Aus der Klinik und Poliklinik für Innere Medizin B: Kardiologie, Angiologie, Pulmologie/Infektiologie und Internistische Intensivmedizin
(Direktor: Univ.- Prof. Dr. Stephan B. Felix)
der Universitätsmedizin der Universität Greifswald

Deep learning prediction of all-cause-mortality in a general population cohort by myocardial strain derived from speckle-tracking-echocardiography.

Inaugural - Dissertation

zur

Erlangung des akademischen

Grades

Doktor der Medizin
(Dr. med.)

der

Universitätsmedizin

der

Universität Greifswald

2022

vorgelegt von:
Fabian Christopher Laqua
geb. am: 17.09.1992
in: Karlsruhe

Dekan: Univ. - Prof. Dr. med. Karlhans Endlich

1. Gutachter: Univ. - Prof. Dr. med. Marcus Dörr

2. Gutachter: Univ.-Prof. Dr. med. Bettina Baeßler

(3. Gutachter:)

Ort, Raum: Greifswald, Seminarraum Innere Medizin B

Tag der Disputation: 21.03.2023

Table of contents

1 Abstract 5

Background 5

Purpose 5

Methods 5

Results 5

Conclusion 5

2 Introduction 6

3 Methods 8

3.1 Study Population, population-at-risk, and time horizon 8

3.2 Outcome 8

3.3 Speckle Tracking Echocardiography 9

3.4 Statistical analysis, machine learning and performance evaluation 9

3.5 Restricted cubic splines neural network 'nnet-Surv-rcsplines' 10

3.6 Specification of model input and benchmark 13

3.7 Performance Evaluation 13

4 Results 16

4.1 Study population 16

4.2 Performance of 'nnet-Surv-rcsplines' models 19

4.3 Sensitivity analysis: performance of 'nnet-survival' 29

5 Discussion 36

5.1 Global peak strain from Speckle-tracking-echocardiography 36

5.2 Speckle-tracking-echocardiography-derived multidimensional data 37

5.3 'nnet-Surv-rcsplines' neural network for survival data with and without presence of competing risks 39

5.4 Study limitations 40
6 Conclusion 42
7 References 43
8 Appendix A 1
8.1 Restricted cubic splines neural network ‘nnet-Surv-rcsplines’ 1
Cause-specific cumulative incidence function and subdistribution hazard 1
Regression modelling 1
Negative Log-Likelihood-Loss for competing risk data 3
Special case Survival data without competing risks 5
8.2 Source code of the Tensorflow 2 Implementation of ‘nnet-Surv-rcsplines’ 6
8.3 Source code of the Tensorflow 2 Implementation of ‘nnet-survival’ 21
8.4 Source code of lowess-smoothed calibration plot for survival data 27
9 Appendix B 34

1 Abstract

Background

Previous work has focused on speckle-tracking echocardiography (STE)-derived global longitudinal and circumferential peak strain as potential superior prognostic metric markers compared with left ventricular ejection fraction (LVEF). However, the value of regional distribution and the respective orientation of left ventricular wall motion (quantified as strain and derived from STE) for survival prediction have not been investigated yet. Moreover, most of the recent studies on risk stratification in primary and secondary prevention do not use neural networks for outcome prediction.

Purpose

To evaluate the performance of neural networks for predicting all cause-mortality with different model inputs in a moderate-sized general population cohort.

Methods

All participants of the second cohort of the population-based Study of Health in Pomerania (SHIP-TREND-0) without prior cardiovascular disease (CVD; acute myocardial infarction, cardiac surgery/intervention, heart failure and stroke) and with transthoracic echocardiography exams were followed for all-cause mortality from baseline examination (2008-2012) until 2019.

A novel deep neural network architecture 'nnet-Surv-rcsplines', that extends the Royston-Parmar- cubic splines survival model to neural networks was proposed and applied to predict all-cause mortality from STE-derived global and/or regional myocardial longitudinal, circumferential, transverse, and radial strain in addition to the components of the ESC SCORE model. The models were evaluated by 8.5-year area-under-the-receiver-operating-characteristic (AUROC) and (scaled) Brier score [(S)BS]and compared to the SCORE model adjusted for mortality rates in Germany in 2010.

Results

In total, 3858 participants (53 % female, median age 51 years) were followed for a median time of 8.4 (95 % CI 8.3 – 8.5) years. Application of 'nnet-Surv-rcsplines' to the components of the ESC SCORE model alone resulted in the best discriminatory performance (AUROC 0.9 [0.86-0.91]) and lowest prediction error (SBS 21[18-23] %). The latter was significantly lower (p <0.001) than the original SCORE model (SBS 11 [9.5 - 13] %), while discrimination did not differ significantly. There was no difference in (S)BS (p= 0.66) when global circumferential and longitudinal strain were added to the model. Solely including STE-data resulted in an informative (AUROC 0.71 [0.69, 0.74]; SBS 3.6 [2.8-4.6] %) but worse (p<0.001) model performance than when considering the sociodemographic and instrumental biomarkers, too.

Conclusion

Regional myocardial strain distribution contains prognostic information for predicting all-cause mortality in a primary prevention sample of subjects without CVD. Still, the incremental prognostic value of STE parameters was not demonstrated. Application of neural networks on available traditional risk factors in primary prevention may improve outcome prediction compared to standard statistical approaches and lead to better treatment decisions.

2 Introduction

Previous work has focused on speckle-tracking echocardiography (STE)-derived global longitudinal and circumferential peak strain as potential superior prognostic metric markers compared with left ventricular ejection fraction (LVEF)[1-3]. The association of those parameters with outcomes (incl. adverse cardiovascular events, cardiovascular- and all-cause mortality) was investigated in different settings, cohorts and for other disease entities[1-10]. Despite the potentially misleading term 'predictor variable', most of the available literature claiming to show 'incremental predictive capabilities' only focuses on 'independent association' in multivariable survival regression, which is subject to causal inference frameworks[11-13].

The development and evaluation of models incorporating STE-derived cardiac motion parameters for predicting survival outcomes have not been subject to research as yet[1,8,9]. Especially information on the regional distribution of peak strain and strain rate parameters with varying orientation (transversal, longitudinal, circumferential, and radial), available due to modern speckle tracking software and ultrasound hardware, has not been integrated into prediction models so far[1,5,7,9]. In this work, the regional distribution and the respective orientation of left ventricular wall motion (quantified as strain and derived from STE) have been treated in the same way as other multi-dimensional data (e.g., radiomics, genomics, metabolomics etc.).

Multi-dimensional data requires specific statistical methods dealing with this high dimensionality to exploit the available information effectively [12-16]. Recently, machine learning methods, specifically deep neural networks, have successfully been applied to different tasks in cardiovascular medicine [17-20]. However, literature on using deep neural networks for risk prediction in primary prevention in the general population is sparse[21-24]. Even the update of the widely used '**S**ystematic **CO**ronary **R**isk **E**valuation'(SCORE) risk charts, namely SCORE2[25] and SCORE2-OP[26], derived from the data of 677,684 individuals, relies on traditional survival/competing risk regression[25]. The used methods do not address potential non-linear effects of covariables and only explicit first-level interaction terms. A major drawback preventing a widespread application of deep learning models for risk prediction lies in the complexity of time-to-event data with potential right- or left-censoring and competing

risks[27,28]. Different approaches have been proposed for applying deep learning models on survival and competing risk datarv[29], Coxnnet[30], nnet-survival[27], DeepHit[31]). While the former two extend Cox's proportional hazard model to neural networks, the latter two are multi-task networks that model the survival distribution at discrete time intervals.

In the present work, a novel deep neural network for survival data (including competing risk data), extending Royston and Parmar's flexible parameterization of the survival function by restricted cubic splines to model the cause-specific cumulative incidence function (CIF) in a neural network, has been proposed[32,33]. This 'nnet-Surv-rcsplines' model has been applied to predict all-cause mortality in a general population cohort. This study has aimed to assess the performance of neural networks for predicting all cause-mortality with different model inputs in a moderate-sized general population cohort. Inputs ranged from components of the SCORE risk chart models to high dimensional STE-data.

3 Methods

For the development and reporting of prediction models, we applied the structure proposed in the TRIPOD and CONSORT-AI statement, taking the study population at risk, the time horizon, the outcome of interest, and the choice of predictors into consideration[12,13,34,35].

3.1 Study Population, population-at-risk, and time horizon

The Study of Health in Pomerania (SHIP) is a population-based, epidemiological project conducted in northeastern Germany[36]. For this work, data from the baseline examinations of the second cohort (SHIP-TREND-0), collected between September 2008 and September 2012, was used. For SHIP-TREND-0, a random, stratified sample of 8,016 adults aged 20 - 79 years was drawn using local population registries in the Federal State of Mecklenburg/West Pomerania[36]. Stratification variables were age, sex, and city/county of residence[36]. In total, 4,420 individuals took part in the examinations (response 50.1 %) and gave informed written consent[36]. The Ethics Committee of the University of Greifswald has approved this study. This study was conducted in accordance with the Declaration of Helsinki, following all relevant guidelines and regulations.

All participants of SHIP-TREND-0 with a complete mortality follow-up and STE data of sufficient quality for the apical four-chamber and the apical 2-chamber view at least have been included in our investigation. As the baseline, we have defined the date of examination at the study site. We have considered 10 years as a relevant time horizon for developing and assessing survival prediction models[13]. In total, data of 3,858 individuals out of 4,420 participants of SHIP-TREND-0 have been analyzed in the present work (cf. **Figure 2**).

3.2 Outcome

Minimization of mortality is a primary target in cardiovascular prevention. Hence, all-cause mortality was chosen as the primary outcome. Patients were followed from baseline until death, emigration or July 2019, whichever came first. Information on the vital status of patients was obtained from official resident data files[36]. Subjects were either classified as censored at the date of record in the population registry or as dead.

3.3 Speckle Tracking Echocardiography

All participants underwent routine M-Mode, B-Mode, continuous-wave Doppler, pulse-wave Doppler and tissue doppler imaging echocardiography. Left ventricular dimensions were measured using the leading-edge convention[37,38]. Moreover, left ventricular mass (LVM) and left ventricular diastolic function markers were evaluated. Echocardiography was conducted using vivid-I cardiovascular ultrasound devices (GE Medical Systems, Waukesha, Wisconsin, WI, USA).

According to a standardized protocol, two-dimensional speckle-tracking-based image analyses of the left ventricle were performed using an offline vendor-independent software (2D Cardiac Performance Analysis v1.2.3.6 b.141117, TomTec Imaging Systems, Unterschleissheim, Germany) by two readers. LVEF was calculated in the standard fashion from end-diastolic and end-systolic volumes derived from a modified Simson biplane approach[38], using the aforementioned software tool. Peak global circumferential strain (GCS) was determined from the parasternal short axis in the papillary muscle plane. Peak global longitudinal strain (GLS) was calculated as the mean of global longitudinal strain values measured in apical two-chamber and apical four-chamber view. In addition, the STE software calculated peak segmental endocardial strain, strain rates (i.e., the first derivative of strain) and the respective time-to-peak.

3.4 Statistical analysis, machine learning and performance evaluation

Baseline characteristics of the study population were compared by inclusion/exclusion using Pearson's χ^2-test for categorical and Wilcoxon rank sum test for continuous variables, respectively. Prior to model training, a two-stage Random Forest single imputation ("MissForest"[39]) method was used to impute missing values in the dataset. In the first stage, the missing values of the clinical covariables (age, sex, serum cholesterol, systolic blood pressure and current smoking status) were imputed using this information only. In the second stage, missing values in the speckle tracking data (e.g., missing segmental strain) were imputed. Among representative deep learning methods for survival data, a discrete-time survival model called 'nnet-survival'[27] served as a backend for a dense neural network with

modern deep learning techniques (Relu-Activation[40], batch normalization[41], dropout regularization, training with ADAM-optimizer[42]).

Furthermore, the continuous-time Royston-Parmar cubic spline regression (RPCS)[32] was extended as a backend for a deep, dense neural network called 'nnet-Surv-rcsplines', detailed in the next paragraph and Appendix 8.1 p. 1 to 6. The 'nnet-survival' model[27] serves as a benchmark for the proposed 'nnet-Surv-rcsplines' model in terms of a sensitivity analysis. The widely used Cox proportional hazard model was not included in the final analyses since convergence failed for several combinations of input variables, which was also part of the motivation to use more appropriate high-dimensional methods.

The regularization of the neural networks (dropout/ L1/L2-penalty) serves as an embedded data-driven variable selection/weighting process[43].

3.5 Restricted cubic splines neural network 'nnet-Surv-rcsplines'

Royston and Parmar introduced a flexible parametric survival model that models the survival function on the logistic or log hazard scale using restricted cubic splines[32,33]. In this work, this approach is generalized to an implementation of a neural network (cf. **Figure 1**). In brief, the neural network is interposed between the input variables and the input vector of the traditional Royston and Parmar model. Changes in the input variables lead to vertical translation of the spline on the respective modelling scale (e.g., log hazard). In contrast to the original (generalized) linear model, also the vertical position of the control points ('knots') could depend on the (potentially a different set of) input variables if specified. In addition, this model was also extended to competing risks, where the cause-specific cumulative incidence function is modelled similar to Fine and Gray's popular competing risk regression model [44,45]. If no hidden neural network layers are chosen, the model simplifies to the original restricted cubic spline model of Royston and Parmar[32]. The complete mathematical derivation of the 'nnet-Surv-rcsplines' model was detailed in Appendix 8.1, p. 1 to 6.

The Tensorflow Layers and Submodels of the actual Python implementation (cf. Appendix 8.2, p. 6 to 20) may be used as modules for more complex neural networks (e.g., convolutional or

recurrent neural networks for image and time series data, respectively). In addition, the high-level scikit-learn-style API can be used out-of-the-box on tabular data.

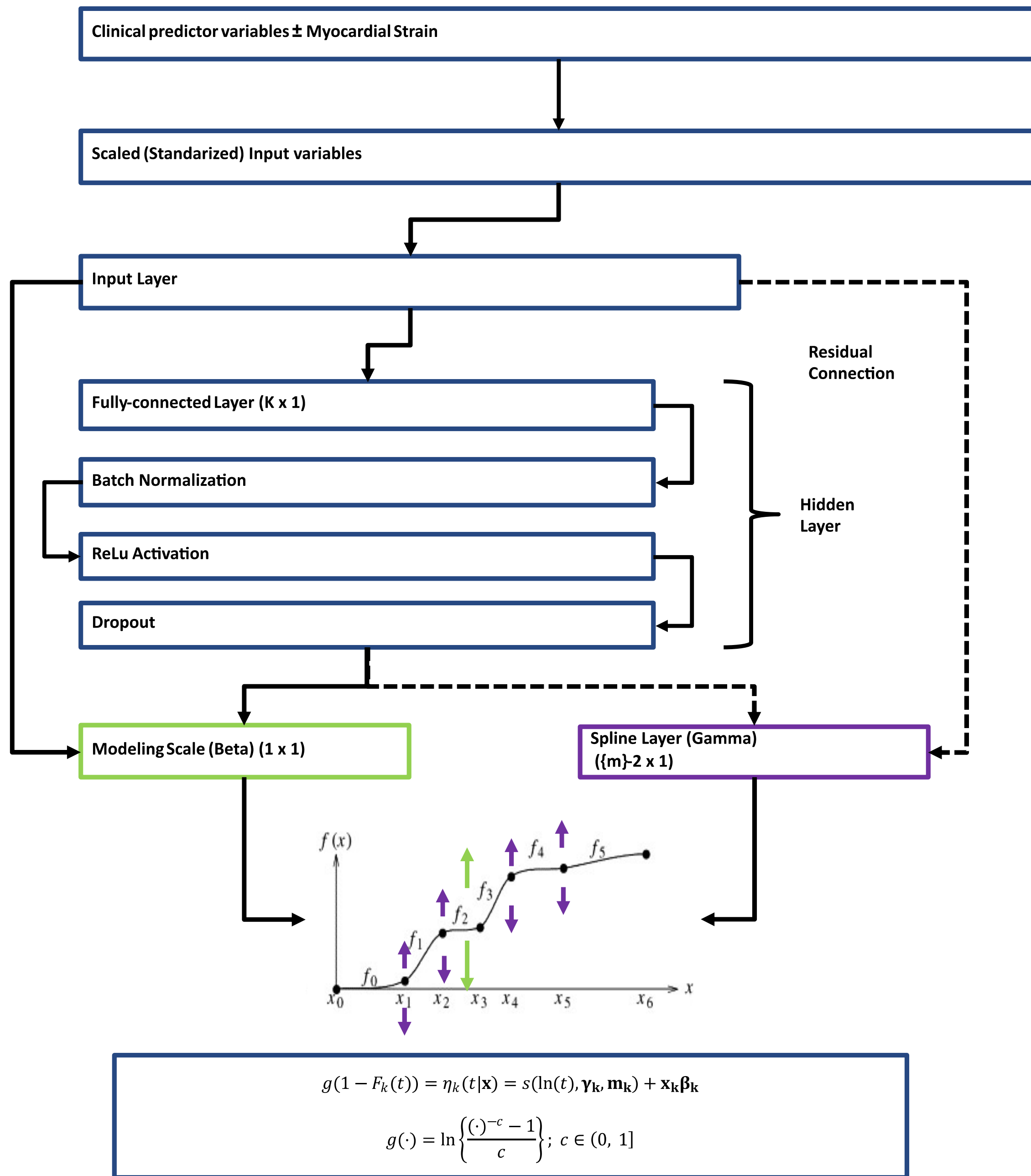

Figure 1. 'nnet-Surv-rcsplines' neural network architecture
The deep layers (blue) serve as the frontend for extracting information from the input. In this case, an extended Multi-layer-Perceptron with modern deep learning components (dropout, ReLu Activation, batch normalization and residual connections) is used. The Spline Layer (Gamma, violet) models the shape of the spline for the cause-specific cumulative incidence function $F_k(t)$ on the respective scale due to the link function $g(\cdot)$ (for $c \to 0$ and c = 1, this is the cumulative hazard and cumulative odds scale, respectively). The Modelling Scale (Beta) models the influence of the input variables on the global spline position (i.e., displacement on the y-axis) on the respective scale.

3.6 Specification of model input and benchmark

The Weibull survival model underlying the widely used SCORE risk charts[46] was considered as a reference. Here, the stratified Weibull regression coefficients reported for Germany (low-risk country) were applied to predict cardiovascular disease (CVD) mortality. The estimated CVD mortality (ICD-10 I10-I15, I20-I25, I44-I51 and I61-I74, R96-99) was multiplied by an age-group- and sex-specific weighting factor derived from official German mortality registries of 2010[47] to estimate the cumulative incidence function for all-cause mortality. This approach assumes that relative risks estimated on the SCORE derivation cohort did not change over time.

For the deep learning models, information on age, sex, high-density lipoprotein, systolic blood pressure and current smoking status (SCORE components), underlying the SCORE risk chart model for predicting cardiovascular mortality[46], was supplied to serve as input for the neural networks. In total, the compared models comprise:

Table 1. Overview of applied models.

i)	the calibrated SCORE model for prediction of all-cause mortality
ii)	neural networks applied to SCORE components only
iii)	neural networks applied to SCORE components, GLS and GCS,
iv)	neural networks applied to predicted SCORE survival-probability and GLS and GCS,
v)	STE-data (global and segmental peak circumferential/ longitudinal/ radial/ transversal strain (rate)).
vi)	SCORE components and STE-data

For the multidimensional STE-data models (v) and vi)), principal component analysis was applied prior to model training for feature reduction. The number of components was chosen to explain ≥ 95 % of the variance.

3.7 Performance Evaluation

A repeated 10-fold internal cross-validation was conducted to train and test the proposed models[48,49]. Specifically, at each cycle, a random 10 % of the training dataset was strictly held-

out and used as the test-set for the models trained on the remaining 90 % of the data. This process was repeated ten times, and the mean of the calculated metrics was reported.

Hyperparameters (i.e., additional model parameters that are not optimized during training) were chosen to minimize the 8.5-year Brier score[28] by repeated bootstrap-bias-corrected 10-fold cross-validation on the training cohort using a randomized search protocol. Optimized hyperparameters comprise learning rate, choice of regularization (dropout vs L1/L2-penalty), dropout rate/lambda for L1/L2, depth and width of the dense neural network and Aranda-Ordaz-link-coefficient.

Briefly, the pooled predictions on the relative hold-out-folds were bootstrapped. The performance of the respective best hyperparameter configuration was calculated on the samples not used in the bootstrap ("out-of-bag") to correct for the optimistic bias induced by hyperparameter tuning[49].

Discrimination ability was assessed by time-varying receiver-operating-characteristics (ROC) in the cumulative incidence, dynamic controls formulation [50], and corresponding area-under-the-curve (AUROC) over the time range of 7 to 10 years after baseline.

Calibration was visually assessed using loess-smoothed calibration plots calculated on jackknife-pseudo-values[51,52].

The predictive accuracy was estimated by inverse probability of censoring weighted mean square error (Brier score, BS) for right-censored data and by a derived scaled R2-like measure, calculated by $R_G^2 = 1 - \frac{BS}{BS_{max}}$, respectively[28,53].

Five hundred bootstrap samples from the pooled out-of-fold predictions were used to calculate 2.5 %- and 97.5 %-percentile confidence intervals. Bootstrap z-tests were calculated on the metrics' differences across models. An alpha level of 0.01 was chosen as the threshold for statistical significance. No correction for multiple testing was applied in concordance with arguments for this approach in observational studies published earlier[54]. However, results should be interpreted as hypothesis generating.

Data preparation and model implementation were performed in Stata 15.1 and Python 3.7.3, respectively, with the following extra modules: Tensorflow 2.4.2[55] lifelines 0.24.12[56], scikit-survival 0.11[57], scikit-learn 1.0.1[58], and ELI5 0·10·1. The used source code includes the proposed 'nnet-Surv-rcsplines' survival model, the Tensorflow 2 implementation of 'nnet-survival'27 and the python implementation of the SCORE risk chart model accessible via github.com/laqua-stack/SHIP_Survival_ECHO. In addition, the source code for the 'nnet-Surv-rcsplines', 'nnet-survival', and the lowess-smoothed calibration plot is given in Appendix p. 6 to 20, 21 to 26 and 27 to 33, respectively.

All experiments were carried out on the high-performance computation cluster of the data centre of the University of Greifswald.

4 Results

4.1 Study population

In total, 3858 participants of SHIP-TREND-0 with valid echocardiography and mortality data (cf. **Figure 2**) and without a history of cardiovascular disease events were followed for a median follow-up time of 8.4 (95 % CI 8.3 – 8.5) years. Baseline characteristics of the study population separated by study inclusion are given in **Table 2**. Excluded participants were predominantly older ($p<0.001$), were more frequently male ($p<0.001$), had more comorbidities (history of hypertension, heart failure, diabetes mellitus, atrial fibrillation, cardiovascular diseases, renal diseases, cancer; all $p<0.001$), and more frequent medication (cf. **Table 2**; $p\leq0.01$). Average myocardial strains (except for peak transversal strain) and left ventricular ejection fraction were also significantly lower in the excluded study population ($p\leq0.001$).

