Multi Target XGBoost Cash Flow Prediction

An Efficient Machine Learning Algorithm For Future Liability Projections

by

Sebastiaan van Schagen

to obtain the degree of Master of Science at the Delft University of Technology, to be defended publicly on Wednesday July 26, 2023 at 10:30 AM.

Supervisor (TUDelft):	Dr. Ir. L.E. Meester
Supervisor (Triple A):	M. Visser MSc actuaris AG
Chair of department:	Prof. Dr. A. Papapantoleon
Thesis committee member:	Dr. Ir. J. Bierkens
Faculty:	EEMCS
Master programme:	Applied Mathematics
Specialisation:	Financial Engineering
Affiliation:	Triple A Risk Finance

An electronic version of this thesis is available at https://repository.tudelft.nl.

Abstract

Insurers are required to have buffers to be able to meet financial obligations that result from their portfolios, which are determined using a cash flow model. The input of such a cash flow model consists among of things, of two mortality tables and the portfolio of an insurer. Mortality rates are simulated using the Lee-Carter model. These simulated rates are in turn used to simulate the cash flow corresponding to a portfolio. This results in one possibility of incoming and outgoing money over a period of time. Lots of simulations are required to get a reliable estimate for the future cash flow which is (depending on the number of simulations) computationally heavy and therefore time consuming. The calculation time is decreased by applying an extreme gradient boosting (XGBoost) machine learning method in which cash flows are considered target variables and the mortality tables are considered features of the model. The trained XGBoost model can predict the cash flows based on the mortality tables. The standard XGBoost model is extended to a multi-target regression model which is able to predict multiple target variables at once. This XGBoost model reduces the computation time and ensures that 99.5% of the predictions deviates within either 1% or 0.5% of the observed values. XGBoost gives a good method of determining a reliable estimate of the future cash flow.

Keywords: *Cash flow simulation, Lee-Carter model, Extreme Gradient Boosting, Multi-Target Regression, Solvency II, Machine Learning*

Preface

The past year I have authored this thesis as a final assignment in obtaining my master's degree in applied mathematics at the Technical University of Delft. This thesis is written in collaboration with Triple A - Risk Finance, for which I am grateful. A lot of people supported me during this period. Firstly, I want to thank my daily supervisor Ludolf Meester for the guidance during this project. The meetings helped me to get more insight in the general approach of a thesis project as well as more subject oriented insights. Secondly, I want to thank Martijn Visser for accompanying me on this adventure. The meetings were always useful and gave me better understanding about the problem and the general financial world as well as machine learning. I would like to thank Antonis Papapantoleon and Joris Bierkens as well for taking part in my thesis committee.

I would like to thank colleagues at Triple A - Risk Finance as well for always being understanding and helpful. Lastly, I want to thank my friends and family. They did motivate me during this project, on top of that they proofread the thesis at multiple occasions and helped me to improve the thesis. Even though some did not have any affinity with the financial world, machine learning or mathematics.

Sebastiaan van Schagen
July 2023, Delft

Contents

Symbols and Abbreviations

Table 1: Symbols, abbreviations, and definitions.

