that differ at least a value ν from each other. Hence, two responses are considered as being a tie when they differ exactly ν from each other. For a naive algorithm inspecting all pairs of predictions, it is trivial to incorporate this threshold ν, but this would still yield a computational complexity of $\mathcal{O}(n^2)$. Instead, we have modified the merge-step of the mergesort-based algorithm of Section 2.2 to be able to incorporate this feature without increasing the computational complexity. As a result, we can compute the thresholded concordance probability in $\mathcal{O}(n \log(n))$ time, which is stated and proven in Proposition 3 below.

Proposition 3. *Let* $\widehat{\boldsymbol{\pi}} = \big(\widehat{\pi}(x_1), \ldots, \widehat{\pi}(x_n)\big)$ *be a vector of predictions and* $y = (y_1, \ldots, y_n)$ *a vector of continuous responses. Using the merge procedure of Algorithm 2, we can can compute the thresholded concordance probability of Equation* (6) *in* $\mathcal{O}(n \log(n))$ *time.*

Proof. We know already from Proposition 2 that the concordance probability $C(0)$ can be estimated in $\mathcal{O}(n \log(n))$ time. In order to introduce a threshold on the difference in responses for comparable pairs, we need to only change the merge step of the previous algorithm. It is thus sufficient to make sure that the adjusted merging step requires $\mathcal{O}(n)$ time. The pseudo-code of Algorithm 2 in Appendix A, where we suppose a 0-based indexing, shows the required adjustments to the merge function.

More specifically, suppose we want to merge the sorted left array x of length n with the sorted right array y of length m to obtain a single sorted array. In doing so, we are comparing the i^{th} element of x with the j^{th} element of y. When $x[i]$ is smaller than $y[j]$, we know that it will also be smaller than all other elements of y on the right of $y[j]$, since y is a sorted array. Hence, because of x being the left array and y the right array, $x[i]$ can create concordant pairs. However, since a tie in the responses is defined as two responses that differ exactly a value ν from each other, it can also create a number of ties. To define this number of ties and concordant pairs, we iterate over all elements after $y[j]$ until the first one strictly larger than $x[i] + \nu$ is found and call the index of this element j_2. While doing so, we also keep track of the number of elements that form a tie with $x[i]$. Hence, the total number of ties can be updated by adding the latter and moreover, $m - j_2$ concordant pairs are created by $x[i]$. On the other hand, when $y[j]$ is strictly smaller than $x[i]$, we know that it will also be smaller than all other elements of x on the right of $x[i]$, since x is a sorted array. Hence, because of y being the right array and x the left array, $y[j]$ can create discordant pairs. However, since a tie in the responses is defined as two responses that differ exactly a value ν from each other, it can also create a number of ties. To define this number of ties and discordant pairs, we iterate over all elements after $x[i]$ until the first one strictly larger than $y[j] + \nu$ is found and call the index of this element i_2. While doing so, we also keep track of the number of elements that form a tie with $y[j]$. Hence, the total number of ties can be updated by adding the latter and moreover, $n - i_2$ discordant pairs are created by $y[j]$.

Note that i_2 (j_2) is updated each time i (j) is updated by iterating over the following larger elements in x (y). Hence, keeping track of i_2 (j_2) requires $\mathcal{O}(n)$ time. As a result, while merging the left and right array in the mergesort algorithm, we can keep track of the number of ties in the responses, as well as the number of concordant and discordant pairs without changing the time complexity of $\mathcal{O}(n \log(n))$. The correction for the presence of ties in the predictions does not change this time complexity, as explained in the proof of Proposition 1. □

Finally, we address the inclusion of weights. The calculation of $\widehat{C}_w(\nu)$ would again be trivial in a naive implementation inspecting all the pairs. By making additional adjustments to the merge-step, we have extended the mergesort-based algorithm to be able to calculate the weighted thresholded concordance probability in linearithmic time. This is stated and proven in Proposition 4 below, and the pseudo-code of the adjusted merge-step can be found in Algorithm 3 of Appendix A.

Proposition 4. *Let $\widehat{\boldsymbol{\pi}} = \big(\widehat{\pi}(x_1), \ldots, \widehat{\pi}(x_n)\big)$ be a vector of predictions, $y = (y_1, \ldots, y_n)$ a vector of continuous responses and $w = (w_1, \ldots, w_n)$ a vector of weights. Using the merge procedure of Algorithm 3, we can can compute the concordance probability of Equation (7) in $\mathcal{O}(n \log(n))$ time.*

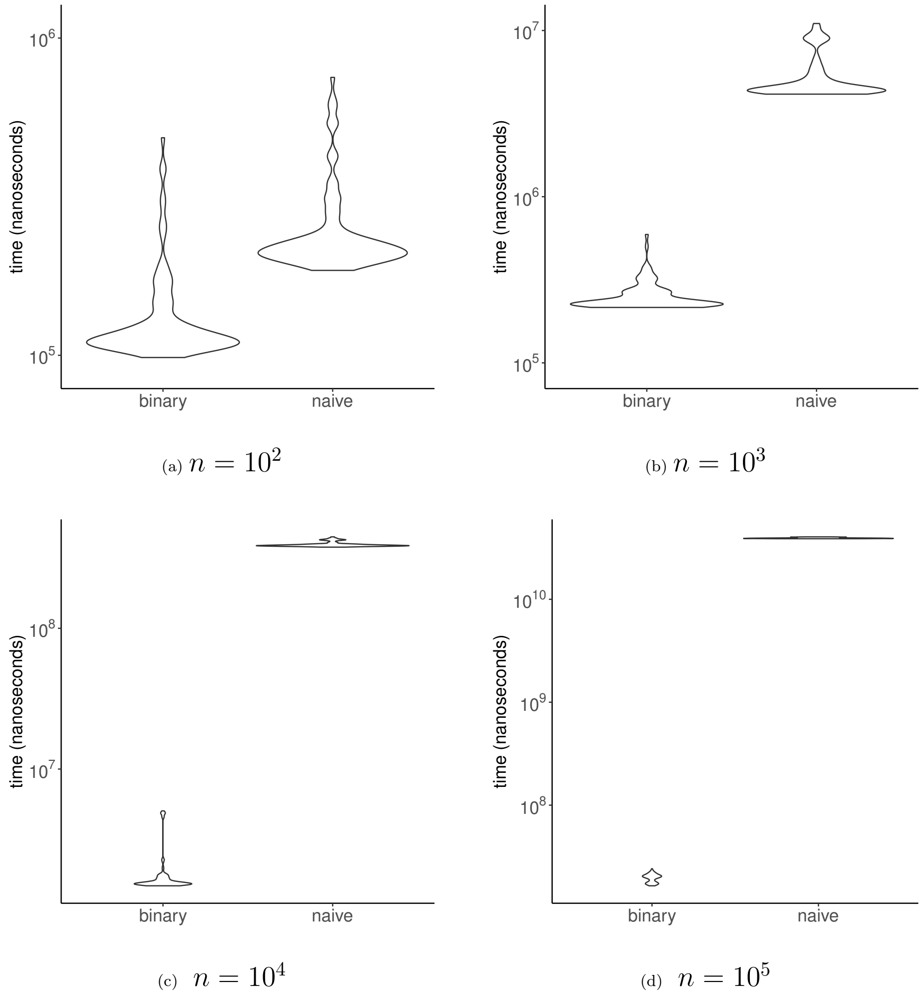
Proof. We know already from Proposition 3 that the thresholded concordance probability $C(\nu)$ can be estimated in $\mathcal{O}(n \log(n))$ time. As can be seen in the pseudo-code of Algorithm 3 in Appendix A, an adaptation is necessary in the merging step used to estimate $C(\nu)$. Building further on the notations introduced in the Proof of Proposition 3, we first define w_x (w_y) as the weights corresponding to the elements in x (y). The sum of all elements in w_y is the initial value of the variable s_y. Each time that the index j_2 is updated, we also update s_y by subtracting $w_y[j_2]$ of it. In other words, s_y represents the sum of the weights of all responses in y from index j_2 on. Hence, when $x[i]$ is smaller than $y[j]$, the weighted number of concordant pairs is increased with $w_x[i]s_y$. Remember that while updating j_2, we also kept track of the number of ties with $x[i]$ in y. This time, we will keep track of the sum of the weights of the elements in y that form a tie with $x[i]$, defined by t_y. Hence, the weighted number of ties introduced by $x[i]$ equals $w_x[i]t_y$.