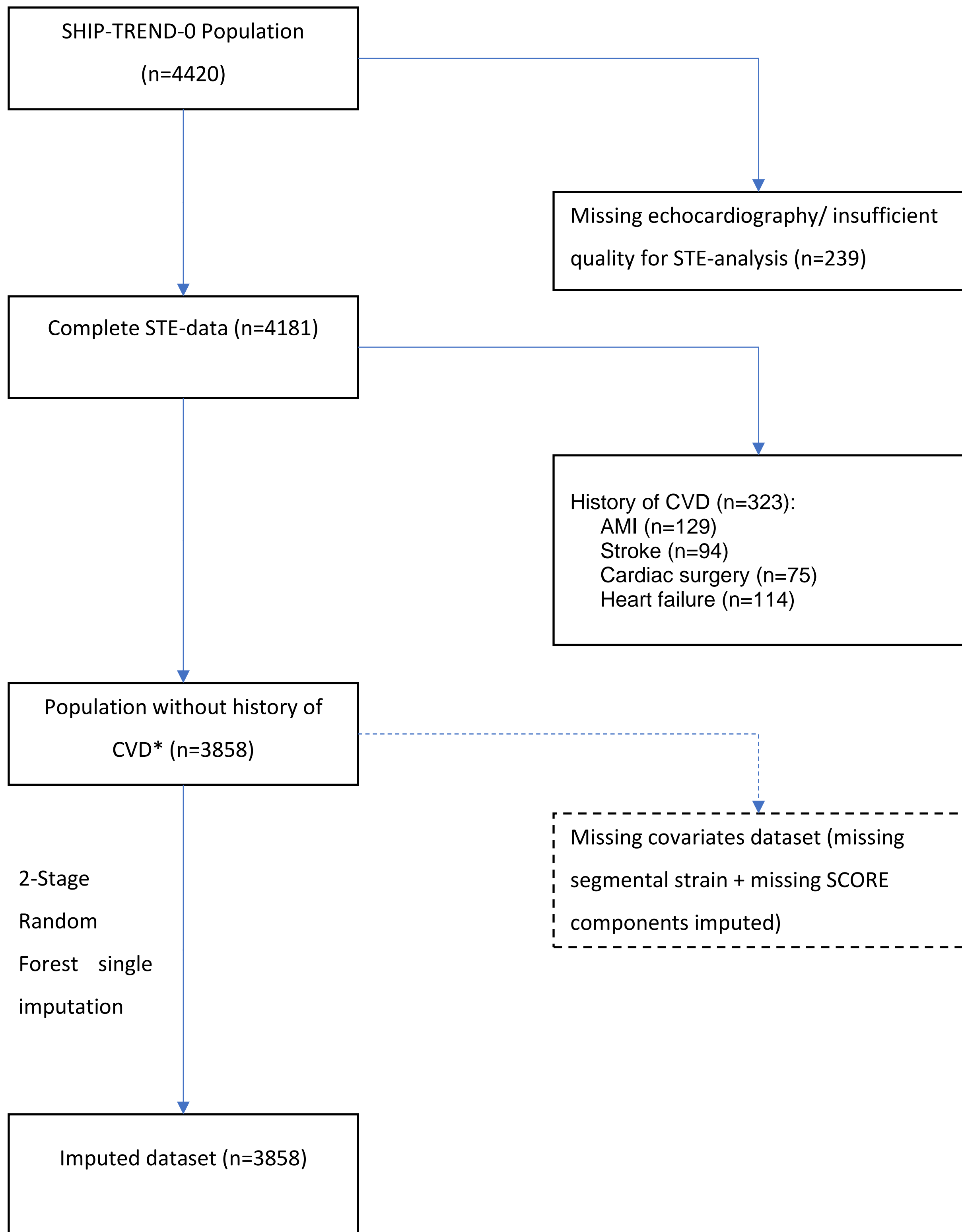

Figure 2. Flowchart of the study population
SHIP = Study of Health in Pomerania, CVD = cardiovascular disease (here defined as history of acute myocardial infarction, stroke, heart failure and cardiac surgery/intervention), AMI = acute myocardial infarction, SCORE = 'Systematic COronary Risk Evaluation', STE = speckle tracking echocardiography

Table 2. Baseline characteristics of the study population

Median (interquartile range) and absolute count (relative percentage) are given for continuous and binary variables, respectively. Excluded patients were those without sufficient echocardiography and a history of cardiovascular disease (acute myocardial infarction, stroke, cardiac surgery/intervention, heart failure). P-values for Pearson's χ2-test for categorical and Wilcoxon rank sum test for continuous variables, respectively. eGFR = estimated glomerular filtration rate, LVEF = Left ventricular ejection fraction, PAI = Platelet aggregation inhibitors, OAD = obstructive airway diseases, P(L/T/C/R)S = peak circumferential/longitudinal/radial/transversal strain, 4ch = apical four chamber view, 2ch = apical two chamber view, sax_pm = short axis view at papillary muscle plane

	Excluded * (N=562)	Study population (N=3,858)	p-value
Age (years)	62 (52-72)	51 (39-63)	<0.001
Male sex	351 (62%)	1,794 (47%)	<0.001
Height (cm)	170 (163-176)	170 (163-177)	0.85
Weight (kg)	84 (75-95)	79 (68-91)	<0.001
Current smoker	99 (18%)	1,084 (28%)	<0.001
Ever smoker	354 (64%)	2,439 (63%)	0.93
History of			
Atrial fibrillation	97 (18%)	138 (4%)	<0.001
Acute myocardial infarction	133 (24%)	0 (0%)	<0.001
Stroke	100 (18%)	0 (0%)	<0.001
Cardiac surgery/intervention	80 (14%)	0 (0%)	<0.001
Heart failure	122 (22%)	0 (0%)	<0.001
Diabetes mellitus	121 (22%)	343 (9%)	<0.001
Renal disease	35 (6%)	108 (3%)	<0.001
Cancer	56 (11%)	232 (6%)	<0.001
Vital parameters			
Systolic blood pressure (mmHg)	129.5 (117.5-141.5)	126.5 (114.0-139.0)	<0.001
Diastolic blood pressure (mmHg)	76.0 (69.5-83.5)	76.5 (70.0-83.5)	0.12
Laboratory parameters			
Serum cholesterol (mmol/l)	5.1 (4.4-5.9)	5.4 (4.7-6.2)	<0.001
High-density lipoprotein (mmol/l)	1.3 (1.1-1.6)	1.4 (1.2-1.7)	<0.001
Low-density lipoprotein (mmol/l)	3.0 (2.5-3.8)	3.3 (2.7-4.0)	<0.001
eGFR (ml/min/1.73 m² BSA)	90.2 (76.4-100.2)	97.6 (86.2-108.3)	<0.001
Medication			
Anti-diabetic drugs incl. insulin	99 (18%)	248 (6%)	<0.001
Antihypertensive medication	366 (65%)	1,319 (34%)	<0.001
Lipid-lowering drugs	219 (39%)	400 (10%)	<0.001
Anticoagulation / PAI	265 (47%)	357 (9%)	<0.001
Drugs for OAD	33 (6%)	140 (4%)	0.010
Anti-neoplastic drugs	3 (1%)	2 (0%)	0.001
Speckle tracking echocardiography			
Average PLS 4ch in %	-15 (-19--12)	-17 (-20--14)	<0.001
Average PTS 4ch in %	15 (9-22)	15 (9-22)	0.64
Average PLS 2ch in %	-15 (-19--11)	-18 (-21--14)	<0.001
Average PTS 2ch in %	13 (7-20)	16 (9-23)	<0.001
Average PCS sax_pm in %	-26 (-32--20)	-28 (-32--23)	0.001
Average PRS sax_pm in %	23 (16-33)	27 (19-35)	<0.001
LVEF in %	53 (45-60)	56 (49-62)	<0.001

4.2 Performance of ‘nnet-Surv-rcsplines’ models

Table 3. Model performance of the respective 'nnet-Surv-rcsplines' models. *SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)*

	Year	**8**	**8.5**	**9**	**9.5**	**10**
	Model					
AUROC	**i) SCORE predicted probability**	0.88 (0.87-0.90)	0.89 (0.88-0.90)	0.88 (0.86-0.89)	0.86 (0.84-0.88)	0.82 (0.79-0.84)
	Nnet-Surv-rcsplines: ii) SCORE components	0.89 (0.88-0.90)	0.90 (0.88-0.91)	0.89 (0.87-0.90)	0.86 (0.84-0.88)	0.83 (0.81-0.86)
	Nnet-Surv-rcsplines: iii) SCORE components + GLS + GCS	0.88 (0.87-0.90)	0.89 (0.88-0.91)	0.88 (0.86-0.90)	0.84 (0.82-0.87)	0.81 (0.78-0.84)
	Nnet-Surv-rcsplines: iv) SCORE pred. prob. + GLS + GCS	0.87 (0.86-0.89)	0.88 (0.86-0.89)	0.86 (0.84-0.88)	0.83 (0.81-0.86)	0.79 (0.75-0.82)
	Nnet-Surv-rcsplines: v) STE-data	0.70 (0.68-0.73)	0.71 (0.69-0.74)	0.71 (0.68-0.73)	0.69 (0.67-0.72)	0.66 (0.62-0.69)
	Nnet-Surv-rcsplines: vi) SCORE components + STE-data	0.84 (0.82-0.86)	0.84 (0.83-0.86)	0.84 (0.82-0.85)	0.83 (0.81-0.85)	0.78 (0.75-0.82)
Brier score (BS)	**i) SCORE predicted probability**	0.033 (0.030-0.036)	0.042 (0.038-0.046)	0.053 (0.049-0.058)	0.074 (0.069-0.080)	0.086 (0.080-0.093)
	Nnet-Surv-rcsplines: ii) SCORE components	0.032 (0.029-0.035)	0.038 (0.035-0.041)	0.047 (0.044-0.051)	0.054 (0.049-0.059)	0.077 (0.068-0.086)
	Nnet-Surv-rcsplines: iii) SCORE components + GLS + GCS	0.032 (0.030-0.035)	0.038 (0.035-0.041)	0.048 (0.044-0.051)	0.055 (0.050-0.060)	0.080 (0.069-0.090)
	Nnet-Surv-rcsplines: iv) SCORE pred. prob. + GLS + GCS	0.033 (0.031-0.036)	0.040 (0.037-0.044)	0.050 (0.046-0.053)	0.056 (0.051-0.061)	0.084 (0.076-0.093)
	Nnet-Surv-rcsplines: v) STE-data	0.036 (0.033-0.039)	0.046 (0.042-0.050)	0.059 (0.055-0.063)	0.070 (0.065-0.075)	0.095 (0.089-0.100)
	Nnet-Surv-rcsplines: vi) SCORE components + STE-data	0.034 (0.031-0.037)	0.041 (0.038-0.045)	0.053 (0.049-0.056)	0.057 (0.052-0.061)	0.094 (0.083-0.100)
R^2 (scaled BS) in %	**i) SCORE predicted probability**	10 (8.5-12)	11 (9.6-13)	13 (11-14)	1.4 (-0.98-3.9)	9.7 (8.0-12)
	Nnet-Surv-rcsplines: ii) SCORE components	13 (11-15)	21 (18-23)	23 (19-26)	28 (25-32)	19 (9.7-28)
	Nnet-Surv-rcsplines: iii) SCORE components + GLS + GCS	11 (8.3-14)	20 (17-23)	22 (18-26)	27 (23-31)	16 (4.9-27)
	Nnet-Surv-rcsplines: iv) SCORE pred. prob. + GLS + GCS	9.3 (7.6-11)	15 (13-17)	19 (15-22)	26 (22-29)	11 (2.4-21)
	Nnet-Surv-rcsplines: v) STE-data	2.4 (1.6-3.3)	3.6 (2.8-4.6)	3.6 (2.3-4.8)	7.4 (5.7-9.1)	0.25 (-5.3-5.1)
	Nnet-Surv-rcsplines: vi) SCORE components + STE-data	7.7 (5.6-9.6)	13 (11-15)	14 (11-17)	24 (21-28)	1.3 (-10-13)

The results of the performance evaluation for the different models are given in **Table 3** and **Figures 3-7**.

The AUROC at the respective year (cf. **Figure 3**) describes the discrimination performance, meaning it measures whether a person who was dead by this time had a higher predicted probability of dying than a person who survived beyond this time. A value of 1.0 means perfect discrimination. If the model had no discriminative ability (i.e., toss of a coin) on the investigated population, this would result in an AUROC of 0.5. AUROC values below 0.5 occur if the model predicts an informative but wrong ordering. Discrimination (ROC and the corresponding AUROC) varied slightly over time from baseline, with a maximum at 8.5 years, where the highest density of outcomes lies as well (cf. **Figure 3**, **Table 3**).

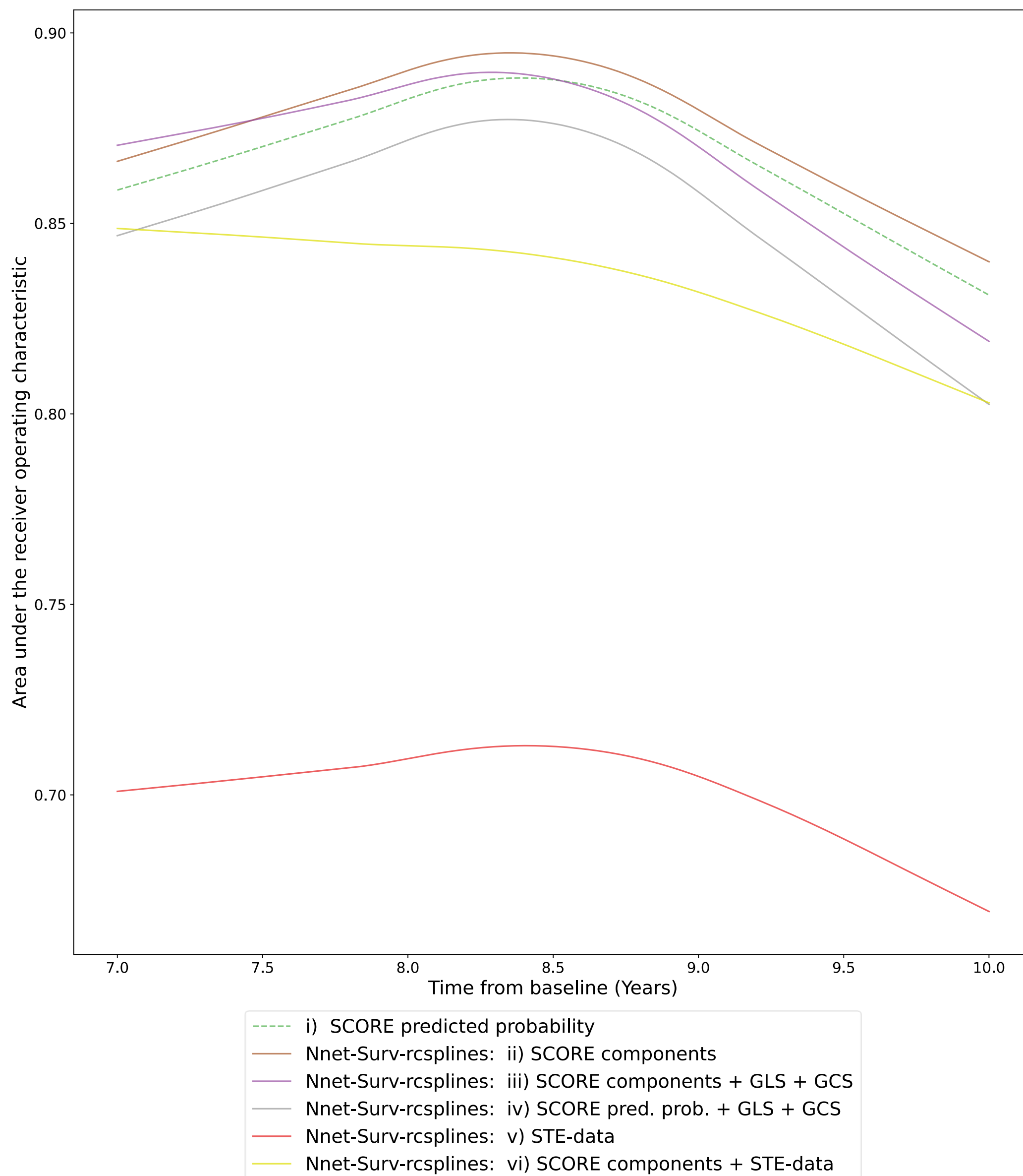

Figure 3. Illustration of time-dependent Area under the Receiver-operating-characteristic (AUROC) for the different 'nnet-Surv-rcsplines'- models.
The inverse-probability-of-censoring weighted cumulative-dynamic AUROC (higher values are better) is plotted from 6 to 10 years after baseline. SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

The scaled Brier score (cf. **Figure 4**) gives the fraction by which the mean square error corrected for right-censoring is reduced compared to an uninformative model (i.e., predicting the same average survival probability for every subject without considering any specific information). A perfect SBS equals 100 %. An SBS of 0 % means that the model provides no information benefit. SBS below 0 means that the prediction error is even higher (e.g., because the model is miscalibrated) than that of a naïve calibrated model.

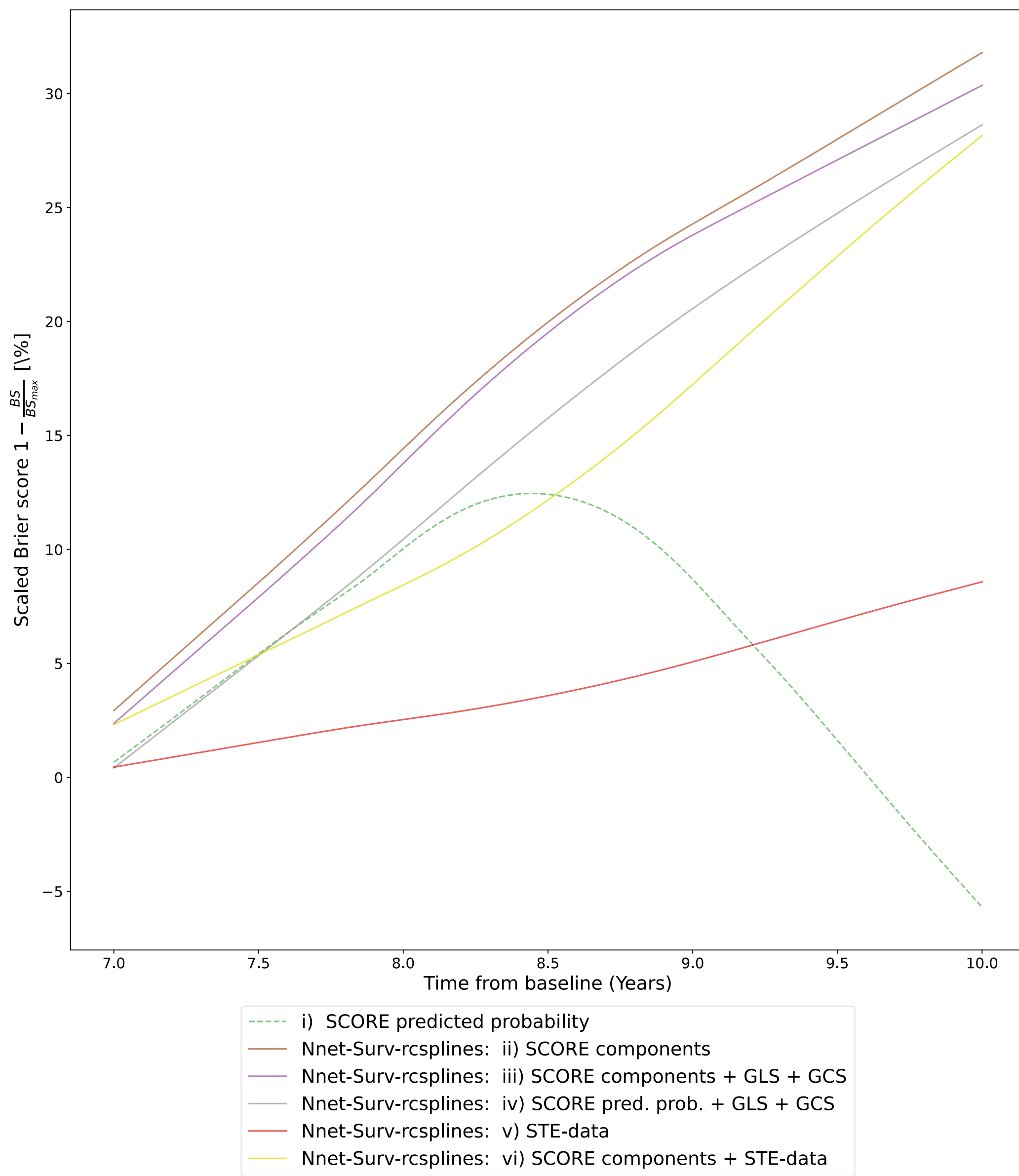

Figure 4. Illustration of time-dependent scaled Brier score for different 'nnet-Surv-rcsplines'- models. *The inverse-probability-of-censoring weighted scaled brier score (higher values are better) is plotted from 6 to 10 years after baseline. SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)*

Overall, the ‘nnet-Surv-rcsplines’ models with SCORE components as input (models ii) and iii)) had the best performance, both in terms of discrimination (AUROC up to 0.9 (0.88-0.91) at 8.5 years from baseline for model iii)) and calibration (SBS up to 28 % (95 % CI 25-32) at 9.5 years from baseline for model iii), cf. also **Figure 3** and **Figure 4**). However, the confidence intervals of models ii) and iii) were highly overlapping. Visually, the ROC curves of model iii) were closest to the top left, while the v) STE-data model showed the poorest discrimination (**Figure 4**). Despite comparative discrimination (AUROC 0.89 (0.87-0.90) at 8.5 years from baseline), the reference model i) showed the worst calibration (cf. departure from diagonal in the calibration plot in **Figure 3**.) with SBS ranging from 1.4 % to 13 %. At 8.5 years from baseline, the benchmark model i) was significantly ($p<0.001$, z ranging from 4.8 to 7.7) worse than models ii)-iv) in terms of (S)BS. Addition of GCS and GLS to the SCORE components (model iii) vs. model ii)) did not significantly change model performance ($p=0.66$, $z = 0.44$).