Symbol	Description
X	Highest age taken into account in the projected mortality tables.
T	Last year taken into account in calibrating the Lee-Carter model.
$M_{x,t}$	Mortality rate of a person of age x in year t who dies during that year.
$\mathbf{M}$	Mortality table of which $M_{x,t}$ are the elements. This matrix consists of $x_{\max} + 1$ rows and $t_{\max} - t_{\min}$ columns.
CF	The difference between incoming and outgoing liquid assets of a company known as the cash flow.
BEL	Best estimate liability under Solvency II representing the expected discounted cash flow with the risk-free interest rate.
SCR	Solvency capital requirement under Solvency II representing the maximum amount expected to be lost over a one-year time horizon with 99.5% accuracy. Solvency II has defined SCRs for different kind of risks.
RM	Risk margin under Solvency II representing the potential costs of transferring insurance obligations to a third parties in case of default. The RM is calculated using the SCR.
$\text{Corr}_{i,j}$	Correlation between the SCRs of different subgroup i and j.
BSCR	Basic solvency capital requirement under Solvency II represents the aggregated SCRs of all subgroups.
CoC	Cost of capital gives the costs of funds of a company. The CoC is generally taken as six percent.
a_x	Average logarithmic mortality rate for age x used in the Lee-Carter model.
$\mathbf{a}$	A vector of length $x_{\max} + 1$ consisting of average logarithmic mortality rates a_x.
b_x	Relative mortality at age x. This value influences the mortality trend ($\mathbf{k}$) according to whether change at a given age is faster or slower than the original mortality trend.
$\mathbf{b}$	Vector of length $x_{\max} + 1$ consisting of the relative mortality b_x
k_t	Mortality trend in year t used in the Lee-Carter model. The mortality trends reflect the changes in in morality rates over the years.
$\mathbf{k}$	Vector of length $t_{\max} - t_{\min}$ with elements $\mathbf{k}_t$.
$\epsilon_{x,t}$	Error term of the Lee-Carter model with zero mean and finite variance for age x and year t
$\mathbf{U}$	Matrix consisting of left eigenvectors of a matrix $\mathbf{A}\mathbf{A}^T$ obtained using a singular value decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$.
$\mathbf{U}_i$	The i-th left eigenvector given as the i-th column of $\mathbf{U}$.

Continued on next page

Table 1: Symbols, abbreviations, and definitions. (Continued)

Symbol	Description
$\mathbf{\Sigma}$	Diagonal matrix consisting of eigenvalues of of a matrix $\mathbf{A}\mathbf{A}^T$ obtained using a singular value decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$.
σ_i	i singular value given by the i-th diagonal element of matrix $\mathbf{\Sigma}$
$\mathbf{V}$	Matrix consisting of right eigenvectors of a matrix $\mathbf{A}\mathbf{A}^T$ obtained using a singular value decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$.
$\mathbf{V}_i$	The i-th right eigenvector given as the i-th column of $\mathbf{V}$.
X_t	Observation of a time series at time t.
r_t	Residual of observation at time t
h	Sets the length of a time series forecast and is known as a lag of h.
ε_t	Error term used in a time series model given as white noise.
p	Order of auto-regressive terms in AR, ARMA and ARIMA models.
d	Degree of differencing in ARIMA model.
q	Order of moving average terms in MA, ARMA and ARIMA models.
λ	Regularisation parameter withing gradient boosting which reduces the prediction's sensitivity to individual observations.
γ	Leaves of a tree which have a gain smaller than this value are removed from a tree. This parameter is known as the pruning parameter.
η	Gives the importance of every tree for a given an additive model. Every tree is multiplied with η and added together to form one final model. This parameter is known as the learning parameter.
$\hat{y}_i$	Prediction of target variable y_i given by a machine learning method.
$\hat{\mathbf{y}}_i$	Prediction of target vector $\mathbf{y}_i$ given by a multi-target machine learning method.

1

Introduction

Most people are insured for several reasons, some insurances are required by law, while others are on a voluntary basis. The insured person is known as the policy holder. An example of such an insurance is a live insurance in which the insurer pays a predefined amount to relatives of the policy holder when the policy holder dies within a certain period. A different kind of insurance can have opposite working. For example, a policy holder only gets money if they survive past a certain age. In both examples pay policy holders a premium for this service. The determination of this premium can however be difficult as one works with future events which are not known at current time like the death of the policy holder. Insurers are required to have sufficient buffers to be able to meet financial obligations that result from their portfolios. To fulfil all future capital requirements, measures of the future financial situation need to be made. One of the measures consists of the discounted incoming cash flow over a given time horizon in comparison to the outstanding liabilities over the same period of time, known as the liability cash flow. The calculation of the cash flow is changed since the introduction of Solvency II in 2016 (DNB, 2022). Solvency II regulations offer two options to determine the cash flow. The first option is prescribed by Solvency II and is denoted as the standard formula. As an alternative, insurers are allowed to use a self-created internal model. All internal models need to be approved by the regulating party and need to fulfil requirements given by the Solvency II regulations. Both modelling approaches estimate the same risk measures. The first measure is given as the expected discounted cash flow and is known as the best estimate liability (BEL). The second measure of risk is taken as the maximum amount expected to be lost over a one-year time horizon with 99.5% accuracy, known as the 99.5% one-year value at risk (VaR). This measure of risk is under Solvency II known as the risk margin.