Analogously, initialize s_x by the sum of all elements in (w_x) and update its value when i_2 is updated, by subtracting $w_x[i_2]$ of it. When $y[j]$ is then strictly smaller than $x[i]$, the weighted number of discordant pairs is updated by adding $w_y[j]s_x$. Similarly, t_x represents the sum of the weights of the elements in x that form a tie with $y[j]$. Consequently, the weighted number of ties introduced by $y[j]$ equals $w_y[j]t_x$.

Note that t_x and s_x (t_y and s_y) are updated each time i_2 (j_2) is updated. Hence, keeping track of these values requires $\mathcal{O}(n)$ time. As a result, while merging the left and right array in the mergesort algorithm, we keep track of the weighted number of ties in the responses, as well as the weighted number of concordant and discordant pairs without changing the time complexity of $\mathcal{O}(n \log(n))$. The correction for the presence of ties in the predictions does not change this time complexity, as explained in the proof of Proposition 1. □

3. Simulation study

In this section, we investigate the computational cost to estimate the concordance probability in a simulation study. The performance is measured by the run times in `R` 4.1.1, on a computer with processor `Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz 2.11 GHz`. First, we focus on the binary setting and simulate n values from a binomial distribution with size 1 and success probability 0.5. These values are considered as the responses y in Equation (2). Similarly, we take n random samples from a uniform distribution between 0 and 1, that serve as the predictions. On these n pairs of responses and predictions, the concordance probability is calculated by both the naive and the binary algorithm as discussed in Section 2. We repeat this process 100 times and store the computation times. For each sample size $n \in \{10^2, 10^3, 10^4, 10^5\}$, the results are shown in Figure 1. It is clear that the binary algorithm estimates the concordance probability faster than the naive one. For $n = 10^2$, the proposed algorithm is about a factor 2 faster than the naive algorithm. For larger sample sizes, however, the difference between the computation times of both approaches is much larger. For $n = 10^5$ for example, the computation times of the naive and proposed methods differ by a factor 10^4.



(a) $n = 10^2$

(b) $n = 10^3$

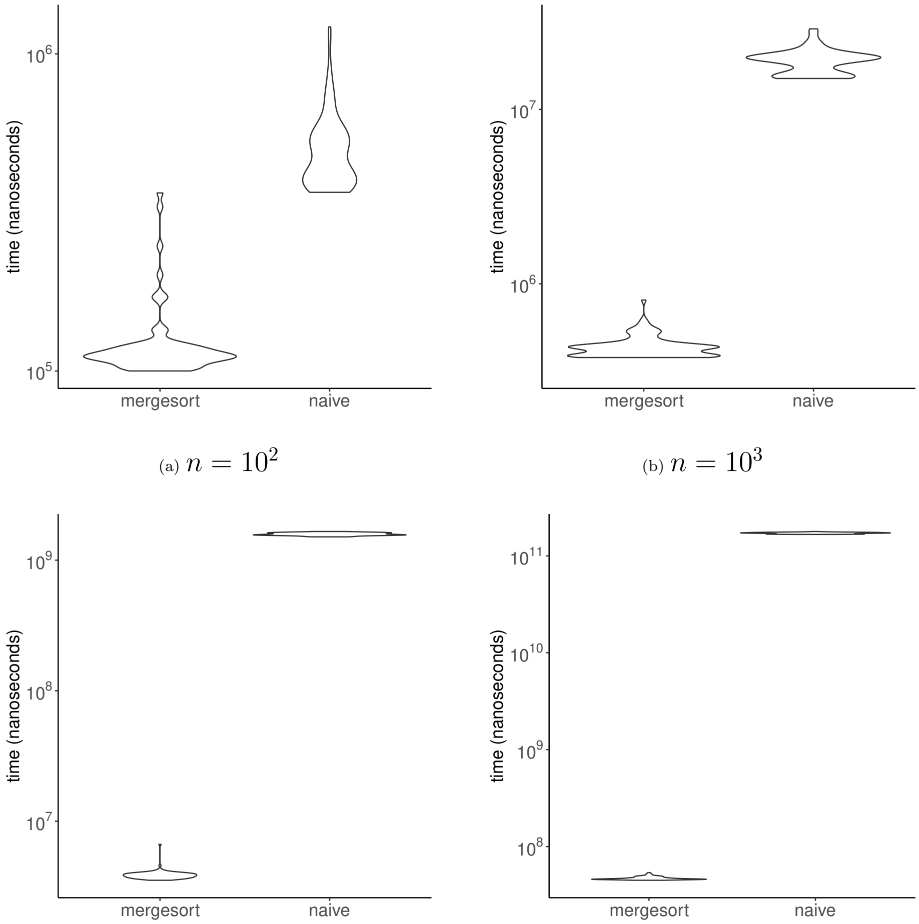
(c) $n = 10^4$

(d) $n = 10^5$

Figure 1: Computation times of the naive and the binary algorithm to estimate the concordance probability in the binary setting for different sample sizes n.

A similar simulation study for the continuous setting is set up, where n responses are sampled from a standard normal distribution. The corresponding predictions equal the responses with an error term, which is also sampled from the standard normal distribution. For these n pairs of responses and predictions, the concordance probability is calculated by both the naive and the mergesort-based algorithm as discussed in Section 2. Once again, we define pairs with the same prediction as incomparable and moreover, we set ν on zero. The entire procedure is repeated 100 times and for each sample size $n \in \{10^2, 10^3, 10^4, 10^5\}$, the computation times are shown in Figure 2.