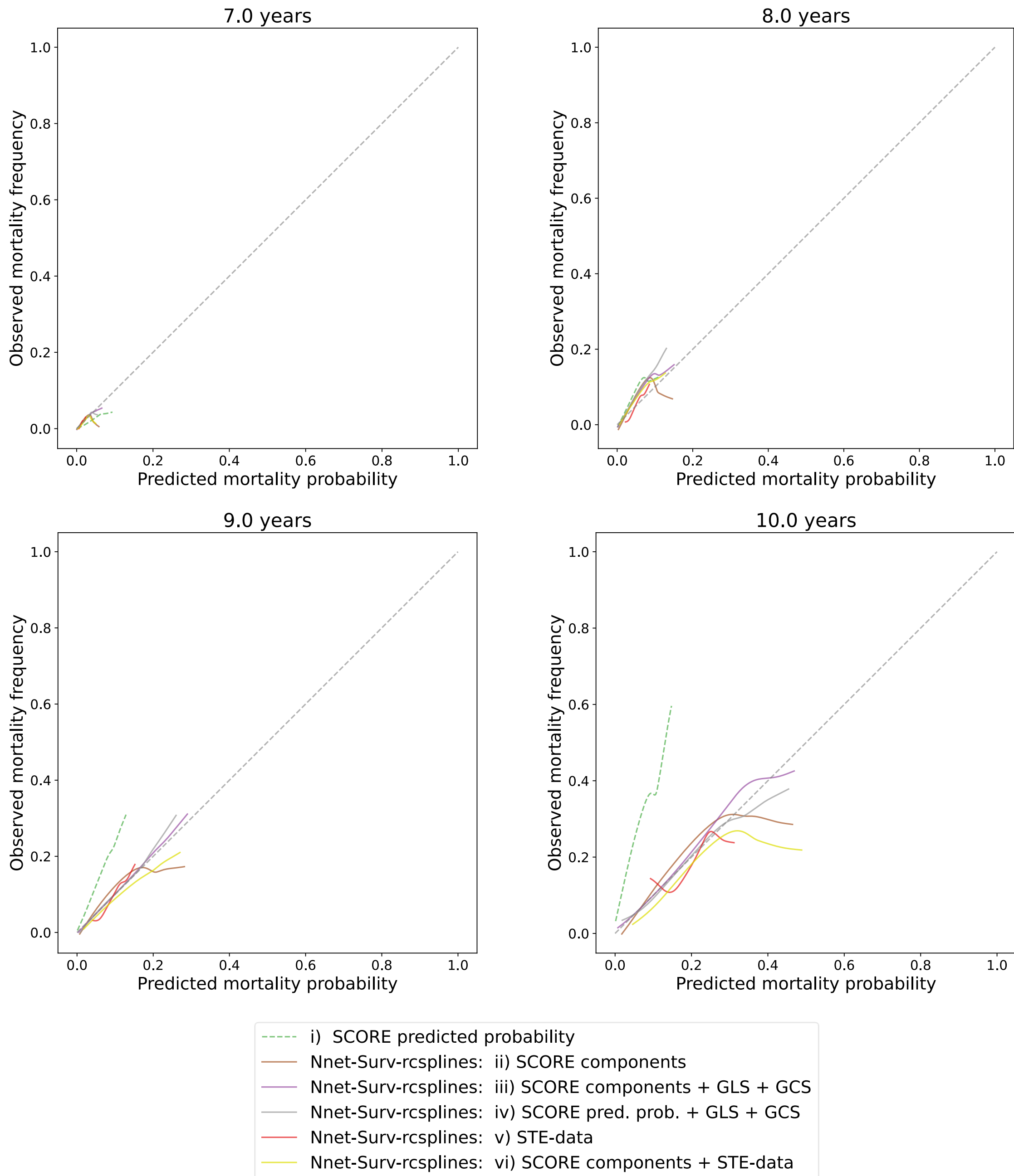

Figure 5. Calibration capabilities for the different machine learning models compared to SCORE risk chart reference model calculated for different points in time.

In the lowess-smoothed calibration plot the observed mortality frequency is plotted against the predicted mortality probability. The closer the curve is to the diagonal, the better the calibration. SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

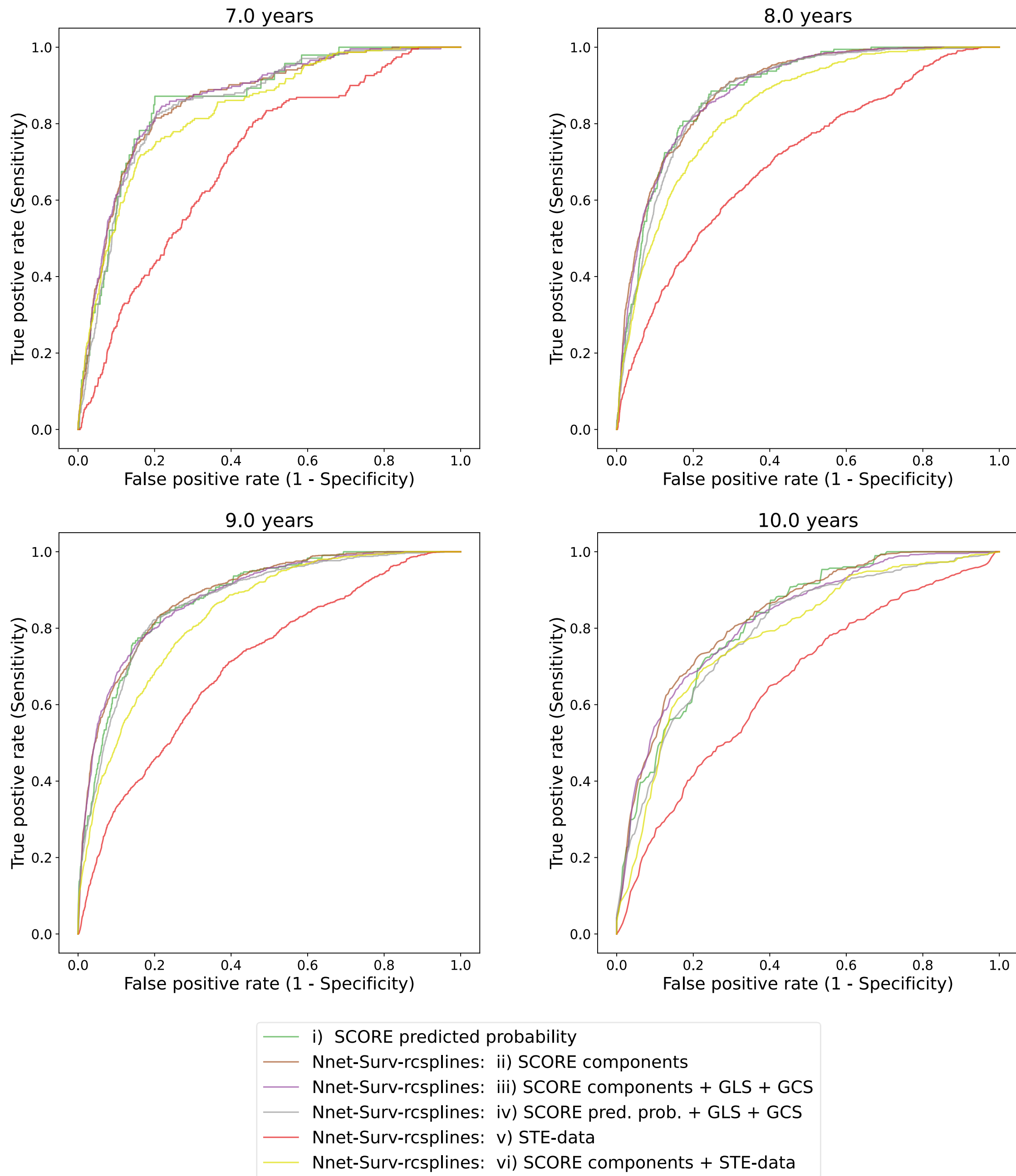

Figure 6. Receiver-operating-characteristics for the different 'nnet-Surv-rcsplines'-models compared to SCORE risk chart reference model calculated for different points in time.

The receiver-operating-characteristic plots true positive rate against false positive rate by varying thresholds (not shown). Discrimination is best for the curve, that is closest to the left upper corner. SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

In addition, the discrimination of the different models is illustrated in Kaplan-Meier plots (**Figure 7**), where the study population is divided into terciles by the predicted survival probability. Overall, separation of the lines concomitant with the models' discriminatory performance is similar across the models, with visually best separation for the models ii) comprising SCORE components only and worst for the model v) with STE-data as input only.

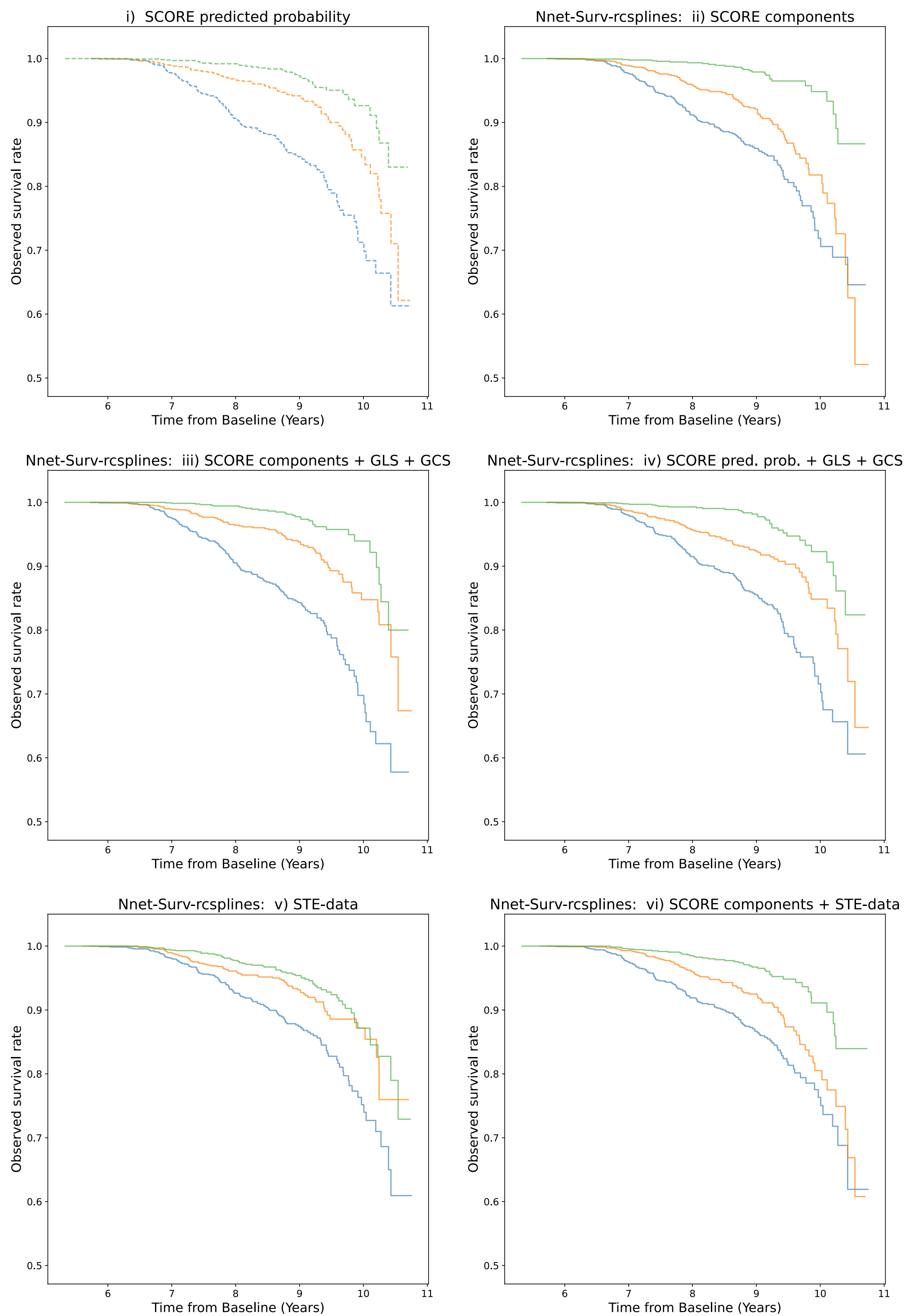

Figure 7. Kaplan-Meier survival curves by tercile of the predicted survival probability for the different 'nnet-Surv-rcsplines'- models compared to SCORE risk chart reference model.
Kaplan-Meier curves for the respective subpopulation, divided by tercile (green upper, yellow middle, blue lower tercile) of the predicted survival probability, are plotted from 6 to 10 years after baseline (Time before 6 years not shown, since no events occurred). SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

4.3 Sensitivity analysis: performance of ‘nnet-survival’

The results of discrete-time-survival neural network ‘nnet-survival’ serve as a sensitivity analysis, and the respective model inputs are presented in **Table 4** and **Figures 8-11**. Compared to the ‘nnet-Surv-rcsplines’, very similar results occurred for the respective model inputs. While minor differences in the absolute model performance metrics for the best set of input variables exist (e.g., 8.5-year SBS of ii) was 22 (19-24) % and 21 (18-23) % for ‘nnet-survival’ and ‘nnet-Surv-rcsplines’, respectively), the results for ‘nnet-Surv-rcsplines’ and ‘nnet-survival’ differed relevantly for the largest model vi). Differences were most pronounced at 10 years from baseline. There were also no significant differences between the models ii) and iii) (p=0.19, z=1.3).

Figure 12 presents the Kaplan-Meier survival curves for the respective subpopulation separated by terciles of the predicted survival probability. Like for the ‘nnet-Surv-rcsplines’, the separation of the lines concomitant with the models’ discriminatory performance is similar across the models, with visually best separation for the models ii) comprising SCORE components only and worst for the model v) with only STE-data as input.

Table 4. Model performance of the respective 'nnet-survival' models.

SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

Metric	Year	8	8.5	9	9.5	10
	Model					
AUROC	**i) SCORE predicted probability**	0.88 (0.87-0.90)	0.89 (0.88-0.90)	0.88 (0.86-0.89)	0.86 (0.84-0.88)	0.82 (0.79-0.84)
	Nnet-survival: ii) SCORE components	0.89 (0.87-0.90)	0.90 (0.88-0.91)	0.88 (0.87-0.90)	0.86 (0.84-0.88)	0.84 (0.81-0.86)
	Nnet-survival: iii) SCORE components + GLS + GCS	0.88 (0.87-0.89)	0.89 (0.88-0.90)	0.88 (0.86-0.89)	0.86 (0.84-0.88)	0.83 (0.81-0.86)
	Nnet-survival: iv) SCORE pred. prob. + GLS + GCS	0.87 (0.84-0.89)	0.88 (0.85-0.89)	0.86 (0.84-0.88)	0.82 (0.79-0.85)	0.78 (0.73-0.81)
	Nnet-survival: v) STE-data	0.71 (0.68-0.73)	0.72 (0.69-0.74)	0.71 (0.68-0.73)	0.70 (0.67-0.72)	0.66 (0.62-0.69)
	Nnet-survival: vi) SCORE components + STE-data	0.83 (0.80-0.86)	0.84 (0.80-0.86)	0.83 (0.79-0.86)	0.82 (0.78-0.84)	0.75 (0.71-0.79)
Brier score (BS)	**i) SCORE predicted probability**	0.033 (0.030-0.036)	0.042 (0.038-0.046)	0.053 (0.049-0.058)	0.074 (0.069-0.080)	0.086 (0.080-0.093)
	Nnet-survival: ii) SCORE components	0.032 (0.029-0.034)	0.037 (0.034-0.040)	0.048 (0.045-0.052)	0.052 (0.048-0.057)	0.082 (0.071-0.093)
	Nnet-survival: iii) SCORE components + GLS + GCS	0.032 (0.030-0.035)	0.038 (0.035-0.041)	0.049 (0.045-0.052)	0.054 (0.049-0.058)	0.082 (0.070-0.098)
	Nnet-survival: iv) SCORE pred. prob. + GLS + GCS	0.034 (0.031-0.036)	0.040 (0.037-0.043)	0.050 (0.047-0.054)	0.057 (0.053-0.062)	0.086 (0.077-0.094)
	Nnet-survival: v) STE-data	0.036 (0.033-0.039)	0.046 (0.042-0.049)	0.059 (0.055-0.063)	0.069 (0.065-0.074)	0.096 (0.089-0.100)
	Nnet-survival: vi) SCORE components + STE-data	0.035 (0.032-0.038)	0.042 (0.039-0.046)	0.056 (0.052-0.060)	0.058 (0.053-0.066)	0.120 (0.082-0.140)
R^2 (scaled BS) in %	**i) SCORE predicted probability**	10 (8.5-12)	11 (9.6-13)	13 (11-14)	1.4 (-0.98-3.9)	9.7 (8.0-12)
	Nnet-survival: ii) SCORE components	13 (9.6-16)	22 (19-24)	21 (18-25)	31 (26-35)	13 (1.1-25)
	Nnet-survival: iii) SCORE components + GLS + GCS	11 (8.1-14)	20 (17-23)	20 (17-24)	29 (25-33)	14 (-3.1-26)
	Nnet-survival: iv) SCORE pred. prob. + GLS + GCS	8.3 (5.1-11)	16 (13-19)	18 (13-21)	24 (20-28)	10 (0.94-19)
	Nnet-survival: v) STE-data	2.5 (1.5-3.6)	3.9 (2.9-5.1)	3.6 (2.0-5.2)	8.0 (6.0-10)	-0.58 (-7.3-5.1)
	Nnet-survival: vi) SCORE components + STE-data	3.4 (-0.6-6.9)	10 (7.6-13)	8.7 (4.8-13)	23 (14-28)	-27 (-54-13)

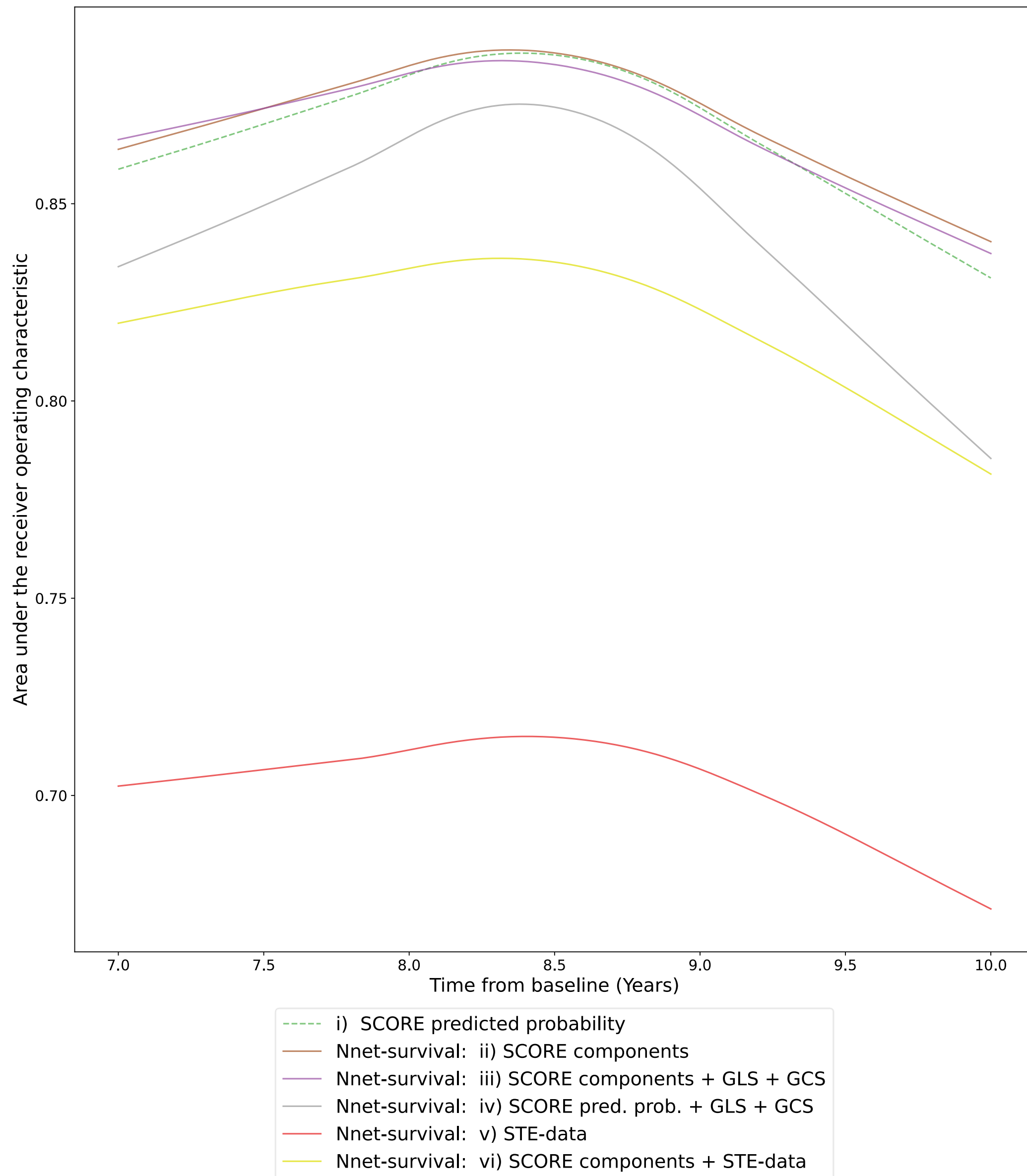

Figure 8. Illustration of time-dependent Area under the Receiver-operating-characteristic (AUROC) for the different 'nnet-survival'- models.