All important data of policy holders is combined into one or multiple portfolio's. A standard portfolio consists of characteristics such as age, gender, premium, payments, sum assured and more. All of these influence the cash flow in a way. The portfolio's can be created specifically for one type of insurance or they can consist of several types of insurances. Independent of the choice of model, one needs a portfolio as well as the mortality rates to determine the cash flow. A portfolio is only known up till the present day. Most portfolio's do not change that much in a small time period and can be taken as time independent for a small horizon. Similarly, the mortality tables are not known for the future and need to be simulated in order to get a mortality rates corresponding to a future time. Mortality rates corresponding to different ages and years are combined into gender specific mortality tables. A simulated male and female mortality table in combination with the current portfolio can be used to determine one possible future liability cash flow. Instead of only using one simulation, multiple simulations can be used to get more insight in the possible situations and thus in the possible future cash flows. This way results in more information about the volatility of the future cash flow. To get a reliable estimate of the future cash flow, a lot of simulated mortality tables are needed.

Lee and Carter have created a forecasting model applicable for mortality rates which is mostly used in the financial world (Lee & Carter, 1992). It is an easy to understand model consisting of only three vectors. The Lee-Carter model is calibrated on historical data and uses an autoregressive integrated moving average (ARIMA) model to forecast the mortality trend. Based on this forecast trend, mortality tables for future time periods can be simulated. These simulated tables can in turn be used to determine possible future cash flows. Doing this consecutively results in lots of possible cash flows and a reliable estimate of the cash flow. On top of this, a confidence interval for the estimate can be determined. Although the Lee-Carter model is often used in mortality table simulation, it does have some shortcomings. For example, it assumes a fixed mortality trend, while in some cases a variable mortality trend is preferred.

A second method of forecasting is using a model which assumes a stochastic mortality trend. Börger created such a model based on the Lee-Carter model (Börger, 2010). The main advantage of this method is that it incorporates dependencies between different age groups. It is however more complicated to interpret and to forecast.

Independent of the forecasting method, it is a computationally heavy process to determine the cash flow for all simulated tables. Determining only one possible cash flow using two tables is already computationally heavy. As an example, for a decently sized portfolio consisting of one hundred thousand policy holders it could take tens of minutes to determine the future cash flow. Doing this for all simulated tables, one could end up with days of calculations. By considering less simulated tables, the calculation time is reduced, while the accuracy of the estimate decreases. This approach is nowadays used by some insurers, in which sometimes only one mortality table is considered. This does however not give a good estimate of the real future liability cash flow. Other insurers do, however, consider a large set of simulated mortality tables. Insurers have to deal with this complexity–accuracy trade-off.

Another option, which is not as active in the market, is applying machine learning. Instead of calculating the cash flow of all the simulations, only a smaller set of simulated tables needs to be used as input for the cash flow model. These simulated tables and their corresponding cash flow values can in turn be used as a training set for a machine learning algorithm. The trained machine learning model gives predictions for the remaining forecast mortality tables. Although this method is not used as often in the financial world, it might result in a less time-consuming calculation. Recently, there has been some research on this modeling method. Castellani et al. discuss the use of machine learning for Solvency II (Castellani et al., 2018). Fiore et al. focus only on the tuning of a neural network to determine the cash flows (Fiore et al., 2018). Both articles show the possibilities of machine learning in decreasing the calculation time while keeping a decent accuracy.

There are a lot of possible machine learning algorithms to consider. A neural neural network as used by Fiore et al. is difficult to understand, while a simple algorithm might result in worse performance. A regression tree is an example of such a simple algorithm and is known as a weak learner. The performance of a regression tree is not sufficient for most cases. Multiple weak learners can be combined into one model which is known as ensemble learning. Extreme gradient boosting (XGBoost) is such a ensemble learning model which combines multiple regression trees to one well performing model. On top of this, XGBoost is optimised for large data sets with a lot of characteristics. Besides, XGBoost uses a lot of parallel calculations in comparison to standard gradient boosting, further decreasing the computation time. Since the mortality tables consist of lots of data, XGBoost is a good candidate to predict the future cash flow.