Just as in the binary setting, the naive way to estimate the concordance probability is much slower than the one based on the mergesort algorithm. The difference is even more pronounced however. For $n = 100$ we obtain a difference in computation times of a factor 4. For $n = 10^5$, we see once more that the fast algorithm has a computation time that is about 10^4 times as small as the naive algorithm. Finally, we also numerically verified



(a) $n = 10^2$

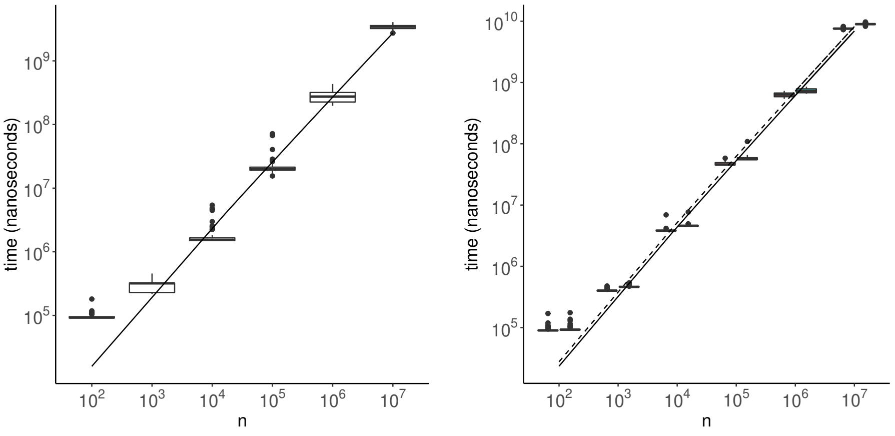
(b) $n = 10^3$

(c) $n = 10^4$

(d) $n = 10^5$

Figure 2: Computation times of the naive and the mergesort-based algorithm to estimate the concordance probability in the continuous setting for different sample sizes n.

Propositions 1, 2 and 3 that state that the (thresholded) concordance probability can be estimated in linearithmic time in the binary and continuous setting. For both settings, we create the responses and the predictions in the same way as before. Moreover, we also consider the extreme sample sizes 10^6 and 10^7. Finally, in the continuous setting, we estimate both C and $C(0.5)$. We consider each discussed situation 100 times and keep track of the computation times, which are shown in Figure 3. The straight line in both sub figures clearly confirms that the computation time for the concordance probability is in both settings of size $\mathcal{O}(n\log(n))$. The fact that the computation times for the sample size 100 seem slightly too high, can be explained by computational overhead which plays a diminishing role for larger sample sizes.



(a) Binary response.

(b) Continuous response with $\nu = 0$ (left) and $\nu = 0.5$ (right).

Figure 3: Computation times of the binary and the mergesort-based algorithm of the concordance probability for different sample sizes n. The (dashed) line is an estimate of the $\sim n\log(n)$ expected computation time (when $\nu = 0.5$).

4. Real data applications

In this section, we illustrate two potential applications of the mergesort-based algorithm to compute the (thresholded) (weighted) concordance probability. The first one focuses on feature selection based on the concordance probability, whereas the second one focuses on the parameter estimation of a model such that the concordance probability is optimized. Both examples focus on a real data set from the insurance sector.

4.1. Feature selection

For the first example, we focus on the data set `pg16trainpol` from the `R`-package `CASdatasets` on CRAN [17]. This data set was used during the pricing game of the French institute of Actuaries in 2016. It contains 87,226 policies for private motor insurance of which their exposure as well as their number of claims is given. Most of the policies experienced no claim, 4.5% had one claim and only 0.3% had two or more claims. Moreover, there are 13 other variables that were also considered and discussed in detail by Ponnet et al. [4], e.g. the geographical area and the vehicle power. These variables are all categorized and their bar plots are shown in Figure B.1 in Appendix B, together with their interpretations. Note that the majority of categorical levels are unknown due to the fact that they have been anonymized for confidentiality reasons. Due to the anonymization, we cannot fully interpret the final model.

The goal is to find, out of these 13 variables, the most relevant predictors to estimate the number of claims. Instead of considering each of the 8,192 possible models by an exhaustive search, we use a faster binary particle swarm optimization (BPSO) algorithm that is introduced by Kennedy and Eberhart [18] and further discussed in detail by Qasim and Algamal [19], Mirjalili and Lewis [20], Khare and Rangnekar [21]. Intuitively, each particle of the swarm is a binary vector of length 13, in which each element represents whether the corresponding predictor variable is selected (1) or not (0). For each particle, the number of claims is estimated by a Poisson model with the corresponding selected variables $\tilde{\boldsymbol{x}}$ as predictors and an offset equal to the logarithm of the exposure. This is a standard approach in insurance claims modeling [22], where we denote the predicted number of claims for observation i by $\widehat{\pi}(\tilde{\boldsymbol{x}}_i)$. We keep track of the model that resulted in the highest value for the fitness function for each particle separately. Moreover, we also record the global best model, which is the model resulting in the highest value for the fitness function over all particles in the swarm. Finally, each particle is updated based on its local and the global best model. This entire procedure is repeated until convergence. The specific settings for running the BPSO algorithm are as follows. We work with a population size of 30 and a maximum number of iterations of 100. The population is generated randomly with each entry of every particle following a Bernoulli distribution with success probability 0.5. The algorithm stops when no improvement has been found for 5 iterations. The mutation operator is the one of [18]: for each entry of the particle is a Bernoulli random variable with success probability determined by a sigmoid transformation of the current velocity of that particle entry. The sigmoid ensures that the probability lies between 0 and 1.

We consider the following fitness function:

$$f(\lambda, \tilde{\boldsymbol{x}}) = \widehat{C}_{w,0,1+}(\tilde{\boldsymbol{x}}) - \lambda \|\boldsymbol{\beta}\|_1,$$

with

$$\widehat{C}_{w,0,1+}(\tilde{\boldsymbol{x}}) = \frac{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_i w_j I\Big(\widehat{\pi}(\tilde{\boldsymbol{x}}_i) > \widehat{\pi}(\tilde{\boldsymbol{x}}_j),\ y_i \geq 1,\ y_j = 0\Big)}{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_i w_j I\Big(\widehat{\pi}(\tilde{\boldsymbol{x}}_i) \neq \widehat{\pi}(\tilde{\boldsymbol{x}}_j),\ y_i \geq 1,\ y_j = 0\Big)},$$

where the weight w_i corresponds to the exposure of claim i. Hence, policies with a smaller duration receive a smaller weight. Furthermore, $\widehat{C}_{w,0,1+}(\tilde{\boldsymbol{x}})$ represents the ability of the model to discriminate policies that encountered at least one accident from policies that did not encounter accidents. With $\lambda \in \mathbb{R}$ the tuning parameter and $\boldsymbol{\beta}$ the model coefficients (without intercept), we can see $\lambda\|\boldsymbol{\beta}\|_1$ as a sparsity penalty. More specifically, it is the lasso penalty introduced by Tibshirani [23] which shrinks the coefficients towards zero.

We randomly split 70% of the data set into a training set, 15% into a validation set and the remaining 15% forms a test set. On this training set, we ran the BPSO algorithm for 50 equally spaced values of λ in the range [0, 0.2]. Hence, we obtained 50 possible models to predict the number of claims. For each model, the concordance probability $\widehat{C}_{w,0,1+}(\tilde{\boldsymbol{x}})$ is calculated on the validation set. Figure 4 shows the maximal value of $\widehat{C}_{w,0,1+}(\tilde{\boldsymbol{x}})$ in function of the number of predictors selected in the model. It shows us that the full model obtains a similar concordance probability compared to the one where only 6 predictors are selected: `FleetMgt`, `FleetSizeCateg`, `PayFreq`, `VehiclAge`, `Deduc` and `VehiclPower`. This model has a concordance probability of 66.88% on the test set, which is only 0.36% smaller than the one of the full model. We thus obtain a model with less than half of the predictors and with virtually identical out-of-sample performance.