The inverse-probability-of-censoring weighted cumulative-dynamic AUROC (higher values are better) is plotted from 6 to 10 years after baseline. SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

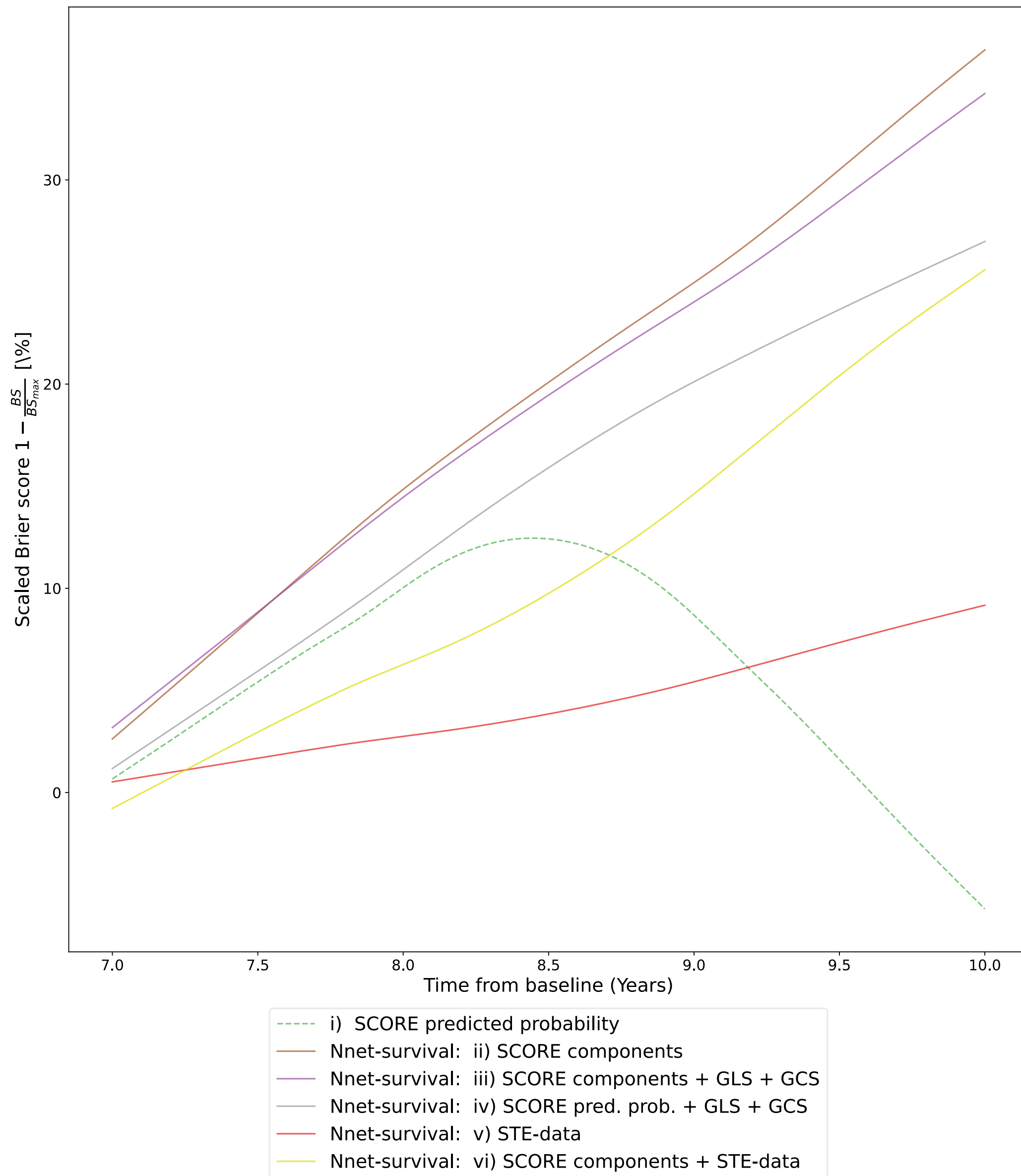

Figure 9. Illustration of time-dependent scaled Brier score for different 'nnet-survival'- models.
The inverse-probability-of-censoring weighted scaled brier score (higher values are better) is plotted from 6 to 10 years after baseline. SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

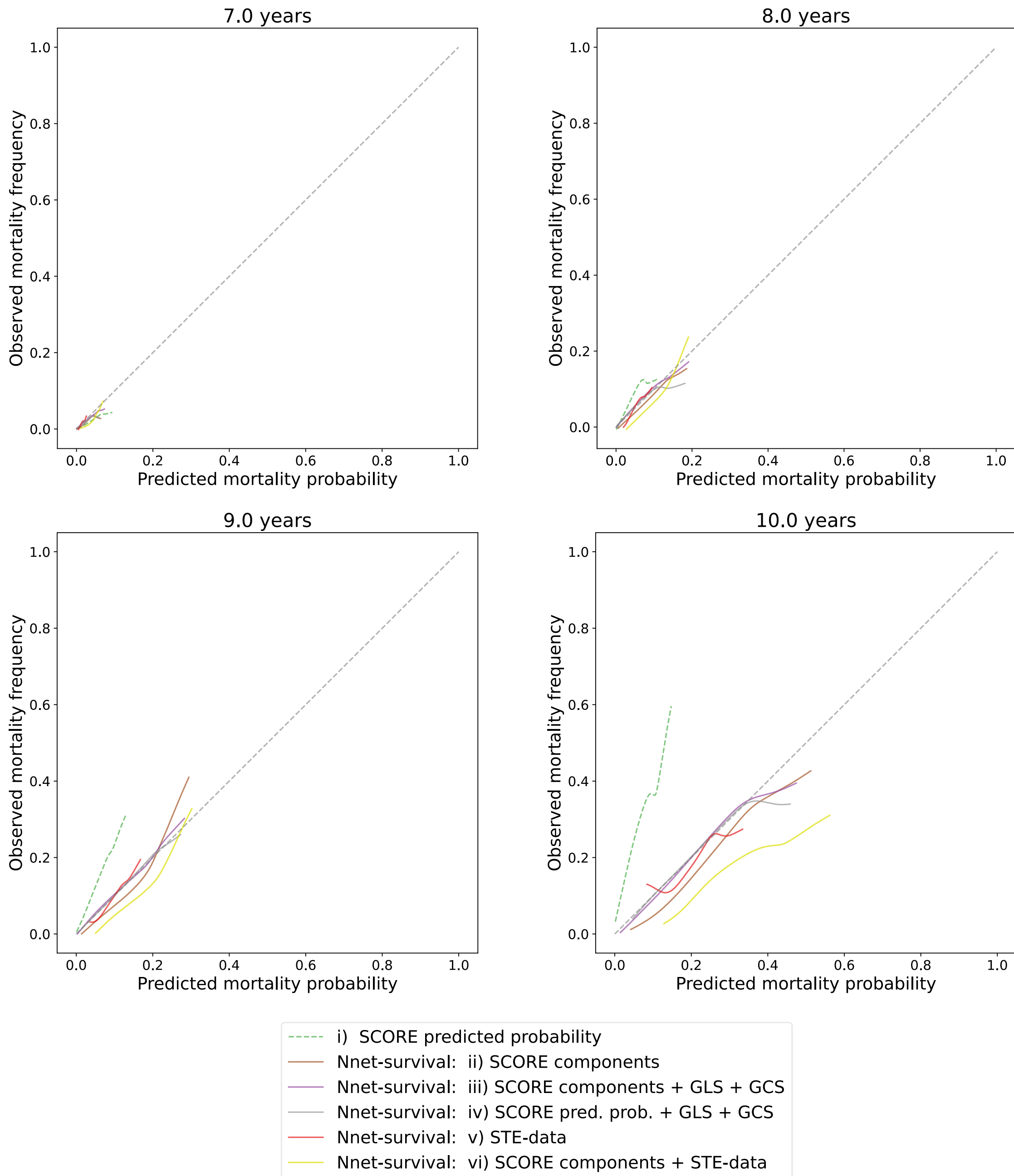

Figure 10. Calibration capabilities for the different 'nnet-survival'-models compared to SCORE risk chart reference model calculated for different points in time.

In the lowess-smoothed calibration plot the observed mortality frequency is plotted against the predicted mortality probability. The closer the curve is to the diagonal, the better the calibration. SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

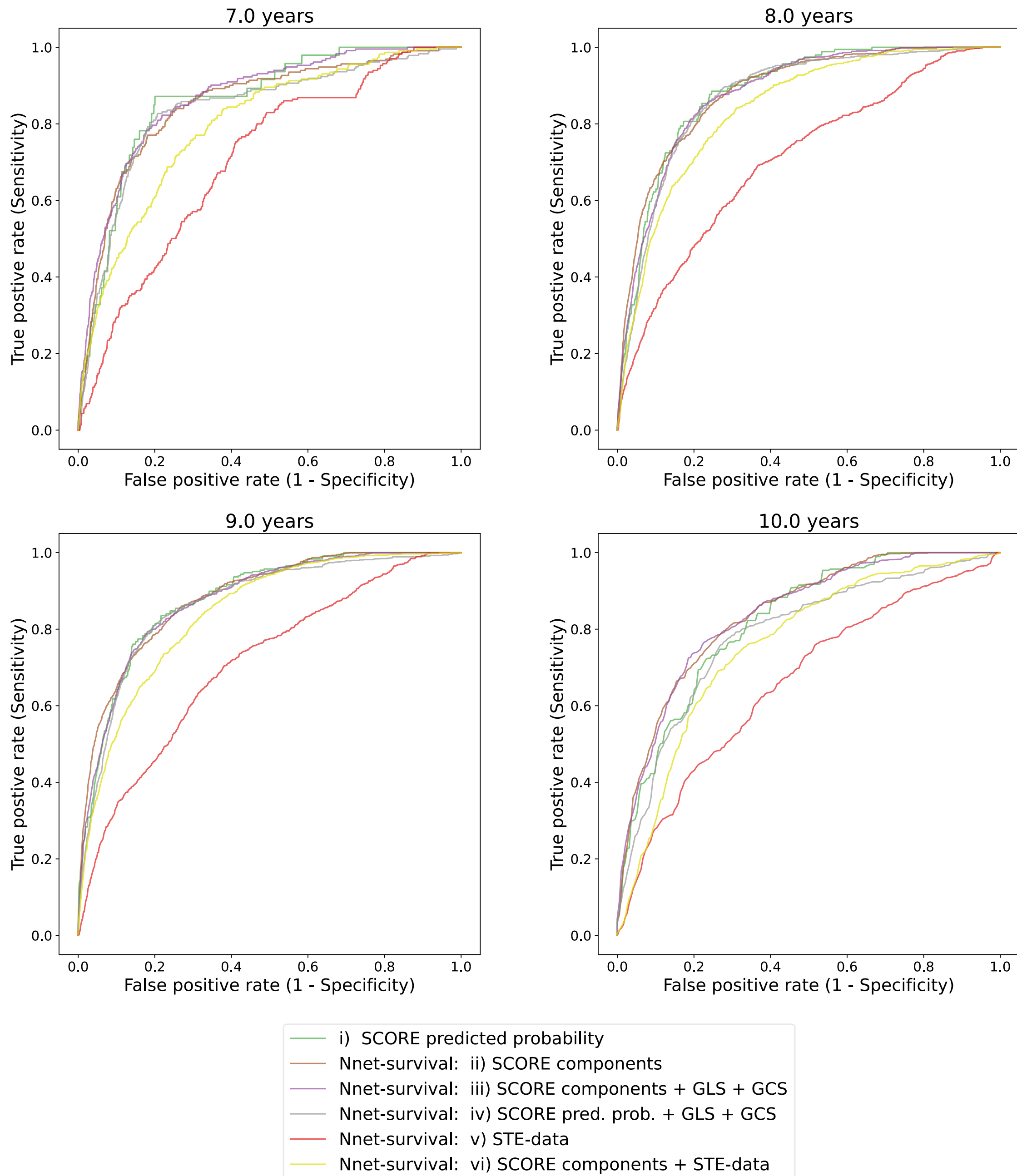

Figure 11 Receiver-operating-characteristics for the different 'nnet-survival'-models compared to SCORE risk chart reference model calculated for different points in time.
The receiver-operating-characteristic plots true positive rate against false positive rate by varying thresholds (not shown). Discrimination is best for the curve, that is closest to the left upper corner. SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

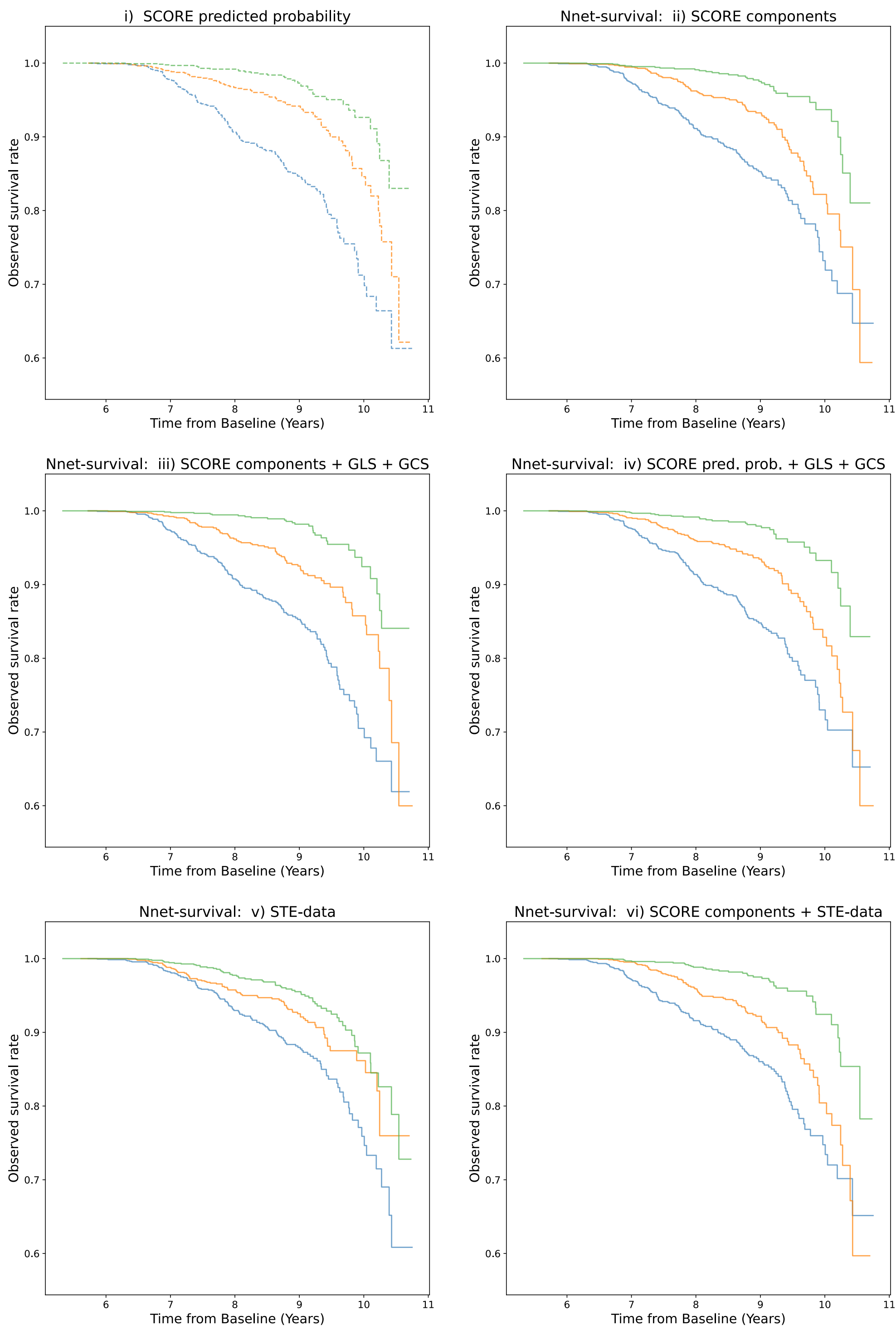

Figure 12. Kaplan-Meier survival curves by tercile of the predicted survival probability for the different 'nnet-survival'- models compared to SCORE risk chart reference model.

Kaplan-Meier curves for the respective subpopulation, separated by tercile (green upper, yellow middle, blue lower tercile) of the predicted survival probability, are plotted from 6 to 10 years after baseline (Time before 6 years not shown, since no events occurred). SCORE components = age, sex, current smoking, systolic blood pressure, serum cholesterol; GCS = global peak circumferential strain, GLS = global peak longitudinal strain, STE-data = global and segmental peak circumferential/longitudinal/radial/transversal strain (rate)

5 Discussion

In the present work, the performance of machine-learning models for predicting all-cause mortality (including a new continuous-time neural network for survival data) based on cardiovascular risk factors and data from speckle-tracking-echocardiography in a general population cohort was evaluated.

The results consistently show an excellent discrimination performance (8.5-year time-dependent AUROC $\approx$ 0.9), both for the established SCORE risk chart model i) and the neural network models ii) – iv). Discrimination of people surviving beyond or dying before a specified time reached reasonable results using STE-data only. Still, the performance was worse than for traditional cardiovascular risk factors. In terms of absolute prediction errors (i.e., discrimination and calibration), the neural network models outperformed the SCORE model. The larger the temporal distance from baseline, the more pronounced the differences were.

5.1 Global peak strain from Speckle-tracking-echocardiography

Several studies analyzed the potential diagnostic and prognostic value of global peak strain values in longitudinal, radial/transverse, and circumferential orientation. They proved independent association with (cardiovascular) outcomes in several disease entities and populations at risk[1,5,7-9]. Those studies aimed to replace LVEF, which is currently used to characterize global cardiac function for treatment recommendations according to most guidelines[59-61], with global peak strain values. LVEF is highly correlated to the peak strain parameters since the surface of an object without a large shape deformation (like the LV) is mathematically linked to its volume[62]. GLS is hence not likely to provide incremental information in terms of prediction. The differences observed in those studies may be explained by using inappropriate statistical models that do not account for non-linear relations to the investigated outcome. I.e., despite containing the same information, the relation of GLS (and GCS) to the probability of adverse events may be better modelled by a linear model on the respective scale (e.g. hazard scale). This may even be favourable if the goal was to find a replacement for LVEF. But there is still no proof of an incremental prognostic value of GLS and GCS, at least not in the general population. However, with emerging deep learning applications for automatic postprocessing of echocardiography images[63], the costs of

extracting all of those function parameters become neglectable, and the intelligent integration of the available information becomes the primary concern.

Therefore, the present study focuses on the predictive capabilities of the parameters of interest, including discrimination and calibration, and not on (independent) association[11,12,64]. Instead of merely confusing the terms “association” and “prediction”[11], both discrimination and calibration were evaluated in strictly separated training and test sets (avoiding data leakage). Also, the results’ uncertainty due to single train-test-splits (instead, repeated 10-fold cross-validation was applied) and optimistic bias (by bootstrap-bias-correction) due to testing of multiple hyperparameter configurations were minimized in this study [13,49]. Moreover, the application of the neural network approach, instead of (generalized) linear survival models, accounts for non-linear relations and interactions of all investigated variables to the survival outcome.

In this prediction framework, the results of our analysis provide no evidence for the incremental prognostic value of peak global longitudinal and circumferential strain over traditional biomarkers (SCORE components) for predicting mortality in the general population.

5.2 Speckle-tracking-echocardiography-derived multidimensional data

Quantification of regional wall motion is the main theoretical advantage of speckle tracking echocardiography over traditional volume-based (in 2D echocardiography typically estimated from end-diastolic and systolic areas in two planes according to Simpson’s biplane approach) characterization of cardiac motion. These regional abnormalities may be related to hibernating myocardium, focal fibrosis, or scars.

The novelty of this work is to consider the multiple parameters (e.g., regional peak longitudinal, transversal, radial and circumferential strain (-rate) for 16/17 myocardial segments[65]) as a multidimensional set of imaging biomarkers as it is done for grayscale and shape feature in ‘Radiomics’, gene expression in ‘Genomics’ and the protein expression in ‘Proteomics’. Potential collinearities preventing model convergence and the ‘curse of dimensionality’ in general make the analysis of multi-dimensional data challenging [66]. Therefore, special treatment in statistical analyzes is needed[15,16].

This work aimed to predict survival outcomes by comprehensively integrating the STE-data. This process includes potential interactions between predictor variables, non-linear relations to the outcome, and the shape of the predicted survival curve. Failing convergence of traditional (generalized linear) statistical models motivated the development of the supervised 'nnet-Surv-rcsplines' neural network for competing risk survival data. In the investigated general population cohort, it was possible to discriminate whether a person is more likely to survive beyond a specified point in time than another by using only information from STE-data (i.e., parameters describing the whole left ventricular cardiac motion). However, the SCORE benchmark model i) and the models including at least the same information as SCORE ii)-iv) showed significantly better discrimination (larger AUROC) and lower prediction error (higher SBS). Likely, the traditional non-imaging biomarkers (age, sex, smoking status, systolic blood pressure and serum cholesterol) contain complementary information or information necessary to model the relation to the outcome (i.e., interaction in statistical terms). Furthermore, the limited sample size may have precluded the model from learning the complex relationships of the high-dimensional input variables to the mortality outcome.

Also, for the nested model vi), including SCORE components and STE-data, no incremental prognostic value could be observed compared to the models relying on non-imaging information only (models i) and ii)).

The information on cardiac motion related to the survival probability may already implicitly be contained in sociodemographic risk factors (e.g., age and sex). In general, it is often observed that if the benchmark model itself already shows excellent performance, adding new information may yield only minor improvements[67,68].

As expected, the calibration of the SCORE risk model yielded poor results since it was the only model that was not trained (and hence not calibrated) on the training data. The actual survival up to 7 years after baseline was underestimated, and from 8 years after baseline and beyond grossly overestimated.

Whether integrating multidimensional cardiac motion parameters with other multidimensional datasets (genomics, metabolomics etc.) can improve the outcome

prediction needs to be addressed in larger cohorts with specific disease entities (e.g., myocardial infarction, heart failure), where survival outcome is more closely linked to cardiac motion [68].

5.3 'nnet-Surv-rcsplines' neural network for survival data with and without presence of competing risks

Unlike prior efforts ('DeepSurv'[29], 'Cox-nnet'[30]), which extend the semi-parametric Cox Proportional Hazard model to a neural network, the present work proposes a flexible parameterization of the transformed cumulative incidence function (CIF) using restricted cubic splines. This allows a direct estimation of the survival function of every individual. Choosing the linking coefficient of the Aranda-Ordaz link function decides on the scale at which the CIF is modelled. For example, the extreme values of $c = 0$ and $c = 1$ correspond to the hazard function and the log odds scale, respectively.

Moreover, a flexible variant also allows direct dependency of the spline parameters on the input variables or neural network weights. Hence, violations of proportionality of the data on the respective scale (e.g., proportional hazard assumption of Cox proportional hazard model) could be modelled without impaired prediction results[69]. However, due to the small sample size and resulting convergence issues, this extension was not applied to the data of this study.

As a sensitivity analysis, the implementation of the multi-task network 'nnet-survival' proposed by Gensheimer et al.[27] was updated for the usage of Tensorflow 2 and applied to the same model inputs as in the primary analysis. In fact, there were only minor differences in the model performances, given the respective model inputs. The differences were most pronounced at 10 years from baseline, which may be caused by limited training sample size and inadequate hyperparameter search space.

Furthermore, a generalization to competing risks with the same Aranda-Ordaz link was proposed. This neural network models the cause-specific CIF on the subdistribution hazard (for $c=0$, like the Fine-Gray competing risk model) and on the subdistribution odds scale (for $c = 1$). In the present work's empirical analysis, the complexity of competing risks was avoided by choosing all-cause mortality as the outcome.

The evaluation of the proposed 'nnet-Surv-rcsplines' model in the context of competing risks in a simulation study and on real biomedical datasets goes beyond the scope of this work. It remains the subject of future research. Unfortunately, publicly available large-scale biomedical datasets with time-to-event data (especially with competing risk data) are currently sparse[31]. In the methodology underlying the recently published update of the SCORE prediction models, SCORE2[25] and SCORE2-OP[26] the Fine-Gray regression[44] is applied to model the competing risk effects of non-cv-mortality. Even though first-order interactions and non-linear effects were explicitly considered, the authors reported only minor improvements in predicting cardiovascular mortality compared to the original SCORE risk chart model.

Application of modern machine learning methods like gradient-boosted trees[70], random survival forests[71], neural networks suitable for competing risk data like 'DeepHit'[31] and the proposed 'nnet-Surv-rcsplines'-model on the large-scale model derivation cohort of the SCORE project may improve the prediction of CV (and non-CV) outcomes and may help to provide better individual treatment and life-style recommendations.

Besides curative and preventive medicine, accurate prediction of competing risk outcomes is highly relevant for finance and insurance companies (especially life insurances, disability insurances, mortgage defaults etc.)[72]. Using the proposed 'nnet-Surv-rcsplines' and other deep learning methods for competing risk survival data may provide better estimates of certain risks. This could allow the insurance of so far 'uninsurable risks' and could lead to cheaper policies for costumers[73].

5.4 Study limitations

Several limitations of this study merit consideration.

Firstly, since the highest observation density occurred at 8.5 years from baseline examination, this point in time (8.5 years from baseline) was chosen to select hyperparameters. Consequently, the inferred estimates of model performance have lower variation than e.g. at 10 years. Nevertheless, this choice is arbitrary, as was the established 10-year horizon from earlier works[25,26,46].