Performance of a machine learning method is normally determined using a metric such as the root mean squared error (RMSE) or just by looking at the residuals. The correctness of this model is tested using prescribed requirements. By decreasing the computation time significantly, the machine learning prediction is allowed to deviate a small percentage in comparison to the real value. The usage of the machine learning approximation is roughly divided into two parts. On the one hand, it is used as a management tool to get a value of the business performance. Based on this

value can insurers make changes in business operations. As this is mainly used for quick changes, it is not that strict. It is required that the prediction deviates within 1% or less of the real value in 99.5% of the cases. On the other hand, the XGBoost approach can used as reporting tool. Every insurer has to keep an overview of their future stability including the cash flows. As this is an official report, the prediction must be more accurate than in the management case. For reporting purposes, it is required that in 99.5% of the cases, the prediction deviates within 0.5% of the real value. Even though it is only a difference of 0.5%, one should not neglect the impact as will be observed. It is required that in both cases the computation time is reduced by at least a factor of two. The most time is lost in creating the training data as determining the cash flow for a lot of mortality tables is time consuming. Hence, by minimizing the training set, while still satisfying the requirements, a new method of estimating the future liability cash flow is defined. To model and test this is a standard portfolio consisting of one hundred thousand Dutch policyholder used. This portfolio consists of different kind of life insurance products such as endowment, term life and whole life insurance. Ages of policy holders range between 0 and 110. The man to woman ratio is almost 1 to 1.

The possibilities of using XGBoost as a method to determine the future liability cash flow are investigated in this research. First, a short introduction of the cash flow model is given, as well as mortality tables, in Chapter 2. Second, the Lee-Carter forecasting model is investigated in Chapter 3. In which the design of the model as well as the uncertainties concerning the model are discussed. This is followed by multiple machine learning algorithms in Chapter 4. In this chapter is a new XGBoost method defined which allows one XGBoost model to predict multiple values at once. Based on the Lee-Carter model and the XGBoost model, results are generated in Chapter 5. At last, a conclusion as well as recommendations for future research are given in Chapter 6.

2

Cash Flow Model

Cash flow models are used to forecast future cash positions, which is done by comparing expected future incoming and outgoing cash. It creates a measure of the solvency of the business. By estimating the future cash flow, business can change business plans to optimize solvency of the business. If there is any uncertainty or risk involved in the incoming and outgoing cash, one needs to consider multiple scenario's to ensure a reliable estimate of the future cash position.

2.1. Mortality Table

Insurers deal with multiple uncertainties while valuing life insurances. One of these uncertainties is the probability of death of the policyholder. The probability of death, known as the mortality rate, of past years can be estimated using population data. The ratio between past population deaths and population size for a specific age x and a given year t is known as the observed mortality rate. These mortality rates are denoted by $q_{x,t}$. Doing so for every combination of age and year, one obtains a mortality table. An example of such a morality table is given in Table 2.1. In the mortality table, the probability of death of a policyholder of age 30 in year 2000 is given by:

$$M_{30,2000} = \mathbb{P}(x : x \text{ is of age 30 in year 2000 and dies during that year}) = q_{30,2000}.$$

Both male and female have their own corresponding mortality table.

		Year				
		1900	1901	...	2018	2019
Age	0	$M_{0,1900}$	$M_{0,1901}$	...	$M_{0,2018}$	$M_{0,2019}$
	1	$M_{1,1900}$	$M_{1,1901}$	...	$M_{1,2018}$	$M_{1,2019}$
	⋮	⋮	⋮	⋱	⋮	⋮
	119	$M_{119,1900}$	$M_{119,1901}$	...	$M_{119,2018}$	$M_{119,2019}$
	120	$M_{120,1900}$	$M_{120,1901}$	...	$M_{120,2018}$	$M_{120,2019}$

Table 2.1: An overview of a mortality table with ages varying from 0 to 120 and years varying between 1900 and 2019. $M_{x,t}$ denotes the mortality rate of someone at age x in year t.