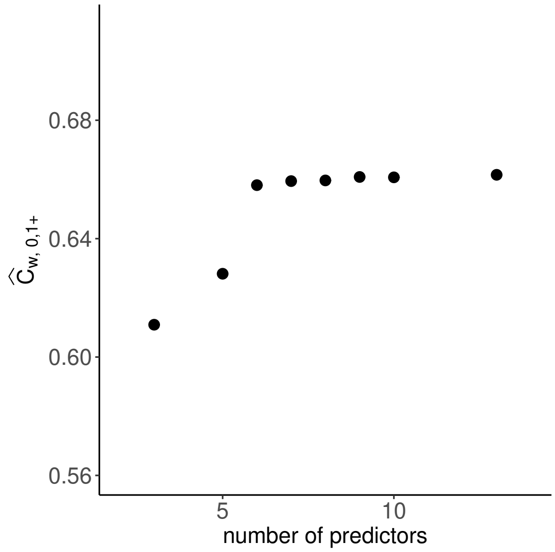


Figure 4: The maximal value for $\widehat{C}_{w,0,1+}$ in function of the number of predictors in the model to predict the number of claims.

4.2. Model training on the concordance probability

Rather than using the concordance probability for feature selection, we can also consider training directly on the concordance probability, i.e. using it as a loss function. This is not trivial, since many regression estimators require the loss-function to be differentiable. This is why differentiable approximations to the concordance probability have been proposed [24, 25, 26, 27], which can then be used in boosting algorithms such as XGBoost or LightGBM and in neural networks [28, 29].

Alternatively, one can take the approach of trying to optimize a non-differentiable loss/fitness function based on the concordance probability. In that case, one has to resort to algorithms that can deal with this non-differentiability. Examples of those are the particle swarm algorithm of the previous section, ant colony optimization, genetic algorithms and Cuckoo search [30, 31, 32, 33]. This approach was also taken in Kalderstam et al. [34]. Any of these approaches require evaluating the concordance probability a very high number of times, and this represents most of the computational cost of the optimization process. As a result, a fast computation of the concordance probability is highly beneficial for this purpose.

We illustrate this on the data of the 2015 pricing game of the French institute of Actuaries, which is publicly available under the name `pg15training` in the `R`-package `CASdatasets` on CRAN. Our goal is to predict the number of claims made by the insured based on six variables, three continuous and three categorical. The three continuous variables are the driver's age (`Age`), the population density in the city that the driver of the car lives in (`Density`) and the car value (`Value`). The three categorical variables are the category of the car (`Category`), the type of the car (`Type`) and the job of the car owner (`Occupation`). These variables have three, six and five categories respectively, and were one-hot encoded before training the models on 70% of the data. The remaining 30% of the data form the test set. Note that more than 70% of the policies had an exposure of 1 and for the other variables, the histograms and bar plots are shown in Figure B.2 in Appendix B.

We will model the response Y by

$$\mathrm{E}[Y|\boldsymbol{X} = \boldsymbol{x}] = e^{f(\boldsymbol{x})+c}$$

where c is an offset given by the logarithm of the exposure, and f is a neural network with a single hidden layer of size 2 and a ReLU activation function. Figure 5 shows the skeleton of the neural network for a single random initialization of the weights.

Typically, this model would be fit using backpropagation on the deviance corresponding with conditional Poisson models. We would like to directly optimize the prediction model for the concordance probability $\widehat{C}_{w,0,1+}$. Therefore, we cannot use backpropagation, and we use a genetic algorithm instead. In particular, we use the `ga` function of the `R`-package `ga` [35]. The specific setup of this genetic algorithm is as follows. The population size is set at 30. Instead of using purely random starts for the genetic algorithm, we use the initial weights from training with a conditional Poisson distribution, with offset equal to the logarithm of the exposure. This initial training step was done using the `R`-package `h2o` [36]. In addition to these initial weights, we add 29 uniformly random generated starting values. The maximum number of iterations is fixed at 100. We used a local arithmetic crossover, with uniform mutation as the mutation operator. Finally, the selection procedure we used is proportional selection after linear scaling of the fitness values.

As both the initial weights for training the neural network and the genetic algorithm use randomness, we repeat the procedure 100 times, starting from a new random initialization of the neural network. The purpose of repeating the whole procedure 100 times, is to eliminate the possibility that the performance of the genetic algorithm is due to a poor or lucky random initialization of the neural net.

To illustrate the potential of our approach, we compare the performance of the classical neural network optimized for the deviance corresponding with conditional Poisson models, with the performance of the neural network optimized for the concordance probability with the genetic algorithm. Figure 6 shows the performance of the procedures over 100 random initializations of the neural network. In the left panel, we see the performance on training and test data for the classical neural network, as well as the neural network trained with the genetic algorithm. We clearly see that both on training and on test data, the genetic algorithm yields substantial improvements in the concordance probability of the resulting model. The right panel of the same figure shows the relative performance improvement. On test data, the genetic algorithm yields up to 10% improvement in the concordance probability over the classical neural network, and never worsens the out-of-sample concordance probability. The obtained outperformance is strongly significant with p-values of close to zero for, e.g., a Wilcoxon rank-sum test. It is clear that directly optimizing for the concordance probability can be very beneficial if this is the metric of interest. A fast computation of the concordance probability is indeed vital, as the many evaluations of the fitness function make the naive approach computationally prohibitive.

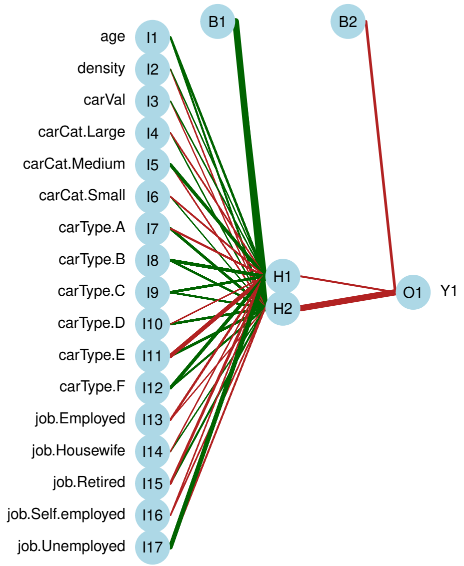


Figure 5: The layout of the neural network. Green indicates positive connections, whereas red indicates negative connections. The thickness of the connections correspond with the absolute size of the coefficients.