Second, it is generally known that complex statistical models (especially neural networks) need large datasets to learn the non-linear relations of model input and output. In our study population, only 350 events occurred in 3858 participants over a median follow-up time of 8.4 years, while no fatal events occurred in the first 6 years of follow-up. Although the cohort was drawn as a stratified sample from local population registries, the number of mortality events was lower, and their distribution does not resemble the expected mortality rates in the general population based on official German mortality statistics[47]. This might be caused by healthy-volunteer bias and result in underestimated event rates if the models were deployed in the general population.

Thirdly, the tested hyperparameter configurations (including the defined search space) could have been unsuitable for solving the problem and might have led to poor optimization results[74].

Further, too high computational expenses related to repeated training in the complex model training and evaluation framework precluded using the rather robust random survival forest[71], which is not a deep learning but a conventional machine learning algorithm using ensemble learning[75].

6 Conclusion

Regional myocardial strain distribution contains prognostic information for predicting all-cause mortality in primary prevention. Still, no prognostic value in addition to readily available traditional clinical and laboratory biomarkers was demonstrated. Moreover, in contrast to prior studies, no incremental predictive value was found for global circumferential and longitudinal strain. This may be explained by the effective integration of the available traditional biomarkers and the strict evaluation in a prediction framework that eliminates several sources of optimistic bias.

The application of neural networks for survival (and competing risks) data, like the flexible parametric 'nnet-Surv-rcsplines' neural network proposed in this work, may improve outcome prediction in primary prevention by exhaustively integrating the available information, including complex non-linear and interactive relations of the outcome to the respective input data. With the emerging availability of large-scale datasets ('BigData') from observational studies and linked electronic health records, this approach might improve treatment decisions compared to standard statistical methods and should be further investigated.

7 References

1. Biering-Sorensen T, Biering-Sorensen SR, Olsen FJ, et al. Global Longitudinal Strain by Echocardiography Predicts Long-Term Risk of Cardiovascular Morbidity and Mortality in a Low-Risk General Population: The Copenhagen City Heart Study. *Circ Cardiovasc Imaging*. 2017;10(3). doi:10.1161/CIRCIMAGING.116.005521.
2. Chang W-T, Lee W-H, Lee W-T, et al. Left ventricular global longitudinal strain is independently associated with mortality in septic shock patients. *Intensive Care Med*. 2015;41(10):1791-1799. doi:10.1007/s00134-015-3970-3.
3. Kuznetsova T, Cauwenberghs N, Knez J, et al. Additive Prognostic Value of Left Ventricular Systolic Dysfunction in a Population-Based Cohort. *Circ Cardiovasc Imaging*. 2016;9(7). doi:10.1161/CIRCIMAGING.116.004661.
4. Kosmala W, Rojek A, Przewlocka-Kosmala M, Mysiak A, Karolko B, Marwick TH. Contributions of Nondiastolic Factors to Exercise Intolerance in Heart Failure With Preserved Ejection Fraction. *J Am Coll Cardiol*. 2016;67(6):659-670. doi:10.1016/j.jacc.2015.10.096.
5. Bertini M, Ng ACT, Antoni ML, et al. Global longitudinal strain predicts long-term survival in patients with chronic ischemic cardiomyopathy. *Circ Cardiovasc Imaging*. 2012;5(3):383-391. doi:10.1161/CIRCIMAGING.111.970434.
6. Buggey J, Alenezi F, Yoon HJ, et al. Left ventricular global longitudinal strain in patients with heart failure with preserved ejection fraction: outcomes following an acute heart failure hospitalization. *ESC Heart Fail*. 2017;4(4):432-439. doi:10.1002/ehf2.12159.
7. Ersboll M, Valeur N, Mogensen UM, et al. Prediction of all-cause mortality and heart failure admissions from global left ventricular longitudinal strain in patients with acute myocardial infarction and preserved left ventricular ejection fraction. *J Am Coll Cardiol*. 2013;61(23):2365-2373. doi:10.1016/j.jacc.2013.02.061.
8. Ng ACT, Prihadi EA, Antoni ML, et al. Left ventricular global longitudinal strain is predictive of all-cause mortality independent of aortic stenosis severity and ejection fraction. *Eur Heart J Cardiovasc Imaging*. 2018;19(8):859-867. doi:10.1093/ehjci/jex189.

9. Modin D, Sengelov M, Jorgensen PG, et al. Global longitudinal strain corrected by RR interval is a superior predictor of all-cause mortality in patients with systolic heart failure and atrial fibrillation. *ESC Heart Fail*. 2018;5(2):311-318. doi:10.1002/ehf2.12220.
10. Kramann R, Erpenbeck J, Schneider RK, et al. Speckle tracking echocardiography detects uremic cardiomyopathy early and predicts cardiovascular mortality in ESRD. *J Am Soc Nephrol*. 2014;25(10):2351-2365. doi:10.1681/ASN.2013070734.
11. Varga TV, Niss K, Estampador AC, Collin CB, Moseley PL. Association is not prediction: A landscape of confused reporting in diabetes - A systematic review. *Diabetes Res Clin Pract*. 2020;170:108497. doi:10.1016/j.diabres.2020.108497.
12. Steyerberg EW. *CLINICAL PREDICTION MODELS: A practical approach to development, validation, and updating*. [Place of publication not identified]: SPRINGER NATURE; 2019.
13. Pencina MJ, Goldstein BA, D'Agostino RB. Prediction Models - Development, Evaluation, and Clinical Application. *N Engl J Med*. 2020;382(17):1583-1586. doi:10.1056/NEJMp2000589.
14. Euler A, Laqua FC, Cester D, et al. Virtual Monoenergetic Images of Dual-Energy CT-Impact on Repeatability, Reproducibility, and Classification in Radiomics. *Cancers (Basel)*. 2021;13(18). doi:10.3390/cancers13184710.
15. Mühlbauer J, Egen L, Kowalewski K-F, et al. Radiomics in Renal Cell Carcinoma-A Systematic Review and Meta-Analysis. *Cancers (Basel)*. 2021;13(6). doi:10.3390/cancers13061348.
16. Mühlbauer J, Kriegmair MC, Schöning L, et al. Value of Radiomics of Perinephric Fat for Prediction of Intraoperative Complexity in Renal Tumor Surgery. *Urol Int*. 2021:1-12. doi:10.1159/000520445.
17. Tokodi M, Schwertner WR, Kovács A, et al. Machine learning-based mortality prediction of patients undergoing cardiac resynchronization therapy: the SEMMELWEIS-CRT score. *Eur Heart J*. 2020;41(18):1747-1756. doi:10.1093/eurheartj/ehz902.
18. van Assen M, Martin SS, Varga-Szemes A, et al. Automatic coronary calcium scoring in chest CT using a deep neural network in direct comparison with non-contrast cardiac CT: A validation study. *Eur J Radiol*. 2021;134:109428. doi:10.1016/j.ejrad.2020.109428.

19. Sandino CM, Lai P, Vasanawala SS, Cheng JY. Accelerating cardiac cine MRI using a deep learning-based ESPIRiT reconstruction. *Magn Reson Med*. 2021;85(1):152-167. doi:10.1002/mrm.28420.
20. Hann E, Popescu IA, Zhang Q, et al. Deep neural network ensemble for on-the-fly quality control-driven segmentation of cardiac MRI T1 mapping. *Med Image Anal*. 2021;71:102029. doi:10.1016/j.media.2021.102029.
21. Tedesco S, Andrulli M, Larsson MÅ, et al. Comparison of Machine Learning Techniques for Mortality Prediction in a Prospective Cohort of Older Adults. *Int J Environ Res Public Health*. 2021;18(23). doi:10.3390/ijerph182312806.
22. Thorsteinsdottir B, Hickson LJ, Giblon R, et al. Validation of prognostic indices for short term mortality in an incident dialysis population of older adults 75. *PLoS One*. 2021;16(1):e0244081. doi:10.1371/journal.pone.0244081.
23. Weng SF, Vaz L, Qureshi N, Kai J. Prediction of premature all-cause mortality: A prospective general population cohort study comparing machine-learning and standard epidemiological approaches. *PLoS One*. 2019;14(3):e0214365. doi:10.1371/journal.pone.0214365.
24. Kodama S, Saito K, Tanaka S, et al. Cardiorespiratory fitness as a quantitative predictor of all-cause mortality and cardiovascular events in healthy men and women: a meta-analysis. *JAMA*. 2009;301(19):2024-2035. doi:10.1001/jama.2009.681.
25. SCORE2 risk prediction algorithms: new models to estimate 10-year risk of cardiovascular disease in Europe. *Eur Heart J*. 2021;42(25):2439-2454. doi:10.1093/eurheartj/ehab309.
26. SCORE2-OP risk prediction algorithms: estimating incident cardiovascular event risk in older persons in four geographical risk regions. *Eur Heart J*. 2021;42(25):2455-2467. doi:10.1093/eurheartj/ehab312.
27. Gensheimer MF, Narasimhan B. A scalable discrete-time survival model for neural networks. *PeerJ*. 2019;7:e6257. doi:10.7717/peerj.6257.
28. Gerds TA, Schumacher M. Consistent estimation of the expected Brier score in general survival models with right-censored event times. *Biom J*. 2006;48(6):1029-1040. doi:10.1002/bimj.200610301.

29. Katzman JL, Shaham U, Cloninger A, Bates J, Jiang T, Kluger Y. DeepSurv: personalized treatment recommender system using a Cox proportional hazards deep neural network. *BMC Med Res Methodol*. 2018;18(1):24. doi:10.1186/s12874-018-0482-1.
30. Ching T, Zhu X, Garmire LX. Cox-nnet: An artificial neural network method for prognosis prediction of high-throughput omics data. *PLoS Comput Biol*. 2018;14(4):e1006076. doi:10.1371/journal.pcbi.1006076.
31. Ryu JY, Lee MY, Lee JH, Lee BH, Oh K-S. DeepHIT: a deep learning framework for prediction of hERG-induced cardiotoxicity. *Bioinformatics*. 2020;36(10):3049-3055. doi:10.1093/bioinformatics/btaa075.
32. Royston P, Parmar MKB. Flexible parametric proportional-hazards and proportional-odds models for censored survival data, with application to prognostic modelling and estimation of treatment effects. *Statist Med*. 2002;21(15):2175-2197. doi:10.1002/sim.1203.
33. Ng R, Kornas K, Sutradhar R, Wodchis WP, Rosella LC. The current application of the Royston-Parmar model for prognostic modeling in health research: a scoping review. *Diagn Progn Res*. 2018;2:4. doi:10.1186/s41512-018-0026-5.
34. Moons KGM, Altman DG, Reitsma JB, et al. Transparent Reporting of a multivariable prediction model for Individual Prognosis or Diagnosis (TRIPOD): explanation and elaboration. *Ann Intern Med*. 2015;162(1):W1-73. doi:10.7326/M14-0698.
35. Liu X, Cruz Rivera S, Moher D, Calvert MJ, Denniston AK. Reporting guidelines for clinical trial reports for interventions involving artificial intelligence: the CONSORT-AI extension. *Nat Med*. 2020;26(9):1364-1374. doi:10.1038/s41591-020-1034-x.
36. Volzke H, Alte D, Schmidt CO, et al. Cohort profile: the study of health in Pomerania. *Int J Epidemiol*. 2011;40(2):294-307. doi:10.1093/ije/dyp394.
37. Lang RM, Bierig M, Devereux RB, et al. Recommendations for chamber quantification: A report from the American Society of Echocardiography's Guidelines and Standards Committee and the Chamber Quantification Writing Group, developed in conjunction with the European Association of Echocardiography, a branch of the European Society of Cardiology. *J Am Soc Echocardiogr*. 2005;18(12):1440-1463. doi:10.1016/j.echo.2005.10.005.

38. Lang RM, Badano LP, Mor-Avi V, et al. Recommendations for cardiac chamber quantification by echocardiography in adults: an update from the American Society of Echocardiography and the European Association of Cardiovascular Imaging. *J Am Soc Echocardiogr*. 2015;28(1):1-39.e14. doi:10.1016/j.echo.2014.10.003.
39. Stekhoven DJ, Bühlmann P. MissForest--non-parametric missing value imputation for mixed-type data. *Bioinformatics*. 2012;28(1):112-118. doi:10.1093/bioinformatics/btr597.
40. Nair V, Hinton GE. Rectified Linear Units Improve Restricted Boltzmann Machines. In: Fürnkranz J, Joachims T, eds. Proceedings of the 27th International Conference on Machine Learning (ICML-10); 2010:807-814.
41. Ioffe S, Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: Bach F, Blei D, eds. Proceedings of the 32nd International Conference on Machine Learning. Vol. 37. Lille, France: PMLR; 2015:448-456. Proceedings of Machine Learning Research.
42. Kingma DP, Ba J. *Adam: A Method for Stochastic Optimization*; 2014. https://arxiv.org/pdf/1412.6980.
43. Hastie T, Tibshirani R, Friedman J. *The Elements of Statistical Learning*. New York, NY: Springer New York; 2009.
44. Fine JP, Gray RJ. A Proportional Hazards Model for the Subdistribution of a Competing Risk. *Journal of the American Statistical Association*. 1999;94(446):496. doi:10.2307/2670170.
45. Jeong J-H, Fine JP. Parametric regression on cumulative incidence function. *Biostatistics*. 2007;8(2):184-196. doi:10.1093/biostatistics/kxj040.
46. Conroy R. Estimation of ten-year risk of fatal cardiovascular disease in Europe: the SCORE project. *Eur Heart J*. 2003;24(11):987-1003. doi:10.1016/s0195-668x(03)00114-3.
47. Deutsches Statistisches Bundesamt. Todesursachenstatistik 2010; 23211-0004: Gestorbene: Deutschland, Jahre, Todesursachen, Geschlecht, Altersgruppen. *Fachserie Statistisches Bundesamt / 12 / 4 / Jährlich*. 2012;(2120400107005). https://www.statistischebibliothek.de/mir/receive/DEHeft_mods_00018825.

48. Steyerberg EW, Vickers AJ, Cook NR, et al. Assessing the performance of prediction models: a framework for some traditional and novel measures. *Epidemiology*. 2010;21(1):128-138. doi:10.1097/EDE.0b013e3181c30fb2.
49. Tsamardinos I, Greasidou E, Borboudakis G. Bootstrapping the out-of-sample predictions for efficient and accurate cross-validation. *Mach Learn*. 2018;107(12):1895-1922. doi:10.1007/s10994-018-5714-4.
50. Lambert J, Chevret S. Summary measure of discrimination in survival models based on cumulative/dynamic time-dependent ROC curves. *Stat Methods Med Res*. 2016;25(5):2088-2102. doi:10.1177/0962280213515571.
51. Gerds TA, Andersen PK, Kattan MW. Calibration plots for risk prediction models in the presence of competing risks. *Stat Med*. 2014;33(18):3191-3203. doi:10.1002/sim.6152.
52. Austin PC, Steyerberg EW. Graphical assessment of internal and external calibration of logistic regression models by using loess smoothers. *Stat Med*. 2014;33(3):517-535. doi:10.1002/sim.5941.
53. Graf E, Schmoor C, Sauerbrei W, Schumacher M. Assessment and comparison of prognostic classification schemes for survival data. *Stat Med*. 1999;18(17-18):2529-2545. doi:10.1002/(sici)1097-0258(19990915/30)18:17/18<2529:aid-sim274>3.0.co;2-5.
54. Rothman KJ. No adjustments are needed for multiple comparisons. *Epidemiology*. 1990;1(1):43-46.
55. *TensorFlow*. Zenodo; 2021.
56. Davidson-Pilon C. lifelines: survival analysis in Python. *Journal of Open Source Software*. 2019;4(40):1317.
57. *scikit-survival*. Zenodo; 2020.
58. Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011;12:2825-2830.
59. Ponikowski P, Voors AA, Anker SD, et al. 2016 ESC Guidelines for the diagnosis and treatment of acute and chronic heart failure: The Task Force for the diagnosis and treatment of acute and chronic heart failure of the European Society of Cardiology (ESC)Developed with the special contribution of the Heart Failure Association (HFA) of the ESC. *Eur Heart J*. 2016;37(27):2129-2200. doi:10.1093/eurheartj/ehw128.

60. Maddox TM, Januzzi JL, Allen LA, et al. 2021 Update to the 2017 ACC Expert Consensus Decision Pathway for Optimization of Heart Failure Treatment: Answers to 10 Pivotal Issues About Heart Failure With Reduced Ejection Fraction: A Report of the American College of Cardiology Solution Set Oversight Committee. *J Am Coll Cardiol*. 2021;77(6):772-810. doi:10.1016/j.jacc.2020.11.022.
61. Yancy CW, Jessup M, Bozkurt B, et al. 2017 ACC/AHA/HFSA Focused Update of the 2013 ACCF/AHA Guideline for the Management of Heart Failure: A Report of the American College of Cardiology/American Heart Association Task Force on Clinical Practice Guidelines and the Heart Failure Society of America. *Circulation*. 2017;136(6):e137-e161. doi:10.1161/CIR.0000000000000509.
62. Delgado V, Mollema SA, Ypenburg C, et al. Relation between global left ventricular longitudinal strain assessed with novel automated function imaging and biplane left ventricular ejection fraction in patients with coronary artery disease. *J Am Soc Echocardiogr*. 2008;21(11):1244-1250. doi:10.1016/j.echo.2008.08.010.
63. Salte IM, Østvik A, Smistad E, et al. Artificial Intelligence for Automatic Measurement of Left Ventricular Strain in Echocardiography. *JACC Cardiovasc Imaging*. 2021;14(10):1918-1928. doi:10.1016/j.jcmg.2021.04.018.
64. Wolff RF, Moons KGM, Riley RD, et al. PROBAST: A Tool to Assess the Risk of Bias and Applicability of Prediction Model Studies. *Ann Intern Med*. 2019;170(1):51-58. doi:10.7326/M18-1376.
65. Cerqueira MD, Weissman NJ, Dilsizian V, et al. Standardized myocardial segmentation and nomenclature for tomographic imaging of the heart. A statement for healthcare professionals from the Cardiac Imaging Committee of the Council on Clinical Cardiology of the American Heart Association. *Circulation*. 2002;105(4):539-542. doi:10.1161/hc0402.102975.
66. Chandrasekaran B, Jain AK. Quantization Complexity and Independent Measurements. *IEEE Trans Comput*. 1974;C-23(1):102-106. doi:10.1109/T-C.1974.223789.
67. Engelhard MM, Navar AM, Pencina MJ. Incremental Benefits of Machine Learning-When Do We Need a Better Mousetrap? *JAMA Cardiol*. 2021;6(6):621-623. doi:10.1001/jamacardio.2021.0139.

68. Khera R, Haimovich J, Hurley NC, et al. Use of Machine Learning Models to Predict Death After Acute Myocardial Infarction. *JAMA Cardiol*. 2021;6(6):633-641. doi:10.1001/jamacardio.2021.0122.
69. Stensrud MJ, Hernán MA. Why Test for Proportional Hazards? *JAMA*. 2020. doi:10.1001/jama.2020.1267.
70. Friedman JH. Stochastic gradient boosting. *Computational Statistics & Data Analysis*. 2002;38(4):367-378. doi:10.1016/S0167-9473(01)00065-2.
71. Ishwaran H, Kogalur UB, Blackstone EH, Lauer MS. Random survival forests. *Ann Appl Stat*. 2008;2(3):841-860. doi:10.1214/08-AOAS169.
72. Blumenstock G, Lessmann S, Seow H-V. Deep learning for survival and competing risk modelling. *Journal of the Operational Research Society*. 2022;73(1):26-38. doi:10.1080/01605682.2020.1838960.
73. Gerds TA, Andersen PK, Kattan MW. Calibration plots for risk prediction models in the presence of competing risks. *Stat Med*. 2014;33(18):3191-3203. doi:10.1002/sim.6152.
74. Bischl B, Binder M, Lang M, et al. *Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges*; 2021. https://arxiv.org/pdf/2107.05847.
75. Herrera VM, Khoshgoftaar TM, Villanustre F, Furht B. Random forest implementation and optimization for Big Data analytics on LexisNexis's high performance computing cluster platform. *J Big Data*. 2019;6(1). doi:10.1186/s40537-019-0232-1.
76. Mozumder SI, Rutherford M, Lambert P. Direct likelihood inference on the cause-specific cumulative incidence function: A flexible parametric regression modelling approach. *Stat Med*. 2018;37(1):82-97. doi:10.1002/sim.7498.
77. ARANDA-ORDAZ FJ. 'On two families of transformations to additivity for binary response data'. *Biometrika*. 1983;70(1):303. doi:10.1093/biomet/70.1.303-a.
78. Chambers RL, Steel DG, Wang S, Welsh A. *Maximum Likelihood Estimation for Sample Surveys*. Boca Raton, London, New York: Chapman and Hall/CRC; 2012. Monographs on statistics and applied probability; 125. https://www.taylorfrancis.com/books/9780429144721.
79. Rossi RJ. *Mathematical Statistics: An introduction to likelihood based inference.* 1st edition. Hoboken, NJ, USA: John Wiley & Sons, Inc; 2018. https://onlinelibrary.wiley.com/doi/book/10.1002/9781118771075.

8 Appendix A

8.1 Restricted cubic splines neural network ‘nnet-Surv-rcsplines’

Cause-specific cumulative incidence function and subdistribution hazard

Let T be the time to event for any of K competing causes $k = 1, \cdots, K$ and D denote the type of event, where $D = 1, \cdots, K$. Then the cumulative incidence function (CIF) $F_k(t)$ is the cumulative probability of dying before or at time t from cause k.

$$F_k(t) = P(T \leq t, D = k)$$

Gray[44] introduces the subdistribution hazard (SDH) for cause k, $h_k^{sd}(t)$, which gives a direct relationship with the cause-specific CIF and has the following mathematical formulation[76]:

$$h_k^{sd}(t) = \lim_{\Delta t \to \infty} \frac{P(t < T \leq t + \Delta t \cap D = k | T > t \cup (t < T \leq t + \Delta t \cap D \neq k))}{\Delta t}$$

$$= \frac{d}{dt} - \ln(1 - F_k(t))$$

$$1 - F_k(t) = \exp\left(-H_k^{sd}(t)\right) = \exp\left[\int_0^t h_k^{sd}(u)du\right]$$

where $H_k^{sd}(t)$ is the cumulative subdistribution hazard. In the present work, this approach is generalized to a neural network implementation.

Regression modelling

Similar to the original implementation proposed by Royston and Parmar[32,33] the cause-specific CIF is modelled by a restricted cubic spline ($s(\ln(t), \boldsymbol{\gamma}_\mathbf{k}, \mathbf{m}_k)$) on the logarithmic time scale connected by a link function $g(\cdot)$, where $\boldsymbol{\gamma}_\mathbf{k}$ is the vector of the spline coefficients and $\mathbf{m}_\mathbf{k}$ is the vector of the knot positions of the spline. Restricted cubic splines are piecewise cubic polynomials with $l \geq 0$ internal knots defined to have continuous first and second derivatives at the internal knots and constrained to be linear beyond boundary knots k_{min}, k_{max} [32]:

$$s(\ln t, \boldsymbol{\gamma}_\mathbf{k}, \mathbf{m}_k)\gamma_{k,0} + \gamma_{k,1}\, ln\, t + \gamma_2 v_{k,1}(ln\, t) + \cdots + \gamma_{k,l+1} v_{k,l}(ln\, t),$$

where $v_{k,j}(x) = (x - m_{k,j})_+^3 - \lambda_{k,j}(x - m_{k,max})_+^3 - (1 - \lambda_{k,j})(x - m_{k,max})_+^3$,

$\lambda_{k,j} = \frac{m_{k,max} - m_{k,j}}{m_{k,max} - m_{k,min}}$ and $(x - a)+ = \max(0, x - a)$.