Future population deaths and population size are unknown and need to be approximated or simulated. The mortality rates are forecast using a forecasting method which simulates possible mortality rates. The Lee-Carter method is such a forecasting method which is often used in the insurance world. Forecasting with use of the Lee-Carter model is shown in Chapter 3.

2.2. Solvency II

All insurers have exposure to risk based on their policyholders. Life insurers have to deal with deaths of the policyholders. A death results in a change of the cash flow, which in turn has a (slight) impact on the financial situation of the insurer. Death of the policyholder in a life product results in a negative cash flow, while the same death in a pension results in a positive cash flow. Insurers are required to have sufficient buffers to be able to meet financial obligations that result from their policy holders.

The insurers are required to have enough capital to ensure the payments to the policyholder. To protect the insurers, regulatory parties have introduced a supervisory framework for the insurers. The aim of this framework is to create a uniform way of classifying and quantifying risk across all insurers of the European Union. In January 2016, the European Commission improved this framework to the now used Solvency II framework (DNB, 2022).

One of the main parts of Solvency II is the capital requirement which give estimates of the financial situation of an insurer. The first risk qualifying estimate under Solvency II is the expected discounted cash flow, denoted as the best estimate liability (BEL). On top of this, insurers need to have a buffer to overcome (extreme) losses. Solvency II requires the 99.5% confidence interval to survive these losses over a one-year horizon, which is given as the 99.5% one-year value at risk (VaR). This measure is denoted as the solvency capital requirement (SCR). In case of default, an insurer should transfer the insurance obligations to third parties. The cost required to do so is denoted as the risk margin (RM). Calculation of the RM is mostly based on aggregating multiple SCRs.

Solvency II guidelines describe a method to determine the SCR called the standard formula. Usage of the standard formula is not obligatory; insurers are free to create their own internal model. Every internal model should be approved by the regulating authorities. The input of both kind of models consists of data of all policyholders. This data is combined into a portfolio and consist among others of age, gender, premium, and payment schedule. On top of this, future mortality rates are needed. The mortality rates are given in the form of a male and female mortality table in the format described in 2.1.

2.3. The Standard Formula

At the introduction of Solvency II, the European Commission created a standard method for determining risk estimates. This method is known as the standard formula and determines the SCR, and RM. The BEL can be calculated for a given combination of male and female mortality table and portfolio. The SCR is on the other hand more complicated. One way of determining the SCR is by simulating different scenarios. However, this becomes time consuming. To overcome this and to simplify the estimation, the standard formula uses estimates of the 99.5% VaR. These estimates are described as stress tests.

2.3.1. Stress Tests

The calculation of the SCR is divided into sub-groups. For every sub-group is a stress test defined which can be used to get an estimation of the SCR. These stress tests correspond to the 99.5% VaR if one uses simulations and are calibrated by the European Commission. The stress tests are sensitive to changes and are updated from time to time (EIOPA, 2021).

The stress test is applied to the cash flow model and the stressed BEL is determined. The calculation of the SCR is determined by taking the difference between the BEL of the unstressed situation and the BEL of the stressed situations.

$$\text{SCR}_k = \text{BEL}_{k,\text{Stressed}} - \text{BEL}_{k,\text{Unstressed}} \tag{2.1}$$

Mortality tables play an important role in the sub-groups mortality, longevity, and catastrophe. These SCR are most relevant for this report. The stress test for mortality reflects uncertainties in deaths of the policy holders. It captures the risk more policyholders die during the projection than expected. The European Insurance and Occupational Pensions Authority (EIOPA) has calibrated

this stress test as a permanent increase of mortality rates of ten percent (CEIOPS, 2009).

$$q'_x = 1.1 \cdot q_x$$

Similarly, the stress test for sub-group longevity reflects uncertainties in mortality rates as well. It captures the risk of more policyholders living longer than expected (CEIOPS, 2009). EIOPA has calibrated this stress test as a permanent decrease of mortality rates with twenty percent.