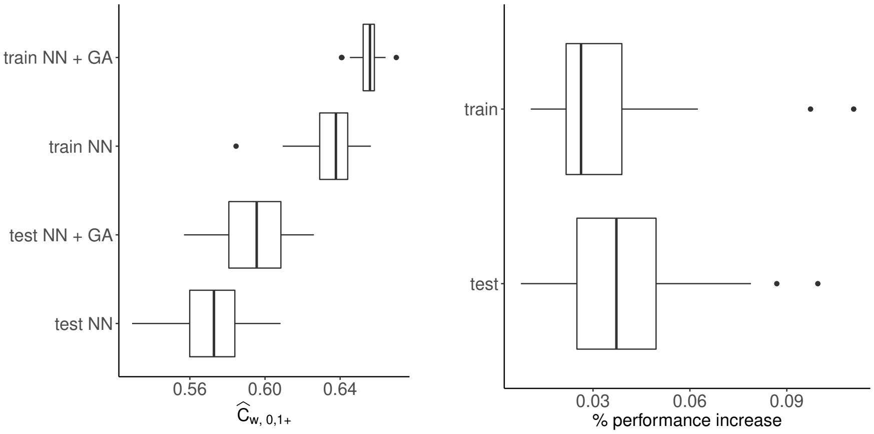


Figure 6: Performance comparison between classical neural network and a neural network trained with a genetic algorithm optimizing directly for the concordance probability is shown in the left panel. The performance increase of the genetic algorithm relative to the classical neural network is shown in the bottom panel.

5. Conclusion

In this article we addressed the efficient computation of the weighted thresholded concordance probability. This measure represents the probability of having a concordant pair of responses with their predictions. The measure optionally incorporates a threshold ν which excludes similar observations. It also has the ability of including weights, so that each pair of a response with its prediction receives a specific weight while calculating the concordance probability. In the currently available algorithms, the concordance probability is estimated in a quadratic run time. Since this is problematic for large data sets, we propose a new algorithm based on the mergesort algorithm. This computes the weighted thresholded concordance probability in linearithmic time, which is both proven and empirically confirmed.

This faster algorithm unlocks various new applications. We have illustrated two such use cases on the problem of insurance claims modeling. The first uses the weighted thresholded concordance probability to perform variable selection. The second uses it in an objective function of a neural network. The applicability of our methods extends beyond these illustrations, however. As the C-index measures the quality of the ranking of the predictions, rather than the precise values of the predictions, the C-index can be of interest any time the main goal is to rank items rather than predict their values. There are many situations in which ranking is the main goal, and we discuss three prominent ones here. An important one is in the context of recommender systems. Examples of those are Netflix suggesting movies to its users or Kindle suggesting new books to buy. In this setting, the interest is primarily in which movie or book the user would prefer out of all possible movies or books, rather than how much the user would like the proposed movie or book. Another application is in information retrieval, where a query of a user prompts the search for the documents best matching the query. Again, the focus is not on the precise quality on the match, but rather on the ranking of the different documents. A final example is that of portfolio asset allocation. In a long-short equity portfolio, the portfolio manager is primarily interested in the ranking of the performances of the equities under considerations rather than their exact performance. In particular, the top ranks and bottom ranks determine the portfolio, irrespective of their precise performance. For each of the previously mentioned examples, there are scenarios where the number of observations (e.g. films to recommend, or files to retrieve) is very large. In that case, it is useful and potentially required that the measure of the quality of the ranking is easy to compute. Therefore, we believe that there are many practical scenarios in which the fast thresholded and weighted concordance probability can prove useful. It is worth noting that the weights and thresholds add additional flexibility with practical relevance. For example, the weights can give a subset of observations (clients, document types, equity classes, etc.) a higher importance in the ranking, prompting the model to prioritize ranking these items accurately. Additionally, the threshold allows for ignoring the quality of the rankings of items that are very close to one another. For example, if two books will be received almost equally well or two assets will perform nearly the same, it may not be as important to rank them correctly. Instead, the model could focus on correctly ranking those books which will be received very differently or those assets which will perform very differently, since getting such a ranking wrong would have a much larger impact on the system.

In addition to potential applications in ranking challenges, a faster algorithm also the benefit that it allows for easier bootstrap-based inference on the weighted thresholded concordance probability. This is also a computationally demanding task where speed of calculation is of vital importance. The alternative which is typically faster, but requires more strict assumptions, is to work with an asymptotic expression for the variance of the C-index. A basic one, which can be derived from Kendall's tau, is given by $\frac{2n+5}{18n(n-1)}$. An alternative and more generally applicable expression, derived from the results in [37, 38], is given by $\frac{\sum_{i=1}^{n}(N_{c,i}-N_{d,i})^2-2n(n-1)}{4n(n-1)(n-2)(n-3)}$, where $N_{c,i}$ and $N_{d,i}$ denote the number of concordant and discordant pairs associated with observation i. This last expression can also be computed in $\mathcal{O}(n\log(n)$ time.

All the algorithms discussed in this paper are written in `R` and `C++` and are available in the `R`-package `fastConcProb` on `github`.

Appendix A. Algorithms

Algorithm 1 merge

```
Input: $y$, $left$, $middle$, $right$, $invCount$, $concCount$, $tiesCount$
Output: Sorted $y$ from index $left$ until index $right$, together with updated values for
  $invCount$, $concCount$ and $tiesCount$
  $Left \leftarrow y[left, middle]$, $Right \leftarrow y[middle + 1, right]$
  $n \leftarrow$ length $Left$, $m \leftarrow$ length $Right$
  $i \leftarrow$ index in $Left$, initialized at 0
  $j \leftarrow$ index in $Right$, initialized at 0
  $k \leftarrow$ index in $y$, initialized at $left$
  while $i < n$ or $j < m$ do
    if $Left[i] < Right[j]$ then
      $concCount \leftarrow concCount + (m - j)$
      $y[k] \leftarrow Left[i]$
      $k \leftarrow k + 1$
      $i \leftarrow i + 1$
    else if $Left[i] > Right[j]$ then
      $invCount \leftarrow invCount + (n - i)$
      $y[k] \leftarrow Right[j]$
      $k \leftarrow k + 1$
      $j \leftarrow j + 1$
    else
      $l \leftarrow j$
      while $l < m$ and $Right[l] == Left[i]$ do
        $l \leftarrow l + 1$
      end while
      $tiesCount \leftarrow tiesCount + (l - j)$
      $concCount \leftarrow concCount + (m - l)$
      $y[k] \leftarrow Left[i]$
      $k \leftarrow k + 1$
      $i \leftarrow i + 1$
    end if
    while $i < n$ do
        $y[k] \leftarrow Left[i]$
        $k \leftarrow k + 1$
        $i \leftarrow i + 1$
    end while
    while $j < m$ do
        $y[k] \leftarrow Right[j]$
        $k \leftarrow k + 1$
        $j \leftarrow j + 1$
    end while
end while
```