Like Royston and Parmar propose[32], the Aranda-Ordaz-link[77] is used.

$$g(1 - F_k(t)) = \eta_k(t|\mathbf{x}) = s(\ln(t), \boldsymbol{\gamma}_\mathbf{k}, \mathbf{m}_\mathbf{k}) + \mathbf{x}_\mathbf{k}\boldsymbol{\beta}_\mathbf{k}$$

$$g(\cdot) = \ln\left\{\frac{(\cdot)^{-c} - 1}{c}\right\};\ c \in (0,\ 1]$$

Note that the notation $\mathbf{x}_k\boldsymbol{\beta}_k$ is similar to standard generalized linear model regression models, where it would describe a linear effect of the model covariables on the respective linked scale. In our work, this term is replaced by the output of a neural network, or more specifically, the output of a Dense Layer with linear activation.

With the Aranda-Ordaz-Link, the cause-specific CIF and consequently the subdistribution hazard or odds (naming and interpretation are dependent on the hyperparameter in the link function, cf. below) of the respective cause *k* is parameterized as:

$$F_k(t) = 1 - (1 + ce^{\eta_k(t)})^{\frac{-1}{c}}$$

$$h_k^{sd}(t) = \frac{d}{dt} - \ln(1 - F_k(t)) = (1 + ce^{\eta_k(t)})^{-1} e^{\eta_k(t)} \frac{d\eta_k(t)}{dt}$$

The Aranda-Ordaz-link family comprises of two special cases: If *c = 1* this simplifies to logit-link:

$\eta_k(t) = \ln\left(\frac{F_k(t)}{1 - F_k(t)}\right)$ and $(1 + ce^{\eta_k(t)})^{\frac{-1}{c}} \overset{c:=1}{=} (1 + e^{\eta_k(t)})^{-1}$

If the shape of $s(\ln(t), \boldsymbol{\gamma}_k, \mathbf{m}_k)$ is independent from model covariates (i.e., in this neural network there is no direct or indirect connection between Input Layer and the 'Gamma-Layer'), the model formulation can be interpreted as a proportional odds model given the output of the dense 'Beta-Layer' $\mathbf{x}_\mathbf{k}\boldsymbol{\beta}_\mathbf{k}$.

And for $c \to 0$ this converges towards the complimentary log-log-link and simplifies to a proportional hazard interpretation:

$$\lim_{c\to 0}\eta_k(t) = \lim_{c\to 0} \ln\left\{\frac{(1-F_k(t))^{-c}-1}{c}\right\} = \ln \lim_{c\to 0}\frac{-\ln(1-F_k(t))(1-F_k(t))^{-c}}{1}$$

$= \ln\{-\ln(1-F_k(t))\}$ and $1-F_k(t) = e^{-e^{\eta_k(t)}}$

In the original work of Royston and Parmar[32], the models were restricted to logit- and complimentary log-log links for easy interpretation. However, since in prediction modeling (especially with high dimensional inputs) the purpose is not the interpretability of the model, but to provide accurate estimations of the outcome, the coefficient c is treated as hyperparameter, that may vary in (0, 1]. The log-log-link can be specified by setting $c = 0$ in the implementation.

Negative Log-Likelihood-Loss for competing risk data

The proposed model is optimized using maximum-likelihood-estimation[78,79]. Hence, the model parameters are selected in a way, that the observed data is most probable (i.e., maximization of a likelihood function) under the statistical model. The authors of [45] give the likelihood for the CIF:

$$L = \prod_{i=1}^{N}\left[\left[\prod_{j=1}^{K}\left[h_j^{sd}(t_i)\left(1-F_j(t_i|\mathbf{x}_i)\right)\right]^{\delta_{ij}}\right]\left[1-\sum_{j=1}^{K}F_j\,(t_i|\mathbf{x}_i)\right]^{1-\sum_{j=1}^{K}\delta_{ij}}\right]$$

Since the natural logarithm is a strictly monotonically increasing function, the maximum of the natural logarithm of a function occurs at the same values as for its argument. However, this offers numerical advantages for practical computation.

In machine learning (incl. deep learning), a loss is minimized instead of maximized by convention. Negative Log-likelihood for an individual *i* is hence

$$-\ln(L_i) = -\sum_{j=1}^{K} \underbrace{\left[\ln(h_j^{sd}(t_i|\mathbf{x}_i)) + \ln(1 - F_j(t_i|\mathbf{x}_i))\right]^{\delta_{ij}}}_{failure\ from\ cause\ j}$$

$$\underbrace{-\ln\left(\left[1 - \sum_{j=1}^{K} F_j\,(t_i|\mathbf{x}_i)\right]^{1-\sum_{j=1}^{K}\delta_{ij}}\right)}_{right-censored}$$

With the parameterization, this results in the case of failure by cause *j* for subject *i*:

$$\ln(L_{i,j;\delta_{ij}=1}(t_i|\mathbf{x}_i)) = \left[\left(-1 - \frac{1}{c}\right)\ln\left(1 + ce^{\eta_j(t_i|\mathbf{x}_i)}\right) + \eta_j(t_i|\mathbf{x}_i) + \ln\left(\frac{d\eta_j(t_i|\mathbf{x}_i)}{dt_i}\right)\right]$$

and in the case of a right-censored subject i:

$$\ln\left(L_{i,\delta_{ij}=0\ \forall j=1,\ldots,K}(t_i|\mathbf{x}_i)\right) = \ln\left(\left[1 + \sum_{j=1}^{K}\left(1 + ce^{\eta_j(t_i|\mathbf{x}_i)}\right)^{\frac{-1}{c}} - 1\right]\right)$$

The argument of the rightmost logarithm in the uncensored log-likelihood needs to be greater than 0 to be well-defined. This corresponds to a strictly monotonically increasing spline function given the data and is addressed by adding a high L2 penalty for negative arguments:

$$p_{\frac{d\eta}{dt}} = \lambda \cdot \sqrt{\sum_{i=1}^{N} R\,ELU(-\frac{ds(t_i|\mathbf{x}_i)}{dln(t)})^2}$$

with $RELU(a) = max(0, a)$.

In addition, the global survival function (the argument term for the log-likelihood in the case of right-censoring) was penalized in the same way. Note that this restriction is automatically fulfilled for K=1 (’usual’ survival modeling without competing risks).

$$p_{1-\sum_{j=1}^{K} F_j(t)} = \lambda \cdot \sqrt{\sum_{i=1}^{N} R\,ELU\left(-1 + \sum_{j=1}^{K} F_j\,(t_i|\mathbf{x}_i)\right)^2}$$

The total negative-log-likelihood-loss function that is to be minimized during training is given by:

$$nll = \sum_{i=1}^{N}\left[-\left[\sum_{j=1}^{K}\ln\left(L_{i,j}(t_i|\mathbf{x}_i)\right)\right] - \ln\left(L_{i,\delta_{ij}=0\;\forall j=1,\ldots,K}(t_i|\mathbf{x}_i)\right)\right] + p_{\frac{d\eta}{dt}} + p_{1-\sum_{j=1}^{K}F_j(t)}$$

This loss function can be optimized with the favourable mini-batch stochastic gradient decent by replacing *N* with the respective batch size. In the present implementation, the adaptive momentum[42] flavour of the stochastic gradient decent optimization procedure is used.

Special case Survival data without competing risks

If there is only one cause of failure and competing risks are not to be taken into account (e.g., when only all-cause-mortality or a nested composite endpoint like major adverse cardiac events including all-cause-mortality is of interest), the above model simplifies substantially due to the simple relation of the CIF and the survival function:

$$S(t) = 1 - F(t) = \exp(-H(t)) = \exp\left[\int_0^t h(u)du\right]$$

where H is the cumulative hazard in the case of complimentary log-log-link.

The CIF and hence also the survival function is then parameterized as:

$$g(S(t)) = g(1 - F(t)) = \eta(t|\mathbf{x}) = s(\ln(t), \boldsymbol{\gamma}, \mathbf{m}) + \mathbf{x}\boldsymbol{\beta}$$

$$nll = \sum_{i=1}^{N}\left[-\left[\ln(L_{i,\delta_i=1}(t_i|\mathbf{x}_i))\right] - \ln\left(L_{i,\delta_i=0}(t_i|\mathbf{x}_i)\right)\right] + p_{\frac{d\eta}{dt}}$$

with the terms for an uncensored and a right-censored subject, respectively.

$$\ln(L_{i,\delta_i=1}(t_i|\mathbf{x}_i)) = \left[\left(-1-\frac{1}{c}\right)\ln\left(1+ce^{\eta(t_i|\mathbf{x}_i)}\right) + \eta(t_i|\mathbf{x}_i) + \ln\left(\frac{d\eta(t_i|\mathbf{x}_i)}{dt_i}\right)\right]$$

$$\ln\left(L_{i,\delta_i=0}(t_i|\mathbf{x}_i)\right) = \ln\left(\left[\left(1+ce^{\eta(t_i|\mathbf{x}_i)}\right)^{\frac{-1}{c}}\right]\right)$$