Algorithm 2 merge with $\nu \geq 0$

```
Input: $y$, $\nu$, $left$, $middle$, $right$, $invCount$, $concCount$, $tiesCount$
Output: Sorted $y$ from index $left$ until index $right$, together with updated values for
    $invCount$, $concCount$ and $tiesCount$

$Left \leftarrow y[left, middle]$, $Right \leftarrow y[middle + 1, right]$

$n \leftarrow$ length $Left$, $m \leftarrow$ length $Right$
$i \leftarrow$ index in $Left$, initialized at 0
$j \leftarrow$ index in $Right$, initialized at 0
$k \leftarrow$ index in $y$, initialized at $left$
$i_2 \leftarrow$ index in $Left$ pointing to smallest element that is at least $\nu$ larger than $Right[j]$,
initialized at 0
$j_2 \leftarrow$index in $Right$ pointing to smallest element that is at least $\nu$ larger than $Left[i]$,
initialized at 0
$tiesCountLeft \leftarrow 0$, $tiesCountRight \leftarrow 0$

while $i < n$ or $j < m$ do
    if $i_2 < n$ and $Left[i_2] \leq Right[j] + \nu$ then
        if $Left[i_2] - \nu == Right[j]$ then
            $tiesCountLeft \leftarrow tiesCountLeft + 1$
        end if
        $i_2 \leftarrow i_2 + 1$
    else if $j_2 < m$ and $Right[j_2] \leq Left[i] + \nu$ then
        if $Right[j_2] - \nu == Left[i]$ then
            $tiesCountRight \leftarrow tiesCountRight + 1$
        end if
        $j_2 \leftarrow j_2 + 1$
```

```
  else if $Left[i] < Right[j]$ then
    $concCount \leftarrow concCount + (m - j_2)$
    $tiesCount \leftarrow tiesCount + tiesCountRight$
    $y[k] \leftarrow Left[i]$
    $k \leftarrow k + 1$
    $i \leftarrow i + 1$
    if $(i < n)$ and $(Left[i]! = Left[i-1])$ then
      $tiesCountRight \leftarrow 0$
    end if
  else
    $invCount \leftarrow invCount + (n - i_2)$
    if $\nu > 0$ then
      $tiesCount \leftarrow tiesCount + tiesCountLeft$
    end if
    $y[k] \leftarrow Right[j]$
    $k \leftarrow k + 1$
    $j \leftarrow j + 1$
    if $(j < m)$ and $(Right[j]! = Right[j-1])$ then
      $tiesCountLeft \leftarrow 0$
    end if
  end if
  while $i < n$ do
    $y[k] \leftarrow Left[i]$
    $k \leftarrow k + 1$
    $i \leftarrow i + 1$
  end while
  while $j < m$ do
    $y[k] \leftarrow Right[j]$
    $k \leftarrow k + 1$
    $j \leftarrow j + 1$
  end while
end while
```

Algorithm 3 merge with $\nu \geq 0$ and weights

```
Input: $y$, $\nu$, $w$, $left$, $middle$, $right$, $invCount$, $concCount$, $tiesCount$
Output: Sorted $y$ and $w$ from index $left$ until index $right$, together with updated values
  for $invCount$, $concCount$ and $tiesCount$

  $Left \leftarrow y[left, middle]$, $Right \leftarrow y[middle + 1, right]$
  $w_{Left} \leftarrow w[left, middle]$, $w_{Right} \leftarrow w[middle + 1, right]$

  $n \leftarrow$ length $Left$, $m \leftarrow$ length $Right$
  $i \leftarrow$ index in $Left$, initialized at 0
  $j \leftarrow$ index in $Right$, initialized at 0
  $k \leftarrow$ index in $y$, initialized at $left$
  $i_2 \leftarrow$ index in $Left$ pointing to smallest element that is at least $\nu$ larger than $Right[j]$,
  initialized at 0
  $j_2 \leftarrow$index in $Right$ pointing to smallest element that is at least $\nu$ larger than $Left[i]$,
  initialized at 0
  $tiesCountLeft \leftarrow 0$, $tiesCountRight \leftarrow 0$
  $s_{Left} \leftarrow \sum w_{Left}, s_{Right} \leftarrow \sum w_{Right}$

  while $i < n$ or $j < m$ do
    if $i_2 < n$ and $Left[i_2] \leq Right[j] + \nu$ then
      if $Left[i_2] - \nu == Right[j]$ then
        $tiesCountLeft \leftarrow tiesCountLeft + w_{Left}[i_2]$
      end if
      $s_{Left} = s_{Left} - w_{Left}[i_2]$
      $i_2 \leftarrow i_2 + 1$
    else if $j_2 < m$ and $Right[j_2] \leq Left[i] + \nu$ then
      if $Right[j_2] - \nu == Left[i]$ then
        $tiesCountRight \leftarrow tiesCountRight + w_{Right}[j_2]$
      end if
      $s_{Right} = s_{Right} - w_{Right}[j_2]$
      $j_2 \leftarrow j_2 + 1$
    else if $Left[i] < Right[j]$ then
        $concCount \leftarrow concCount + w_{Left}[i] s_{Right}$
        $tiesCount \leftarrow tiesCount + w_{Left}[i] tiesCountRight$
        $y[k] \leftarrow Left[i]$
        $w[k] \leftarrow w_{Left}[i]$
        $k \leftarrow k + 1$
        $i \leftarrow i + 1$
        if $(i < n)$ and $(Left[i]! = Left[i-1])$ then
            $tiesCountRight \leftarrow 0$
        end if
    else
        $invCount \leftarrow invCount + w_{Right}[j] s_{Left}$
        if $\nu > 0$ then
            $tiesCount \leftarrow tiesCount + w_{Right}[j] tiesCountLeft$
        end if
        $y[k] \leftarrow Right[j]$
        $w[k] \leftarrow w_{Right}[j]$
        $k \leftarrow k + 1$
        $j \leftarrow j + 1$
        if $(j < m)$ and $(Right[j]! = Right[j-1])$ then
            $tiesCountLeft \leftarrow 0$
        end if
    end if
    while $i < n$ do
        $y[k] \leftarrow Left[i]$
        $w[k] \leftarrow w_{Left[i]}$
        $k \leftarrow k + 1$
        $i \leftarrow i + 1$
    end while
    while $j < m$ do
        $y[k] \leftarrow Right[j]$
        $w[k] \leftarrow w_{Right[j]}$
        $k \leftarrow k + 1$
        $j \leftarrow j + 1$
    end while
end while
```

Appendix B. Data description

Name	Interpretation
`Year`	The covering year. Categorical variable with 3 levels (2011, 2012 and 2013).
`VehiclPower`	The vehicle power. Categorical variable with 11 levels (P1, P2, . . . , P11).
`VehiclAge`	The vehicle age. Categorical variable with 2 levels ($0 - 5$ years, > 5 years).
`Deduc`	The deductible category. Categorical variable with 6 levels (0 euro, $1 - 200$ euro, $201 - 300$ euro, $301 - 400$ euro, $401 - 600$ euro, > 600 euro).
`BusinessType`	The business type. Categorical variable with 8 levels (B1, B2, . . . , B8).
`ChannelDist`	The distribution channel. Categorical variable with 3 levels (D1, D2, D3).
`PolicyAgeCateg`	The category of the policy age. Categorical variable with 6 levels ($0 - 1$ year, $1 - 2$ years, $2 - 3$ years, $3 - 4$ years, $4 - 5$ years, > 5 years).
`PolicyCateg`	The category of the policy. Categorical variable with 4 levels (C2,C3,C4,C5).
`CompanyCreation`	A dummy indicating if the company has been created.
`FleetMgt`	The fleet management category. Categorical variable with 2 levels (N, P).
`FleetSizeCateg`	The fleet size category. Categorical variable with 2 levels (S1, S2).
`Area`	The geographical area. Categorical variable with 6 levels (A1, A2, . . . , A6).
`PayFreq`	The payment frequency. Categorical variable with 3 levels (quarter, semester, year).

Table B.1: The selected predictor variables from the data set `pg16trainpol`, that are used to predict the number of claims.

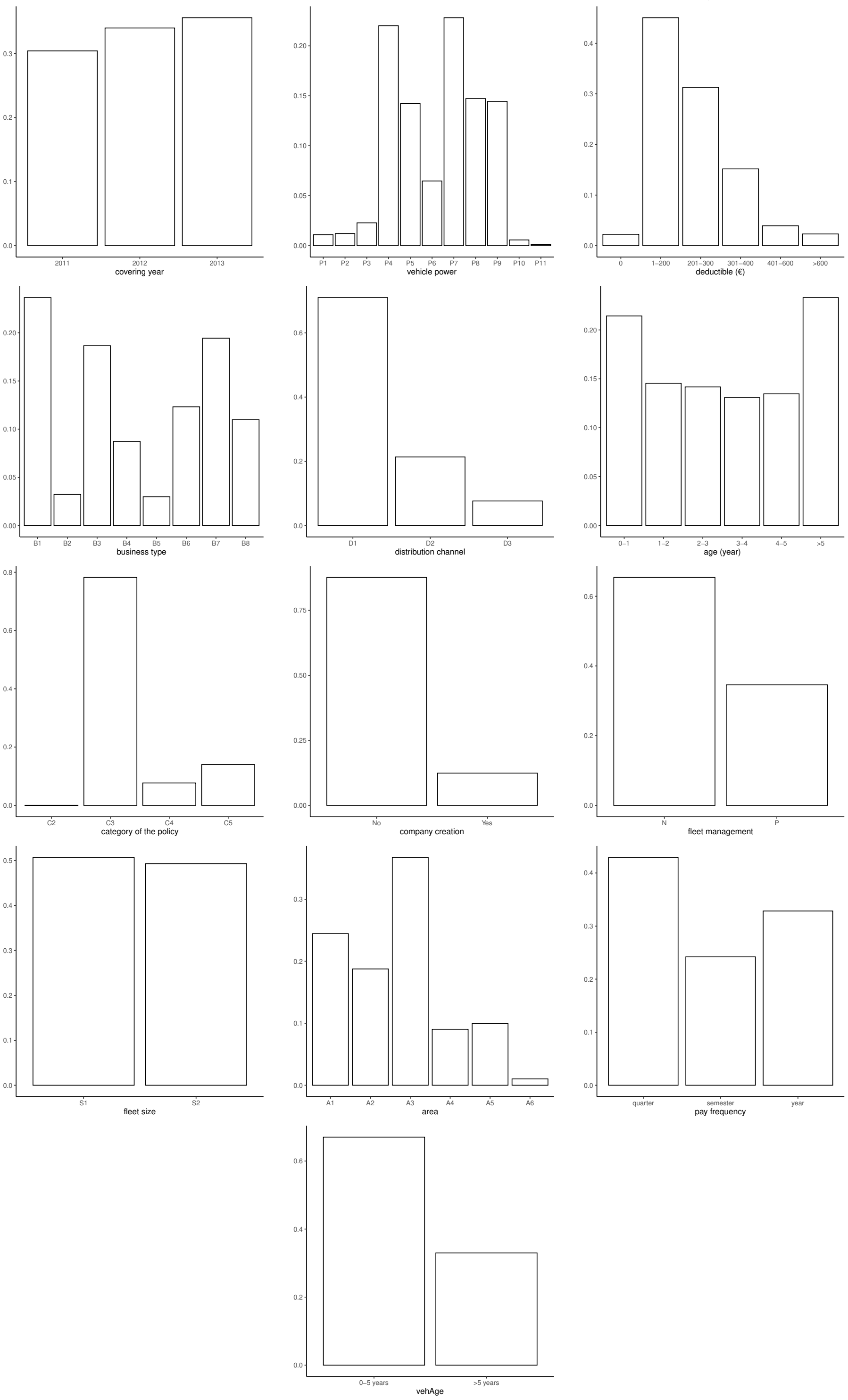


Figure B.1: Bar plots for each considered predictor variable from the data set `pg16trainpol` to predict the number of claims.

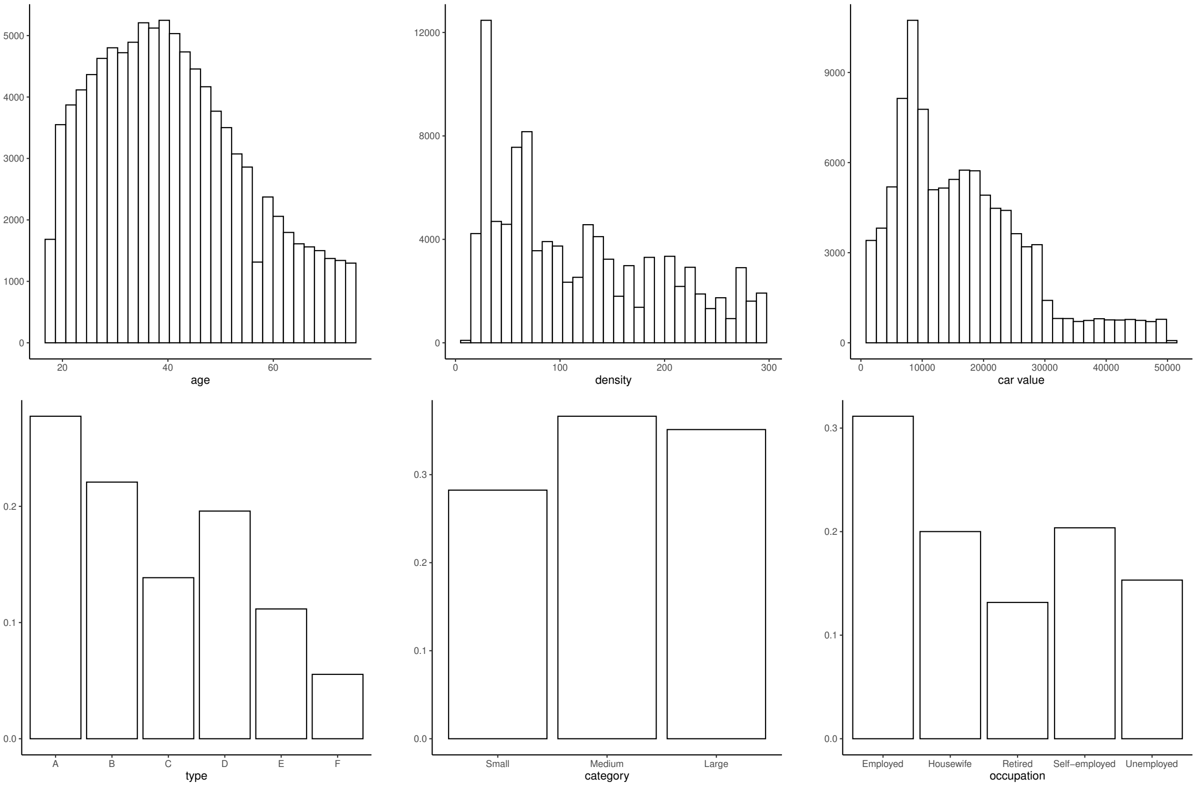


Figure B.2: Histograms and bar plots of the considered variables from the data set `pg15taining` to predict the number of claims.