8.2 Source code of the Tensorflow 2 Implementation of 'nnet-Surv-rcsplines'

```
# -*- coding: utf-8 -*-
"""

@author: Fabian Laqua
github.com/laqua-stack
"""

from sklearn.base import BaseEstimator
from sklearn.model_selection import train_test_split
from sksurv.util import check_y_survival
import numpy as np
from numpy.lib import recfunctions as rfn
import math
from scipy.stats.mstats import mquantiles
from itertools import compress, chain

# Model
import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Add, Layer, Concatenate, Input, Dense, Dropout,
Activation, BatchNormalization
from tensorflow.keras.regularizers import l1_l2
from tensorflow.keras.initializers import he_normal
from tensorflow.python.keras import regularizers
# training
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, Callback,
LearningRateScheduler

from tensorflow.keras.optimizers import Adam


class BiasLayer(tf.keras.layers.Layer):
    def __init__(self, units=1, *args, **kwargs):
        self.units = units
        super(BiasLayer, self).__init__(*args, **kwargs)

    def build(self, input_shape):
        self.bias = self.add_weight('bias',
                                    shape=self.units,
                                    initializer='zeros',
                                    trainable=True)

    def call(self, x, **kwargs):
        return self.bias


class FinalLayer(Layer):
    def __init__(self,
                 initializer=he_normal(),
                 **kwargs
                 ):
        self.initializer = initializer
        super(FinalLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        self.inputshape = input_shape
        self.beta = self.add_weight('beta', shape=input_shape[1:],
                                    initializer=self.initializer,
                                    trainable=True)
        self.zerobeta = tf.zeros(shape=input_shape[1:], )
        super(FinalLayer, self).build(input_shape)

    def compute_output_shape(self, inputShape):
```

```
        return None, 1

    def call(self, inputs, training=None):
        betax = tf.cond(training,
                        lambda: tf.matmul(inputs, self.beta),
                        lambda: tf.matmul(inputs, self.zerobeta))

        return betax


class FinalLayerGamma(Layer):
    def __init__(self, int_knots_number=1, flexgamma=False,
                 kernel_regularizer=None,
                 **kwargs):
        self.flexgamma = flexgamma
        self.int_knots_number = int_knots_number
        self.kernel_regularizer = regularizers.get(kernel_regularizer)
        super(FinalLayerGamma, self).__init__(
            **kwargs)

    def build(self, input_shape):
        #        weight_shape = (input_shape[1:], ) if not
isinstance(input_shape[1:], tuple) else input_shape[1:]
        self.inputshape = input_shape
        if not self.flexgamma:
            self.gammaweight = tf.zeros(input_shape[1:] + (1 +
self.int_knots_number,), dtype=tf.dtypes.float32,
                                        name=None)
            self.gammabias = self.add_weight('gamma',
                                             shape=(1 + self.int_knots_number,),
                                             initializer=he_normal(),
                                             trainable=True)
        else:
            self.gammaweight = self.add_weight('gamma_weight',
                                               shape=input_shape[1:] + (1 +
self.int_knots_number,),

                                               initializer='glorot_normal',
                                               regularizer=self.kernel_regularizer,
                                               trainable=True)
            self.gammabias = self.add_weight('gamma_bias',
                                             shape=(1 + self.int_knots_number,),
                                             initializer='glorot_normal',
                                             trainable=True)

        super(FinalLayerGamma, self).build(input_shape)

    def compute_output_shape(self, inputShape):
        return (None, 1 + self.int_knots_number)

    def call(self, inputs, **kwargs):
        ret = tf.matmul(inputs, self.gammaweight) + self.gammabias
        return ret


def get_knots(m,
              y_train=None,
              kint=None,
              kmin=None,
              kmax=None
              ):
    """
    m number of interior knots,
    if kint kmin and kmax are given, y_train is not needed
    otherwise default quantile log scale knot placment is used according
    to uncensored failure times in y_train
    """
```

```
    if kint is None or kmin is None or kmax is None:
        #        print('assuming default knot placment')
        if y_train is None:
            print('y_train must not be None for default knot placement')
            raise TypeError
        y_n = y_train
        y_uncensored = y_n[:, 1][y_n[:, 0] == 1.]
        # compute quantiles of uncensored follow-up times
        quant = [q / (m + 1) for q in range(1, m + 1)]
        kint = mquantiles(np.log(y_uncensored), quant).astype('float32')
        kmin = math.log(y_uncensored.min())
        kmax = math.log(y_uncensored.max())

    else:
        print('custom knot placment')
    return kint, kmin, kmax


# custom negative likelihood loss
def nll(m,
        y_train=None,
        kint=None,
        kmin=None,
        kmax=None,
        loss='cumhaz',
        arandaordaz_c=1.,
        negative_dsdx_penalizer=1e4,
        scale_loss=1,
        ):
    """
    m number of interior knots,
    if kint kmin and kmax are given, y_train is not needed
    otherwise default quantile log scale knot placment is used according
    to uncensored failure times in y_train
    """
    arandaordaz_c = np.float32(arandaordaz_c) if not isinstance(arandaordaz_c,
np.float32) else arandaordaz_c
    arandaordaz_c = tf.convert_to_tensor(arandaordaz_c, dtype=tf.float32)
    negative_dsdx_penalizer = tf.convert_to_tensor(negative_dsdx_penalizer,
dtype=tf.float32)
    scale_loss = tf.convert_to_tensor(float(scale_loss), dtype=tf.float32)
    k2m = tf.expand_dims(kint, axis=0)
    lambda_j = (kmax - k2m) / (kmax - kmin)
    mlambda_j = tf.constant(1.0) - lambda_j

    def sxgamma(x, gamma):
        theta2m = tf.math.accumulate_n(
            [tf.math.pow(K.relu(tf.subtract(x, k2m)), 3),
             -tf.multiply(lambda_j, tf.math.pow(K.relu(tf.subtract(x, kmin)), 3)),
             -tf.multiply(mlambda_j, tf.math.pow(K.relu(tf.subtract(x, kmax)), 3)),
]
        )
        ret = tf.math.multiply(gamma[:, 0:1], x) + tf.reduce_sum(tf.multiply(
            gamma[:, 1:], theta2m), axis=1, keepdims=True, name='dot')
        return ret

    def dsdx(x, gamma):
        dsdx = gamma[:, 0:1] + (tf.constant(3.0) * tf.reduce_sum(
            tf.multiply(
                gamma[:, 1:],
                tf.math.accumulate_n([K.square(K.relu(tf.subtract(x, k2m))),
                                      - tf.multiply(lambda_j,
K.square(K.relu(tf.subtract(x, kmin)))),
                                      - tf.multiply(mlambda_j,
K.square(K.relu(tf.subtract(x, kmax))))],
```

```
                                        shape=[None, m]), ),
            axis=1, keepdims=True))
        #        log_dsdx = tf.where(dsdx < tf.constant(0.), - 50 * tf.abs(dsdx),
K.log(dsdx_clipped))

        return dsdx

    @tf.custom_gradient
    def p1csoftplusc(eta):
        cp1c = 1. + (1. / arandaordaz_c)

        def grad(upstream):
            return upstream * (arandaordaz_c + 1) * (1 / (arandaordaz_c +
tf.math.exp(-eta)))
            # return tf.cond(tf.equal(arandaordaz_c, 1.),
            #         lambda: upstream * 2 * (1 / (1 + tf.math.exp(-eta))),
            #         # since cp1c is always positive, no special treatment of zero
is necessary
            #         lambda: upstream * (arandaordaz_c + 1) * (1 / (arandaordaz_c
+ tf.math.exp(-eta))),
            #         )

        return tf.cond(tf.equal(arandaordaz_c, 1.),
                       lambda: 2 * tf.math.softplus(eta),
                       lambda: tf.math.xlog1py(cp1c, arandaordaz_c *
tf.math.exp(eta))), grad

    # @tf.custom_gradient
    def negative_dsdx_penalty(dsdx_tens):
        return tf.clip_by_value(
            (negative_dsdx_penalizer / scale_loss) * tf.math.sqrt(
                tf.nn.l2_loss(tf.nn.softplus(tf.negative(dsdx_tens)))),
            0, 1e20)

    def negative_dsdx_penalty_relu(dsdx_tens):
        return tf.clip_by_value(
            (negative_dsdx_penalizer / scale_loss) *
tf.math.sqrt(tf.nn.l2_loss(tf.nn.relu(tf.negative(dsdx_tens)))),
            0, 1e20)

    # @tf.function
    def loss_cumhaz(y_true, y_pred):
        """
            y_true , shaped  event, time
            y_pred , shaped (b + gamma1 + gammax ...)
            kint interior knots
        """
        beta = tf.expand_dims(y_pred[:, 0], 1)
        gamma = y_pred[:, 1:]
        #        beta = tf.expand_dims(beta, 1)

        t = y_true[:, 1]
        x = tf.expand_dims(K.log(t), 1)

        eta = beta + sxgamma(x, gamma)

        b_censored = tf.equal(y_true[:, 0], 0.)
        dsdx_uncens = dsdx(tf.boolean_mask(x, ~b_censored), tf.boolean_mask(gamma,
~b_censored))
        dsdx_clipped = tf.clip_by_value(dsdx_uncens, 1e-20, 1e30)  # clip to avoid
neg. cumhazards

        uncensored = tf.math.accumulate_n(
            [- tf.boolean_mask(x, ~b_censored),
             K.log(dsdx_clipped),
             tf.boolean_mask(eta, ~b_censored),
```

```
            -K.exp(tf.boolean_mask(eta, ~b_censored))]
        )
        # return uncensored

        censored = -K.exp(tf.boolean_mask(eta, b_censored))
        nonpen_loglik = (1 / scale_loss) * (tf.reduce_sum(uncensored) +
tf.reduce_sum(censored))
        # neg_ds_dx_penalty = negative_dsdx_penalty(dsdx_uncens)
        # loglik = nonpen_loglik - neg_ds_dx_penalty
        return -nonpen_loglik

    def loss_arandaordaz(y_true, y_pred):
        """
            y_true , shaped  event, time
            y_pred , shaped (b + gamma1 + gammax ...)
            kint interior knots
        """
        beta = tf.expand_dims(y_pred[:, 0], 1)
        gamma = y_pred[:, 1:]
        #        beta = tf.expand_dims(beta, 1)

        t = y_true[:, 1]
        x = tf.expand_dims(K.log(t), 1)

        eta = beta + sxgamma(x, gamma)

        b_censored = tf.equal(y_true[:, 0], 0.)
        #        return tf.boolean_mask(x,~b_censored)
        dsdx_uncens = dsdx(tf.boolean_mask(x, ~b_censored), tf.boolean_mask(gamma,
~b_censored))
        dsdx_clipped = tf.clip_by_value(dsdx_uncens, 1e-20, 1e30)  # clip to avoid
neg. cumhazards
        soft_eta = -p1csoftplusc(tf.boolean_mask(eta, ~b_censored))
        uncensored = tf.math.accumulate_n(
            [- tf.boolean_mask(x, ~b_censored),
             K.log(dsdx_clipped),
             tf.boolean_mask(eta, ~b_censored),
             soft_eta]
        )

        censored = -p1csoftplusc(tf.boolean_mask(eta, b_censored))
        nonpen_loglik = (1 / scale_loss) * (tf.reduce_sum(uncensored) +
tf.reduce_sum(censored))
        neg_ds_dx_penalty = negative_dsdx_penalty(dsdx_uncens)
        # penalty = tf.cond(tf.greater(neg_ds_dx_penalty, 0.), lambda:
neg_ds_dx_penalty, lambda: 0. )
        loglik = nonpen_loglik - neg_ds_dx_penalty
        return - loglik

    def dsdx_penality(y_true, y_pred):
        """
            y_true , shaped  event, time
            y_pred , shaped (b + gamma1 + gammax ...)
            kint interior knots
        """
        beta = tf.expand_dims(y_pred[:, 0], 1)
        gamma = y_pred[:, 1:]
        #        beta = tf.expand_dims(beta, 1)

        t = y_true[:, 1]
        x = tf.expand_dims(K.log(t), 1)

        b_censored = tf.equal(y_true[:, 0], 0.)
        dsdx_uncens = dsdx(tf.boolean_mask(x, ~b_censored), tf.boolean_mask(gamma,
~b_censored))
```

```
        return negative_dsdx_penalty_relu(dsdx_uncens)

    if loss == 'cumhaz':
        return loss_cumhaz
    if loss == 'aranda_ordaz':
        if tf.equal(arandaordaz_c, 0.).numpy():
            return loss_cumhaz
        else:
            return loss_arandaordaz

    if loss == 'dsdx_penalty':
        return dsdx_penality


class DeepBnReluDrop(Layer):
    def __init__(self,
                 hidden_layers_sizes,
                 bias_initializer=he_normal(),
                 kernel_initializer=he_normal(),
                 kernel_regularizer=None,
                 dropout_rate=None,
                 **kwargs,
                 ):
        self.hidden_layers_sizes = hidden_layers_sizes
        self.kernel_regularizer = kernel_regularizer
        self.kernel_initializer = kernel_initializer
        self.bias_initializer = bias_initializer
        self.dropout_rate = dropout_rate
        super(DeepBnReluDrop, self).__init__(**kwargs)

        # Layers

    def call(self, inputs, **kwargs):
        x = inputs
        x = self.dense(inputs)
        if self.dropout_rate is not None:
            x = self.dropout(x)
        x = self.relu(x)
        x = self.bn(x)
        return x

    def build(self, input_shape):
        self.dense = Dense(self.hidden_layers_sizes,
                           kernel_initializer=self.kernel_initializer,
                           bias_initializer=self.bias_initializer,
                           kernel_regularizer=self.kernel_regularizer,
                           name='Dense')
        self.bn = BatchNormalization()
        self.relu = Activation('relu')
        if self.dropout_rate is not None:
            self.dropout = Dropout(rate=self.dropout_rate)

        super(DeepBnReluDrop, self).build(input_shape)


class ResBlock(Layer):
    def __init__(self,
                 hidden_layers_sizes,
                 bias_initializer=he_normal(),
                 kernel_initializer=he_normal(),
                 kernel_regularizer=None,
                 dropout_rate=None,
                 **kwargs,
                 ):
        self.version = 'v0'
        self.hidden_layers_sizes = hidden_layers_sizes
```

```
        self.kernel_regularizer = kernel_regularizer
        self.kernel_initializer = kernel_initializer
        self.bias_initializer = bias_initializer
        self.dropout_rate = dropout_rate
        super(ResBlock, self).__init__(**kwargs)

        # Layers

    def call(self, inputs, **kwargs):

        x = inputs
        if self.version == 'v0':
            # # pre-activation
            # bn1 = self.bn(x)
            # relu = self.relu(bn1)
            dense1 = self.dense(x)
            relu2 = self.relu(dense1)
            bn2 = self.bn(relu2)
        else:
            raise NotImplementedError('%s is currently not supported' %
self.version)
        # x = self.dense(x)
        # if self.dropout_rate is not None:
        #     x = self.dropout(x)
        # skip connections
        skip = Add()([bn2, inputs])

        return skip

    def build(self, input_shape):
        self.dense = Dense(self.hidden_layers_sizes,
                           kernel_initializer=self.kernel_initializer,
                           bias_initializer=self.bias_initializer,
                           kernel_regularizer=self.kernel_regularizer,
                           name='Dense')
        self.bn = BatchNormalization()
        self.relu = Activation('relu')
        if self.dropout_rate is not None:
            self.dropout = Dropout(rate=self.dropout_rate)

        super(ResBlock, self).build(input_shape)


class nnet_rpsplinesEstimator(BaseEstimator):
    class RPSplineModel(Model):
        def __init__(self,
                     inputshape=None,
                     anc_inputshape=None,
                     Estimator=None,
                     **kwargs):
            super(nnet_rpsplinesEstimator.RPSplineModel, self).__init__(**kwargs)
            self.est = Estimator
            self.inputshape = inputshape
            self.anc_inputshape = anc_inputshape

            # Layers:
            self.HiddenLayer = ResBlock(self.est.hidden_layers_sizes,
                                        kernel_initializer=he_normal(),
                                        bias_initializer=he_normal(),
                                        dropout_rate=self.est.dropout_rate,

kernel_regularizer=self.est.kernel_regularizer,
                                        name='DeepBeta')
            self.InputLayer = DeepBnReluDrop(self.est.hidden_layers_sizes,
                                             kernel_initializer=he_normal(),
                                             bias_initializer=he_normal(),
```

```
                                            dropout_rate=self.est.dropout_rate,

kernel_regularizer=self.est.kernel_regularizer,
                                            name='InputDense')
            self.AncHiddenLayers = ResBlock(
                self.est.anc_hidden_layers_sizes,
                dropout_rate=self.est.dropout_rate,
                bias_initializer=he_normal(),
                kernel_regularizer=self.est.kernel_regularizer,
                name='DeepGamma')
            self.Betax = Dense(1,
                               kernel_initializer=he_normal(),
                               bias_initializer=he_normal(),
                               use_bias=True,
                               kernel_regularizer=self.est.kernel_regularizer,
                               name='Betax',
                               trainable=True)
            # self.Beta0 = BiasLayer(units=1, name='Beta0')
            self.GammaLayer = FinalLayerGamma(
                self.est.int_knots_number,
                self.est.flexgamma,
                kernel_regularizer=self.est.kernel_regularizer,
                name='Gamma')

            self.OutputLayer = Concatenate(name='Output')

        #        def build(self, input_shape):
        #            self.inputshape = input_shape
        #
super(nnet_rpsplinesEstimator.RPSplineModel,self).build(input_shape)

        def call(self, inputs, **kwargs):
            inp = inputs
            if isinstance(inputs, list):
                inp, anc_inp = inputs
            else:
                if self.est.anc_input:
                    raise AssertionError('If anc_input version is run, anc input
needs to be given along inp.')

            current_layer = inp

            deep_layers = []
            # input layer
            if self.est.hiddenlayers > 0:
                deep_layers.append(self.InputLayer(current_layer))

            # hidden layers
            for j in range(1, self.est.hiddenlayers):
                deep_layers.append(self.HiddenLayer(deep_layers[-1]))

            # Final Beta layer
            betax_layer = self.Betax(deep_layers[-1])
            final_beta_layer = betax_layer  # Add()([betax_layer,
self.Beta0(current_layer)])
            if self.est.flexgamma:
                # model shape of spline explicitely
                gamma_layer = self.GammaLayer(deep_layers[-1])
            else:
                gamma_layer = self.GammaLayer(current_layer)

            concat = self.OutputLayer([final_beta_layer, gamma_layer])
            return concat

    def __init__(self,
                 conf,
```

```
                 loss='aranda_ordaz',
                 arandaordaz_c=0,
                 l1_ratio=1,
                 penalizer=0,
                 learning_rate=1e-3,
                 batch_size=1024,
                 dropout=False,
                 epochs=10000,
                 earlystopping=True,
                 train_ratio=0.8,
                 int_knots_number=1,
                 flexgamma=False,
                 hidden_layers_sizes=10,
                 hiddenlayers=1,
                 anc_hidden_layers_sizes=2,
                 anc_hiddenlayers=1,
                 scale=12,
                 random_state=None,
                 neg_dsdx_penalizer=1e4,
                 verbose=0,
                 debug=True,
                 ):
        # config general
        self.verbose = verbose
        self.debug = debug
        self.event_times_ = conf.eval_times
        self.conf = conf

        # training
        self.learning_rate = learning_rate
        self.batchsize = int(batch_size)
        self.epochs = epochs

        # early stopping
        self.earlystopping = earlystopping
        self.train_ratio = train_ratio  # ratio of used train/(train + val)

        # model specific
        self.loss = loss

        # model specific
        self.loss = loss
        self.arandaordaz_c = np.float32(arandaordaz_c) if not
isinstance(arandaordaz_c, np.float32) else arandaordaz_c
        # 0 = hazard scale, 1 = logit scale
        self.scale = scale  # scale factor for time
        self.int_knots_number = int_knots_number  # internal knots -> k + 2 total

        self.hidden_layers_sizes = hidden_layers_sizes  # size of hidden layer
        self.hiddenlayers = hiddenlayers  # depth of hidden layer

        self.flexgamma = flexgamma  # model gamma params also?
        self.anc_input = False
        self.anc_hidden_layers_sizes = anc_hidden_layers_sizes
        self.anc_hiddenlayers = anc_hiddenlayers
        self.univariate_prefitting = False

        # penalization
        self.penalizer = penalizer
        self.l1_ratio = l1_ratio
        if dropout:
            self.kernel_regularizer = None
            self.dropout_rate = penalizer
        else:
            self.dropout_rate = None
            self.l1_ratio = l1_ratio
```

```
            self.kernel_regularizer = l1_l2(l1=self.penalizer * self.l1_ratio,
l2=self.penalizer * (1 - self.l1_ratio))

        self.neg_dsdx_penalizer = neg_dsdx_penalizer

        # init model, train set,
        self.model = None
        self.y_train = None
        self.y_train_uncensored = None

    def __enter__(self, ):
        return self

    def __exit__(self, *err):
        #        self.sess.close()
        return False

    def GetRPSplineModel(self, X_train=None, y_train=None, Ancillary_X=None,
                         training_X=None,
                         ):
        """
        y_train needed for knot placement
        y_train should be numpy array shaped (n_samples, 2)
        with first field binary event indicator, second field time of
censoring/event
        float type needed
        """

        inp = Input((X_train.shape[-1],), name='Input')

        if self.anc_input:
            anc_inp = Input((Ancillary_X.shape[-1],), name='AncillaryInput')
            inp_list = [inp, anc_inp]
        else:
            inp_list = inp
        out = self.RPSplineModel(Estimator=self, )(inp_list)
        model = Model(inputs=inp_list, outputs=out)

        # get knot placement
        kint, kmin, kmax = get_knots(self.int_knots_number, y_train, )
        self.kint, self.kmin, self.kmax = kint, kmin, kmax

        # get negative likelihood loss
        nll_loss = nll(self.int_knots_number, y_train, kint, kmin, kmax,
                       loss=self.loss, arandaordaz_c=self.arandaordaz_c,
                       negative_dsdx_penalizer=self.neg_dsdx_penalizer,
                       scale_loss=1.)

        # get additional metric to monitor monotonicity penalty
        penalty_metric = nll(self.int_knots_number, y_train, kint, kmin, kmax,
                             loss='dsdx_penalty',
negative_dsdx_penalizer=self.neg_dsdx_penalizer * 100,
                             scale_loss=1.)
        # compile model and init splines to be monotonically
        # increasing high lr is wanted for a quick overshoot across zero
        model.compile(loss=penalty_metric,
                      metrics=[penalty_metric],
                      optimizer=Adam(lr=0.3))
        #        model.build((None, X_train.shape[-1], ))
        # callbacks = [EarlyStopping(monitor='loss', patience=5),
        #              ]

        # init spline to be monotonically increasing given the data
        if not np.array(penalty_metric(tf.convert_to_tensor(y_train),
model(training_X)) == 0.0).item():
            lowest_loss = np.inf
```

```
            thresh = 1e-6
            patience = 1
            patience_step = 0
            for step in range(self.epochs):
                with tf.GradientTape() as tape:
                    y_pred = model(training_X)
                    loss = model.compiled_loss(tf.convert_to_tensor(y_train),
y_pred)
                if self.verbose > 0:
                    print('Epoch:', step, 'loss:', loss)

                # stop when monotonical spline is there
                if loss < thresh:
                    break
                # # early stopping
                # if (lowest_loss - loss) < thresh:
                #     patience_step += 1
                #     if patience_step >= patience:
                #         break
                # else:
                #     lowest_loss = loss
                #     patience_step = 0
                gradients = tape.gradient(loss, model.trainable_variables)
                model.optimizer.apply_gradients(zip(gradients,
model.trainable_variables))

        # get layers for later processing
        b_spline_model = [str(var.name).find("spline_model") >= 0 for var in
model.layers]
        spline_model = list(compress(model.layers, b_spline_model))[0]

        #        search beta_layers
        b_beta = [str(var.name).find("Betax") >= 0 for var in spline_model.layers]
        betax = list(compress(spline_model.layers, b_beta))[0]
        # b_beta0 = [str(var.name).find("Beta0") >= 0 for var in
spline_model.layers]
        # beta0 = list(compress(spline_model.layers, b_beta0))[0]
        b_deepbeta = [str(var.name).find("DeepBeta") >= 0 for var in
spline_model.layers]
        deepbeta = list(compress(spline_model.layers, b_deepbeta))
        #        search gamma layers
        b_gamma = [str(var.name) == "Gamma" for var in spline_model.layers]
        gamma = list(compress(spline_model.layers, b_gamma))[0]
        b_deepgamma = [str(var.name).find("DeepGamma") >= 0 for var in
spline_model.layers]
        deepgamma = list(compress(spline_model.layers, b_deepgamma))
        layers_dict = {'betax': betax,
                       # 'beta0': beta0,
                       'deepbeta': deepbeta,
                       'deepgamma': deepgamma,
                       'gamma': gamma}

        model.compile(loss=nll_loss,
                      metrics=[penalty_metric],
                      optimizer=Adam(lr=self.learning_rate))
        if self.verbose > 0:
            model.summary()
        loss_metric = dict(nll_loss=nll_loss, penalty_metric=penalty_metric)
        return model, layers_dict, loss_metric

    @tf.autograph.experimental.do_not_convert
    def fit(self, X, y, anc_X=None):
        # prepare y data
        event = y['dead'].copy().astype(np.float32)
        time = y['time'].copy().astype(np.float32) * self.scale
        y_train = np.stack([event, time], axis=-1)
```

```
        # ancillary df
        if anc_X is None:
            anc_X = X

        # init model

        # if earlystopping == early stopping with val_loss, else 'loss'
        if self.earlystopping:
            self.epochs = 10000
            X_train, X_val, y_train, y_val, anc_X_train, anc_X_val =
train_test_split(
                X, y_train, anc_X,
                shuffle=True, train_size=self.train_ratio, )
            callbacks = [EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True),
                         # LossHistory(logpath=self.conf.job_dir + "logs/"),
                         ]
            if self.anc_input:
                training_X = {'Input': X_train, 'AncillaryInput': anc_X_train}
                val_X = {'Input': X_val, 'AncillaryInput': anc_X_val}

            else:
                training_X = {'Input': X_train}
                val_X = {'Input': X_val}
            self.model, self.layers_dict, self.loss_metric =
self.GetRPSplineModel(X_train,

y_train,

anc_X_train,

training_X)

            if self.univariate_prefitting:
                # init spline by univariate regression before real training

                if self.verbose > 0:
                    print("Prefitting")
                    print("weights:", len(self.layers_dict['gamma'].weights))
                    print("trainable_weights:",
len(self.layers_dict['gamma'].trainable_weights))
                    print("non_trainable_weights:",
len(self.layers_dict['gamma'].non_trainable_weights))
                for layer in chain([self.layers_dict['betax']],
self.layers_dict['deepbeta'],
                                   self.layers_dict['deepgamma']):
                    layer.trainable = False
                self.model.compile(loss=self.loss_metric['nll_loss'],
                                   metrics=[self.loss_metric['penalty_metric']],
                                   optimizer=Adam(lr=self.learning_rate),
                                   )
                self.model.fit(training_X,
                               y_train,
                               callbacks=callbacks,
                               epochs=self.epochs,
                               verbose=self.verbose,
                               validation_data=(val_X, y_val),
                               )
                for layer in chain([self.layers_dict['betax']],
self.layers_dict['deepbeta'],
                                   self.layers_dict['deepgamma']):
                    layer.trainable = True
                self.layers_dict['gamma'].trainable = True
            else:
                self.layers_dict['gamma'].trainable = True
```

```
            self.debug = False
            if self.debug:
                # custom fit loop
                model = self.model
                y_train_tf = tf.convert_to_tensor(y_train)
                old_loss = np.inf
                step = 0

                for step in range(self.epochs):
                    with tf.GradientTape() as tape:
                        # tape.watch(model.trainable_variables[-1])
                        y_pred = model(training_X)
                        loss = model.compiled_loss(y_train_tf, y_pred)
                    print('Epoch:', step, 'loss:', loss)

                    # early stopping
                    # if old_loss - loss < 0.001:
                    #     break

                    old_loss = loss
                    gradients = tape.gradient(loss, model.trainable_variables)
                    model.optimizer.apply_gradients(zip(gradients,
model.trainable_variables))

            self.model.fit(training_X,
                           tf.convert_to_tensor(y_train),
                           callbacks=callbacks,
                           epochs=self.epochs,
                           validation_data=(val_X, tf.convert_to_tensor(y_val)),
                           verbose=self.verbose,
                           batch_size=int(self.batchsize)
                           )

        else:
            if self.anc_input:
                training_X = {'Input': X, 'AncillaryInput': anc_X}
            else:
                training_X = {'Input': X}
            callbacks = [EarlyStopping(monitor='loss', patience=10,
restore_best_weights=True),
                          # LossHistory(logpath=self.conf.job_dir + "logs/"),
                          ]
            self.model, self.layers_dict, self.loss_metric =
self.GetRPSplineModel(X,

y_train,

anc_X,

training_X)

            self.model.fit(training_X,
                           y_train,
                           callbacks=callbacks,
                           epochs=self.epochs,
                           verbose=self.verbose)
        #        print(self.model.summary())
        self.y_train = y_train
        self.y_train_uncensored = y_train[y_train[:, 0] == 1.]
        self.event_times_ = np.sort(y_train[:, 1][y_train[:, 0] == 1.])

    def predict(self, X):
        """Predict risk score.
```

```
        Parameters
        ----------
        X : array-like, shape = (n_samples, n_features)
            Data matrix.

        Returns
        -------
        risk_scores : ndarray, shape = (n_samples,)
            Predicted risk scores.
        """
        estimate = self.predict_survival_function(X)
        #         estimate_t=estimate['y'][estimate['x']<=self.conf.brier_t]
        ind = np.argmin(estimate['x'][estimate['x'] >= self.conf.brier_t], axis=0)
        return -estimate['y'][:, ind]

    def _predict(self, X):
        try:
            beta_X, anc_X = X
            return self.model.predict([beta_X, anc_X], batch_size=self.batchsize,
verbose=0)
        except ValueError as e:
            # print(e)
            return self.model.predict(X, batch_size=self.batchsize, verbose=0)
        except Exception as er:
            print(er)
            raise er

    @tf.function
    def _eta(self, y_pred):
        kint, kmin, kmax = self.kint, self.kmin, self.kmax

        k2m = tf.expand_dims(kint, axis=0)
        lambda_j = (kmax - k2m) / (kmax - kmin)
        mlambda_j = tf.constant(1.0) - lambda_j

        def sxgamma(x, gamma):
            theta2m = tf.math.accumulate_n(
                [tf.math.pow(K.relu(tf.subtract(x, k2m)), 3),
                 -tf.multiply(lambda_j, tf.math.pow(K.relu(tf.subtract(x, kmin)),
3)),
                 -tf.multiply(mlambda_j, tf.math.pow(K.relu(tf.subtract(x, kmax)),
3)), ]
            )
            ret = tf.math.multiply(gamma[:, 0:1], tf.transpose(x)) + tf.matmul(
                gamma[:, 1:], tf.transpose(theta2m), name='dot')
            return ret

        beta = tf.expand_dims(y_pred[:, 0], 1)
        gamma = y_pred[:, 1:]

        t = tf.convert_to_tensor(self.event_times_, dtype=tf.float32)
        #         self.event_times_ = t
        x = tf.expand_dims(K.log(t), 1)
        eta = beta + sxgamma(x, gamma)
        return eta

    # @tf.function
    def _aranda_ordaz_link(self, X):
        eta = self._eta(self._predict(X))
        c = self.arandaordaz_c

        def inv_logit_link():
            return tf.math.sigmoid(-eta)

        def inv_log_log_link():
            return tf.math.exp(-tf.math.exp(eta))
```

```
    def inv_aranda_ordaz_link():
        return tf.math.pow(1 + (c * tf.math.exp(eta)), -1 / c)

    return tf.cond(tf.constant(c == 1, dtype=tf.bool),
                   inv_logit_link,  # logit link
                   lambda: tf.cond(tf.constant(c == 0, dtype=tf.bool),
                                   inv_log_log_link,  # log log
                                   inv_aranda_ordaz_link,  # aranda-ordaz
                                   ))

def _predict_survival_function_y(self, X):

    return self._aranda_ordaz_link(X)

#        cumhaz = self._cumhaz(self._predict(X))# tf 2.
#        return np.exp(-np.array(cumhaz))

def predict_survival_function(self, X):
    """Predict survival function.
    predicts survival function at breaks timepoints.

    Parameters
    ----------
    X : array-like, shape = (n_samples, n_features)
        Data matrix.

    Returns
    -------
    survival : ndarray, shape = (n_samples, n_event_times)
        Predicted survival functions.
    """
    try:
        beta_X, anc_X = X
        survival_fcn = np.zeros((beta_X.shape[0],
                                 self.event_times_.shape[0]),
                                dtype=[('x', 'float64'), ('y', 'float64')])
        shape = X.shape[0]
    except ValueError as e:
        repr(e)

        survival_fcn = np.zeros((X.shape[0],
                                 self.event_times_.shape[0]),
                                dtype=[('x', 'float64'), ('y', 'float64')])
        shape = X.shape[0]
        if self.anc_input:
            X = (X, X)

    survival_fcn['x'] = np.tile(self.event_times_, (shape, 1)) / self.scale
    survival_fcn['y'] = self._predict_survival_function_y(X)
    return survival_fcn.copy()
```

8.3 Source code of the Tensorflow 2 Implementation of 'nnet-survival'

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr  8 16:05:23 2020

@author: Fabian

Based on
"""

from sklearn.base import BaseEstimator
from sklearn.model_selection import train_test_split
from sksurv.util import check_y_survival
import numpy as np

# Model

import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.layers import Layer
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout, Activation,
BatchNormalization
from tensorflow.keras.regularizers import l1_l2
# training
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, Callback,
LearningRateScheduler

from tensorflow.keras.optimizers import Adam


# Utilities from M Gensheimer's nnet-survival


def surv_likelihood(n_intervals):
    """Create custom Keras loss function for neural network survival model.
  Arguments
      n_intervals: the number of survival time intervals
  Returns
      Custom loss function that can be used with Keras
  """

    def loss(y_true, y_pred):
        """
    Required to have only 2 arguments by Keras.
    Arguments
        y_true: Tensor.
          First half of the values is 1 if individual survived that interval, 0 if
not.
          Second half of the values is for individuals who failed, and is 1 for
time interval during which failure
          occured, 0 for other intervals.
          See make_surv_array function.
        y_pred: Tensor, predicted survival probability (1-hazard probability) for
each time interval.
    Returns
        Vector of losses for this minibatch.
    """
        cens_uncens = 1. + y_true[:, 0:n_intervals] * (y_pred - 1.)  # component
for all individuals
        uncens = 1. - y_true[:, n_intervals:2 * n_intervals] * y_pred  # component
for only uncensored individuals
        return K.sum(-K.log(K.clip(K.concatenate((cens_uncens, uncens)),
K.epsilon(), None)),
```

```
                    axis=-1)  # return -log likelihood

    return loss


def surv_likelihood_rnn(n_intervals):
    """
    Create custom Keras loss function for neural network survival model. Used for
recurrent neural networks with
    time-distributed output.
    This function is very similar to surv_likelihood but deals with the extra
dimension of y_true and y_pred that
    exists because of the time-distributed output.
  """

    def loss(y_true, y_pred):
        cens_uncens = 1. + y_true[0, :, 0:n_intervals] * (y_pred - 1.)  # component
for all patients
        uncens = 1. - y_true[0, :, n_intervals:2 * n_intervals] * y_pred  #
component for only uncensored patients
        return K.sum(-K.log(K.clip(K.concatenate((cens_uncens, uncens)),
K.epsilon(), None)),
                    axis=-1)  # return -log likelihood

    return loss


def make_surv_array(t, f, breaks):
    """Transforms censored survival data into vector format that can be used in
Keras.
    Arguments
        t: Array of failure/censoring times.
        f: Censoring indicator. 1 if failed, 0 if censored.
        breaks: Locations of breaks between time intervals for discrete-time
survival model (always includes 0)
    Returns
        Two-dimensional array of survival data, dimensions are number of
individuals X number of time intervals*2
  """
    n_samples = t.shape[0]
    n_intervals = len(breaks) - 1
    timegap = breaks[1:] - breaks[:-1]
    breaks_midpoint = breaks[:-1] + 0.5 * timegap
    y_train = np.zeros((n_samples, n_intervals * 2))
    for i in range(n_samples):
        if f[i]:  # if failed (not censored)
            y_train[i, 0:n_intervals] = 1.0 * (t[i] >= breaks[1:])
            # give credit for surviving each time interval where failure time >=
upper limit
            if t[i] < breaks[-1]:
                # if failure time is greater than end of last time interval, no
time interval will have failure marked
                y_train[i, n_intervals + np.where(t[i] < breaks[1:])[0][
                    0]] = 1  # mark failure at first bin where survival time <
upper break-point
        else:  # if censored
            y_train[i, 0:n_intervals] = 1.0 * (t[i] >= breaks_midpoint)
            # if censored and lived more than half-way through interval, give
credit for surviving the interval.
    return y_train


def nnet_pred_surv(y_pred, breaks, fu_time):
    # Predicted survival probability from Nnet-survival model
    # Inputs are Numpy arrays.
    # y_pred: Rectangular array, each individual's conditional probability of
```

```
surviving each time interval
    # breaks: Break-points for time intervals used for Nnet-survival model,
starting with 0
    # fu_time: Follow-up time point at which predictions are needed
    #
    # Returns: predicted survival probability for each individual at specified
follow-up time
    y_pred = np.cumprod(y_pred, axis=1)
    pred_surv = []
    for i in range(y_pred.shape[0]):
        pred_surv.append(np.interp(fu_time, breaks[1:], y_pred[i, :]))
    return np.array(pred_surv)


class PropHazards(Layer):

    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(PropHazards, self).__init__(**kwargs)

    def build(self, input_shape):
        # Create a trainable weight variable for this layer.
        self.kernel = self.add_weight(name='kernel',
                                      shape=(1, self.output_dim),
                                      # initializer='uniform',
                                      initializer='zeros',
                                      trainable=True)
        super(PropHazards, self).build(input_shape)  # Be sure to call this
somewhere!

    def call(self, x):
        # The conditional probability of surviving each time interval (given that
has survived to beginning of interval)
        # is affected by the input data according to eq. 18.13 in Harrell F.,
        # Regression Modeling Strategies 2nd ed. (available free online)
        return K.pow(K.sigmoid(self.kernel), K.exp(x))

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)


class SurvivalFcn:

    def __init__(self,
                 x,
                 y,
                 ):
        self.x = x
        self.y = y


class NnetSurvivalEstimator(BaseEstimator):
    """
    scikit learn style wrapper for the Gensheimer's nnet-survival Neural network
for survival data
    """
    def __init__(self,
                 conf,
                 l1_ratio=1,
                 penalizer=0,
                 learning_rate=1e-3,
                 dropout=False,
                 prophazard=False,
                 epochs=10000,
                 earlystopping=True,
                 train_ratio=0.8,
```

```
                 hidden_layers_sizes=10,
                 hiddenlayers=1,
                 batch_size=256,
                 breaks=None,
                 random_state=None,
                 ):
        # config general
        if breaks is None:
            if hasattr(conf, 'breaks'):
                self.event_times_ = conf.breaks[1:]
            else:
                self.default_breaks_per_event_ = 10
                self.event_times_ = None
        else:
            self.event_times_ = breaks
        self.conf = conf
        self.learning_rate = learning_rate
        self.batchsize = batch_size
        self.epochs = epochs
        # early stopping
        self.earlystopping = True
        self.train_ratio = train_ratio

        # model specific
        self.hidden_layers_sizes = hidden_layers_sizes
        self.prophazard = prophazard
        self.hiddenlayers = hiddenlayers
        self.penalizer = penalizer
        self.l1_ratio = l1_ratio
        if dropout:
            self.kernel_regularizer = None
            self.dropout_rate = penalizer
        else:
            self.dropout_rate = None
            self.l1_ratio = l1_ratio
            self.kernel_regularizer = l1_l2(l1=self.penalizer * self.l1_ratio,
l2=self.penalizer * (1 - self.l1_ratio))
        self.model = None  # Keras Model

    def GetDenseModel(self, ):

        n_intervals = len(self.event_times_) - 1
        model = Sequential()
        # hidden layers:
        for j in range(self.hiddenlayers):
            model.add(Dense(self.hidden_layers_sizes,
                            bias_initializer='zeros',
                            kernel_regularizer=self.kernel_regularizer,
                            )
                      )
            model.add(BatchNormalization())
            model.add(Activation('relu'))
            # dropout layer
            if self.dropout_rate is not None:
                model.add(Dropout(rate=self.dropout_rate))
        # discrete time interval odds and probability:
        if self.prophazard:
            model.add(Dense(1, use_bias=0, kernel_initializer='zeros'))
            model.add(PropHazards(n_intervals))
        # flexible non-propotional hazard model.
        else:
            model.add(Dense(n_intervals))
            model.add(Activation('sigmoid'))
        model.compile(loss=surv_likelihood(n_intervals),
                      optimizer=Adam(lr=self.learning_rate))
```

```
        return model

    def fit(self, X, y, ):

        # prepare y data
        event, time = check_y_survival(y)

        # get event_times_ from train dataset
        if self.event_times_ is None:
            no_breaks = int(round(np.count_nonzero(event) /
self.default_breaks_per_event_))
            self.event_times_ = np.linspace(time[event].min(), time[event].max(),
no_breaks)

        y_train = make_surv_array(time, event, self.event_times_)

        # get Keras model
        self.model = self.GetDenseModel()

        # prepare callbacks
        if self.earlystopping:
            self.epochs = 10000
            X_train, X_val, y_train, y_val = train_test_split(X, y_train,
shuffle=True, train_size=self.train_ratio, )
            callbacks = [EarlyStopping(monitor='val_loss', patience=10),
                         ]
            self.model.fit(X_train, y_train, callbacks=callbacks,
epochs=self.epochs, validation_data=(X_val, y_val),
                           verbose=0)

        else:
            callbacks = [  # EarlyStopping(monitor='val_loss', patience=10),
            ]
            self.model.fit(X, y_train, callbacks=callbacks, epochs=self.epochs,
verbose=0)

    def predict(self, X, t=None):
        """
        Predict Survival Probability at time t

        Parameters
        ----------
        X : array-like, shape = (n_samples, n_features)
            Data matrix.

        t: time to predict the survival probability at

        Returns
        -------
        risk_scores : ndarray, shape = (n_samples,)
            Predicted risk scores.
        """
        estimate = self.predict_survival_function(X)
        if t is None:
            raise ValueError('You need to specify a time, at which to predict the
survival probability!')
        ind = np.argmin(estimate['x'][estimate['x'] >= t], axis=0)
        return -estimate['y'][:, ind]

    def _predict(self, X):
        return self.model.predict(X, batch_size=self.batchsize, verbose=0)

    def _predict_survival_function_y(self, X):
        return np.cumprod(self._predict(X), axis=1)

    def predict_survival_function(self, X):
```

```
"""Predict survival function.
predicts survival function at breaks timepoints.

Parameters
----------
X : array-like, shape = (n_samples, n_features)
    Data matrix.

Returns
-------
survival : ndarray, shape = (n_samples, n_event_times)
    Predicted survival functions.
"""
survival_fcn = np.zeros(
    (X.shape[0], self.event_times_.shape[0] - 1),
    dtype=[('x', 'float64'), ('y', 'float64')])
survival_fcn['x'] = np.tile(self.event_times_[:-1], (X.shape[0], 1))
survival_fcn['y'] = self._predict_survival_function_y(X)
return survival_fcn.copy()
```

8.4 Source code of lowess-smoothed calibration plot for survival data

```
import matplotlib.pyplot as plt

# sklearn imports
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
from sklearn.calibration import calibration_curve as sklearncb
from sklearn.utils import check_array, check_consistent_length

# scikit survival imports
from sksurv.nonparametric import CensoringDistributionEstimator,
SurvivalFunctionEstimator
from sksurv.util import check_y_survival

# scikit misc
# from skmisc.loess import loess

# First-party
from .metrics import _interpolate_survfunc

# Third-party
from moepy import lowess
from joblib import Parallel, delayed
from functools import partial
import numpy
import pandas as pd


def _calib_plot(fig, ax, fu_time, n_bins, pred_surv, time, dead,
                color, label, scatter=True,
                error_bars=0, alpha=1., markersize=1., markertype='o'):
    """
    Kaplan Meier Estimates for n_bins of predicted survival pred_surv at fu_time
plotted against mean pred_surv
    typically deciles are used, despite arbitrary choice.
    Deprecated --> better use loess based/nearest neighbor estimated plot
    """
    # TODO: exchange lifelines KaplanMeierFitter with
sksurv.nonparametric.SurvivalFunctionEstimator. Confidence interval estimation
needed in advance.
    from lifelines import KaplanMeierFitter
    import matplotlib.pyplot as plt
    #    cuts = numpy.concatenate((numpy.array([-1e6]),numpy.percentile(pred_surv,
numpy.arange(100/n_bins,100,100/n_bins)),numpy.array([1e6])))
    #    bin = pd.cut(pred_surv,cuts,labels=False)
    bins = pd.qcut(pred_surv, q=n_bins, retbins=True, duplicates='drop')[0]._codes
    kmf = KaplanMeierFitter()
    est = []
    ci_upper = []
    ci_lower = []
    mean_pred_surv = []
    for which_bin in range(max(bins) + 1):
        kmf.fit(time[bins == which_bin], event_observed=dead[bins == which_bin])
        est.append(numpy.interp(fu_time, kmf.survival_function_.index.values,
kmf.survival_function_.KM_estimate))
        ci_upper.append(numpy.interp(fu_time, kmf.survival_function_.index.values,
                                     kmf.confidence_interval_.loc[:,
'KM_estimate_upper_0.95']))
        ci_lower.append(numpy.interp(fu_time, kmf.survival_function_.index.values,
                                     kmf.confidence_interval_.loc[:,
'KM_estimate_lower_0.95']))
        mean_pred_surv.append(numpy.mean(pred_surv[bins == which_bin]))
    est = numpy.array(est)
    ci_upper = numpy.array(ci_upper)
    ci_lower = numpy.array(ci_lower)
```

```
    if error_bars:
        ax.errorbar(mean_pred_surv, est,
yerr=numpy.transpose(numpy.column_stack((est - ci_lower, ci_upper - est))),
                    fmt='o',
                    c=color, label=label)
    else:
        ax.plot(mean_pred_surv, est, markertype, c=color, label=label, alpha=alpha,
markersize=markersize)

    if scatter:
        actual = numpy.empty(pred_surv.shape)
        for bi in bins:
            actual[bins == bi] = est[bi]
        ax.plot(pred_surv, actual, 'o', c=color, label=label, alpha=alpha,
markersize=.5 * markersize)
    return fig, ax, (numpy.array(mean_pred_surv), est, ci_upper, ci_lower)


def _calib_plot_loess(fig, ax, fu_time,
                      pred_surv, time, dead,
                      color, label,
                      ci=False, ci_alpha=.05, trunc=(1, 99),
                      alpha=1., markersize=1., linestyle='solid',
                      pseudovals=True,  # wether to use Jackknife pseudovals or
not.
                      loess=True,
                      scatter=True,
                      ):
    """
    plot calibration curve based on pseudovalues: est_i(t) = n * (prodlim(t)) - (n-
1) * prodlim_i(t)
    see r package prodlim https://rdrr.io/cran/prodlim/man/jackknife.html
    see also    Gerds et al.,Calibration plots for risk prediction models in the
presence of competing risks., https://doi.org/10.1002/sim.6152
                Graw et al., https://doi.org/10.1007/s10985-008-9107-z

    We combined the pseudovalues approach with skmisc.loess.loess smoothing
(https://dx.doi.org/10.1002%2Fsim.5941)
    span = 0.75 (default) works good according to
https://dx.doi.org/10.1002%2Fsim.5941

    Parameters:
    -----------
    * fig : plt.Figure
        matplotlib's figure object, where the plot should be plotted at

    * ax : plt.Axes
        matplotlib's axes object, where the plot should be plotted at


    * fu_time : int / float
        time for which cumulative calibration should be calculated

    * pred_surv : array, shape = (n_samples,)
        array with predicted survival probability at time t

    * time: array, shape = (n_samples,)
        array with time of event or censoring time

    * dead: array, shape = (n_samples,)
        event indicator array, True = event, False = no event

    * color: str
        color for the plot

    * label: str
```

```
        label of the plot

    * ci: bool
        controls, whether confidence intervals should be calculated

    * ci_alpha: float
        confidence niveau, default 0.05

    * trunc: tuple, type int / float
        truncate pred_surv at `trunc[0/1]`'th percentiles.

    * alpha: float
        alpha transluicency for plot

    * markersize: float
        markersize for pyplot

    * markertype: str
        type of marker used by matplotlib.pyplot, default '-' = line

    * loess: bool
        use loess smoother, if false (not recommended, as not tested) use k nearest
neighbors.

    * scatter: bool
        plot (pseudo)values as scatter plot as well (for debugging)

    Returns:
    --------

    * s_pred_surv: array, shape = (n_samples*)
        random subsample (only if n_samples >30000) of predicted survival
probability array at fu_time
    * actual: array, shape = (n_samples*)
        loess smoothed actual survival probability for each reduced sample
    * ci_lower:array, array, shape = (n_samples*)
        lower confidence limit if ci= True, otherwise ci_lower equals actual
    * ci_upper:array, array, shape = (n_samples*)
        upper confidence limit if ci= True, otherwise ci_lower equals actual

    Example:
    --------

    """
    est = numpy.zeros(pred_surv.shape)

    # get jacknife pseudovals for whole population if internal validation
    # get jacknife pseudovals for one complete sample in case of repeated CV.
    if pseudovals:
        est = _jacknife(fu_time, time, dead)
    else:
        est = 1 - dead

    # sort predicted probability of survival
    order = numpy.argsort(pred_surv)
    est_ord = est[order]
    pred_surv_ord = pred_surv[order]

    # Truncate pred_surv_ord at trunc[0/1]
    trunc_ind = numpy.logical_and(pred_surv_ord >= numpy.percentile(pred_surv_ord,
trunc[0]),
                                                    pred_surv_ord <=
numpy.percentile(pred_surv_ord, trunc[1]))
    pred_surv_ord = pred_surv_ord[trunc_ind]
    est_ord = est_ord[trunc_ind]
```

```
    # fit lowess
    fig, ax = _loess_smoothed_plot(y_plot=est_ord, x_plot=pred_surv_ord,
                                   fig=fig, ax=ax,
                                   ci_alpha=ci_alpha,
                                   plot_ci=ci,
                                   markersize=markersize,
                                   linestyle=linestyle,
                                   alpha=alpha,
                                   color=color,
                                   label=label)
    return fig, ax


def _jacknife(fu_time, time, dead, verbose=True):
    """
    calc pseudovalues according to TA Gerds
    https://rdrr.io/cran/prodlim/man/jackknife.html
    """
    # TODO: exchange lifelines KaplanMeierFitter with sksurv.nonparametric kaplan
meier estimator.
    from sklearn.model_selection import LeaveOneOut
    from lifelines import KaplanMeierFitter
    import time as t
    import sys
    # generate pseudovalues
    llo = LeaveOneOut()
    kmf = KaplanMeierFitter()
    kmf.fit(time, event_observed=dead)
    kme = numpy.interp(fu_time, kmf.survival_function_.index.values,
                       kmf.survival_function_.KM_estimate)  # linear interpolation

    pseudovals = []

    def get_pseudoval(train_index, test_index, kme=None):
        kmfi = KaplanMeierFitter()
        kmfi.fit(time[train_index], event_observed=dead[train_index])
        kmei = numpy.interp(fu_time, kmfi.survival_function_.index.values,
                            kmfi.survival_function_.KM_estimate)  # linear
interpolation
        return (time.shape[0] * kme) - ((time.shape[0] - 1) * kmei)

    print('start')
    start = t.time()
    with Parallel(n_jobs=8, backend='loky') as parallel:
        get_pseudoval_kme = partial(get_pseudoval, **{'kme': kme})
        # runs = [(train_index, test_index) for train_index, test_index in
llo.split(time)]
        res = parallel(delayed(get_pseudoval_kme)(train_index, test_index, )
                       for train_index, test_index in llo.split(time))
        pseudovals.append(res)
    print('finished in %s secs' % '{:6.1f}'.format(t.time() - start))
    return numpy.array(pseudovals).squeeze()


def _loess_smoothed_plot(x_plot, y_plot,
                         fig, ax,
                         color='#377eb8',
                         label="Loess-smoothed curve",
                         alpha=0.7,
                         linestyle='solid',
                         markersize=2.,
                         ci_alpha=.05,
                         plot_ci: bool = False,
                         ):
    lowess_fitter = lowess.Lowess()
    lowess_fitter.fit(x_plot, y_plot, frac=0.75, robust_iters=1)
```

```
    x_pred = numpy.linspace(x_plot.min(), x_plot.max(), 1000)
    y_pred = lowess_fitter.predict(x_pred)

    ax.plot(x_pred, y_pred, c=color, label=label, alpha=alpha,
markersize=markersize,
            linestyle=linestyle)
    if plot_ci:
        df_quantiles = lowess.quantile_model(x_plot, y_plot, frac=0., num_fits=100,
                                             qs=[ci_alpha / 2, 1 - (ci_alpha / 2)],
                                             robust_iters=1)
        ax.fill_between(df_quantiles.index, df_quantiles[ci_alpha / 2],
df_quantiles[1 - (ci_alpha / 2)],
                        edgecolor=color, facecolor=color, alpha=0.5 * alpha,
                        antialiased=True)
    return fig, ax


def calibration_curve(survival_train, survival_test, estimate, times,
                      fu_time,
                      fig=None, ax=None,
                      n_bins=10,
                      my_alpha=0.7,
                      my_markersize=4.,
                      color='#377eb8',
                      label='Actual versus predicted survival probability with 95%
CI',
                      ci=True,
                      ci_alpha=.05,
                      pseudovals=True,
                      loess=True,
                      internal_validation=True,
                      scatter=False,
                      ):
    """
    A calibration plot based on pseudovalues (pseudovals = True):
    A product limit estimator (Kaplan-Meier) is used to generate a jackknife
pseudo-value
    for the i'th observation, by calculating the product limit estimate for
    a n-1 subsample without the i'th observation.
    Pseudovalues are then calculated by:

        est_i(t) = n * (prodlim(t)) - (n-1) * prodlim_i(t)

    where prodlim is the KM estimate for the whole sample and prodlim_i is the
    one applied to the subsample without i'th observation.

    see r package prodlim https://rdrr.io/cran/prodlim/man/jackknife.html
    see also    Gerds et al.,Calibration plots for risk prediction models in the
presence of competing risks., https://doi.org/10.1002/sim.6152
                Graw et al., https://doi.org/10.1007/s10985-008-9107-z

    We combined the pseudovalues approach with loess smoothing
(https://dx.doi.org/10.1002%2Fsim.5941)
    Note: span = 0.75 (default) works good according to
https://dx.doi.org/10.1002%2Fsim.5941

    Parameters:
    -----------
    survival_train : structured array, shape = (n_train_samples,)
        Survival times for training data to estimate if training
        and testing data are drawn from same sample.
        Set internal_validation to True in this case.
        Otherwise, use surival_test again as input.
        A structured array containing the binary event indicator
        as first field, and time of event or time of censoring as
        second field.
```

```
    survival_test : structured array, shape = (n_samples,)
        Survival times of test data.
        A structured array containing the binary event indicator
        as first field, and time of event or time of censoring as
        second field.

    estimate : array-like, shape = (n_samples,n_times)
        Estimated risk of experiencing an event for test data at `times`.

    times : array-like, shape = (n_times,)
        The time points for which the predicted Survival function
        is calculated and interpolation for a specific follow-up-time
        is calculated. Values must be
        within the range of follow-up times of the test data
        `survival_test`.

    fu_time : float,
        The timepoint for which the calibration curve should be plotted.

    * fig : plt.Figure
        matplotlib's figure object, where the plot should be plotted at

    * ax : plt.Axes
        matplotlib's axes object, where the plot should be plotted at


    * color: str
        color for the plot

    * label: str
        label of the plot

    * trunc: int / float
        truncate pred_surv at trunc'th percentile.

    * my_alpha: float
        alpha transluicency for plot

    * my_markersize: float
        markersize for pyplot

    * ci: bool
        controls, whether confidence intervals should be calculated

    * ci_alpha: float
        confidence niveau, default 0.05
    * loess: bool
        use loess smoothing (recommended) or k nearest neighbors regression?
    * pseudovals: bool
        controls, whether pseudovals or binning approach should be used.
    * internal_validation: bool
        survival_train and survival_test are considered as beeing drawn
        from same sample and are both used for calculation of Kaplan-Meier-
estimates

    Returns:
    --------
        * fig, ax: tuple
            Figure and Axes object, where the curve is plotted at

    Example:
    --------
    @TODO: give a MWE.
    """
    # check fig, ax obj
```

```
    if fig is None or ax is None:
        fig, ax = plt.subplots()
    # check survival arrays for test_data
    test_event, test_time = check_y_survival(survival_test)
    train_event, train_time = check_y_survival(survival_train)
    times = check_array(numpy.atleast_1d(times), ensure_2d=False,
dtype=test_time.dtype)

    if internal_validation:
        test_time_traintest = numpy.concatenate(test_time, train_time)
        test_event_traintest = numpy.concatenate(test_event, train_event)
    else:
        test_time_traintest = train_time
        test_event_traintest = train_event
    # interpolate predicted survival at fu_time.
    # pred_surv = _interpolate_survfunc(times, fu_time, estimate,)
    pred_surv = estimate

    # sort by pred_surv in ascending order
    order = numpy.argsort(pred_surv)
    pred_surv = pred_surv[order]
    test_time = test_time[order]
    test_event = test_event[order]
    #    test_time_traintest = test_time_traintest[order]
    #    test_event_traintest = test_event_traintest[order]
    if numpy.greater(test_time[~test_event], fu_time).all():  # greater to allow
for day-wise (minor) interval-censoring
        print('No right-censoring at fu_time (%s) --> no jacknife pseudovalues are
used' % '{:3d}'.format(fu_time))
        pseudovals = False
    if loess:  # use pseudovals approach with loess smoothing
        fig, ax = _calib_plot_loess(fig, ax, fu_time,
                                    pred_surv, test_time, test_event,
                                    test_time_traintest, test_event_traintest,
                                    color=color,
                                    label=label,
                                    ci=ci,
                                    alpha=my_alpha,
        45                           markersize=my_markersize,
                                    markertype='-',
                                    loess=loess,
                                    pseudovals=pseudovals,
                                    scatter=scatter, )
    else:  # use traditional binning (visual analog to Hosmer-Lemshaw-test)
        fig, ax, (pred, actual, ci_upper, ci_lower) = _calib_plot(fig, ax, fu_time,
n_bins,
                                                                  pred_surv,
test_time, test_event,
                                                                  color=color,
                                                                  label=label,
                                                                  error_bars=ci,
                                                                  alpha=my_alpha,

markersize=my_markersize,
                                                                  markertype='-')

    return fig, ax
```

9 Appendix B

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Dissertation selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Die Dissertation ist bisher keiner anderen Fakultät, keiner anderen wissenschaftlichen Einrichtung vorgelegt worden.

Ich erkläre, dass ich bisher kein Promotionsverfahren erfolglos beendet habe und dass eine Aberkennung eines bereits erworbenen Doktorgrades nicht vorliegt.

Datum

Unterschrift