Name	Interpretation
`Age`	The drivers' age, expressed in years.
`Density`	The population density (number of inhabitants per square km) in the city that the driver of the car lives in.
`Value`	The car value in euro.
`Type`	The car type. Categorical variable with 6 levels (A, B, C, D, E, F).
`Category`	The car category. Categorical variable with 3 levels (Small, Medium, Large).
`Occupation`	The occupation of the driver. Categorical variable with 5 levels (Employed, Housewife, Retired, Self-employed and Unemployed).

Table B.2: The selected predictor variable from the data set `pg15training`, that are used to predict the number of claims.

Declaration of interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors gratefully acknowledge the support of International Funds KU Leuven, grant C16/15/068.

References

[1] M. J. Pencina, R. B. D'Agostino, Overall c as a measure of discrimination in survival analysis: model specific population value and confidence interval estimation, Statistics in medicine 23 (2004) 2109–2123.

[2] L. Thomas, The concordance statistic, 2021. `https://cran.r-project.org/web/packages/survival/vignettes/concordance.pdf` [Accessed: 2021-04-28].

[3] R. Van Oirbeek, J. Ponnet, T. Verdonck, Computational efficient approximations of the concordance probability in a big data setting (2021). Under Review.

[4] J. Ponnet, R. Van Oirbeek, T. Verdonck, Concordance probability for insurance pricing models, Risks 9 (2021) 178.

[5] H. Putter, dynpred: Companion Package to "Dynamic Prediction in Clinical Survival Analysis", 2015. URL: `https://CRAN.R-project.org/package=dynpred`, r package version 0.1.2.

[6] G. Heller, Q. Mo, Estimating the concordance probability in a survival analysis with a discrete number of risk groups, Lifetime data analysis 22 (2016) 263–279.

[7] F. E. H. Jr, Hmisc: Harrell Miscellaneous, 2021. URL: https://cran.r-project.org/package=Hmisc, r package version 4.5-0.

[8] T. A. Gerds, pec: Prediction Error Curves for Risk Prediction Models in Survival Analysis, 2020. URL: https://CRAN.R-project.org/package=pec, r package version 2020.11.17.

[9] B. Haibe-Kains, M. Schroeder, C. Olsen, C. Sotiriou, G. Bontempi, J. Quackenbush, S. Branders, Z. Safikhani, survcomp: Performance Assessment and Comparison for Survival Analysis, 2008. URL: http://www.bioconductor.org/packages/release/bioc/html/survcomp.html, r package version 1.42.0.

[10] T. M. Therneau, survival: A Package for Survival Analysis in R, 2021. URL: https://CRAN.R-project.org/package=survival, r package version 3.2-13.

[11] H. H. Goldstine, J. Von Neumann, J. Von Neumann, Planning and coding of problems for an electronic computing instrument (1947).

[12] M. G. Kendall, A new measure of rank correlation, Biometrika 30 (1938) 81–93.

[13] P. Filzmoser, H. Fritz, K. Kalcher, pcaPP: Robust PCA by Projection Pursuit, 2022. URL: https://CRAN.R-project.org/package=pcaPP, r package version 2.0-1.

[14] M. Hofert, I. Kojadinovic, M. Maechler, J. Yan, copula: Multivariate Dependence with Copulas, 2022. URL: https://CRAN.R-project.org/package=copula, r package version 1.1-1.

[15] S. S. Skiena, The algorithm design manual, volume 2, Springer, 1998.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2022.

[17] C. Dutang, A. Charpentier, M. C. Dutang, Package 'casdatasets', Christophe Dutang and Arthur Charpentier (2020).

[18] J. Kennedy, R. C. Eberhart, A discrete binary version of the particle swarm algorithm, in: 1997 IEEE International conference on systems, man, and cybernetics. Computational cybernetics and simulation, volume 5, IEEE, 1997, pp. 4104–4108.

[19] O. S. Qasim, Z. Y. Algamal, Feature selection using particle swarm optimization-based logistic regression model, Chemometrics and Intelligent Laboratory Systems 182 (2018) 41–46.

[20] S. Mirjalili, A. Lewis, S-shaped versus v-shaped transfer functions for binary particle swarm optimization, Swarm and Evolutionary Computation 9 (2013) 1–14.

[21] A. Khare, S. Rangnekar, Particle swarm optimization: A review, Applied Soft Computing (2012).

[22] P. De Jong, G. Z. Heller, et al., Generalized linear models for insurance data, Cambridge Books (2008).

[23] R. Tibshirani, Regression shrinkage and selection via the lasso, Journal of the Royal Statistical Society: Series B (Methodological) 58 (1996) 267–288.

[24] L. Yan, D. Verbel, O. Saidi, Predicting prostate cancer recurrence via maximizing the concordance index, in: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, 2004, pp. 479–485.

[25] Y. Chen, Z. Jia, D. Mercola, X. Xie, A gradient boosting algorithm for survival analysis via direct optimization of concordance index, Computational and mathematical methods in medicine 2013 (2013).

[26] A. Mayr, B. Hofner, M. Schmid, Boosting the discriminatory power of sparse survival models via optimization of the concordance index and stability selection, BMC bioinformatics 17 (2016) 1–12.

[27] V. Mingote, A. Miguel, A. Ortega, E. Lleida, Optimization of the area under the roc curve using neural network supervectors for text-dependent speaker verification, Computer Speech & Language 63 (2020) 101078.

[28] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, et al., Xgboost: extreme gradient boosting, R package version 0.4-2 1 (2015) 1–4.

[29] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, Lightgbm: A highly efficient gradient boosting decision tree, Advances in neural information processing systems 30 (2017).

[30] M. Dorigo, M. Birattari, T. Stutzle, Ant colony optimization, IEEE computational intelligence magazine 1 (2006) 28–39.

[31] N. Sreeja, A. Sankar, A hierarchical heterogeneous ant colony optimization based approach for efficient action rule mining, Swarm and Evolutionary Computation 29 (2016) 1–12.

[32] L. D. Chambers, The practical handbook of genetic algorithms: applications, Chapman and Hall/CRC, 2000.

[33] I. Fister, X.-S. Yang, D. Fister, Cuckoo search: a brief literature review, Cuckoo search and firefly algorithm (2014) 49–62.

[34] J. Kalderstam, P. Edén, P.-O. Bendahl, C. Strand, M. Fernö, M. Ohlsson, Training artificial neural networks directly on the concordance index for censored data using genetic algorithms, Artificial intelligence in medicine 58 (2013) 125–132.

[35] L. Scrucca, Ga: a package for genetic algorithms in r, Journal of Statistical Software 53 (2013) 1–37.

[36] E. LeDell, N. Gill, S. Aiello, A. Fu, A. Candel, C. Click, T. Kraljevic, T. Nykodym, P. Aboyoun, M. Kurka, et al., Package 'h2o', dim 2 (2018) 17.

[37] H. Daniels, M. G. Kendall, The significance of rank correlations where parental correlation exists, Biometrika 34 (1947) 197–208.

[38] N. Cliff, V. Charlin, Variances and covariances of kendall's tau and their estimation, Multivariate Behavioral Research 26 (1991) 693–707.