$$q'_x = 0.8 \cdot q_x$$

Lastly, the catastrophe risk is the risk of extreme death events which are not captured by the mortality sub-group (CEIOPS, 2009). EIOPA has estimated the catastrophe stress test as in addition of 0.0015 to the mortality rates, for the next twelve months.

$$q'_x = q_x + 0.0015$$

2.3.2. Aggregation of SCRs

The individual determined SCRs are aggregated into one SCR of the parent group as given in Equation 2.2. The aggregation is based on correlations between the sub-groups. $\text{Corr}_{k,l}$ is given as an advice of the Committee of European Insurance and Occupational Pensions Supervisors (CEIOPS, 2010).

$$\text{SCR} = \sqrt{\sum_{k,l} \text{Corr}_{k,l} \cdot \text{SCR}_k \cdot \text{SCR}_l} \tag{2.2}$$

The SCRs of all risk groups are again aggregated to one value. This value is the Basis Solvency Capital Requirement (BSCR). A prescribed correlation matrix is given by Solvency II, see Table 2.2.

i,j	Market	Default	Life	Health	Non-life
Market	1	0,25	0,25	0,25	0,25
Default	0,25	1	0,25	0,25	0,5
Life	0,25	0,25	1	0,25	0
Health	0,25	0,25	0,25	1	0
Non-life	0,25	0,5	0	0	1

Table 2.2: Correlation coefficients of risk groups.

In the standard formula the BSCR is determined using Equation 2.2. In this equation, k and l represent the different risk group and is $\text{Corr}_{k,l}$ given in Table 2.2. Besides the BSCR, insurers need to determine the risk of loss originating from failed internal processes, personnel or from external events. This risk is denoted as the operational risk. On top of this, the compensation of unexpected losses as well as deferred taxes is required, in Solvency II known as the adjustment risk. (EIOPA, 2009). The sum of the BSCR, operational risk and adjustments forms the final SCR (Equation 2.3).

$$\text{SCR} = \text{BSCR} + \text{Operational Risk} + \text{Adjustments} \tag{2.3}$$

The one-year RM is determined from the BSCR, as in Equation 2.4. It is defined as the product of the discounted BSCR (by risk free interest rate r) and the cost of capital (CoC). The CoC is generally taken as six percent.

$$\text{RM} = \text{CoC} \cdot \frac{\text{BSCR}}{1+r} \tag{2.4}$$

In most instances, a larger time horizon is desirable. The BSCR can be determined for a larger time horizon (t) BSCR(t). Doing so and altering Equation 2.4, the multiple year RM can be determined.

3

Lee-Carter Forecasting

Forecasting of mortality tables can be done in multiple ways. One option is using the Lee-Carter model introduced in 1992 (Lee & Carter, 1992). This model is mostly used in forecasting mortality tables due to its simplicity of calibrating and choice model parameters. Over the years, multiple extensions for this model have been created such as the Börger model (Börger, 2010).

The Lee-Carter model determines the mortality rate ($\mathbf{M}_{x,t}$) of age x in year t with use of Equation 3.1. Age x ranges from zero to some predefined X, similar, year t ranges from one to some predefined T. Lee and Carter define an error term $\epsilon_{x,t}$ with zero expectation and variance σ^2. All error terms are independent identical normally distributed $\epsilon_{x,t} \overset{iid}{\sim} \mathcal{N}(0, \sigma^2)$

$$\ln(M_{x,t}) = a_x + b_x k_t + \epsilon_{x,t} \qquad x \in [0, X] \text{ and } t \in [1, T] \tag{3.1}$$

Suppose a solution to Equation 3.1 is given by the combination of vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{k}$. As one observes, for any scalar c, the combination $\mathbf{a}$, $c \cdot \mathbf{b}$ and $\frac{1}{c}\mathbf{k}$ forms a solution as well. Similarly, $\mathbf{a} - c \cdot \mathbf{b}$, $\mathbf{b}$ and $\mathbf{k} + c$ forms a solution as well. Hence, to ensure uniqueness of the solution, Constraints 3.2a and 3.2b are introduced (Lee & Carter, 1992).

$$\sum_{x=0}^{X} b_x = 1 \tag{3.2a}$$

$$\sum_{t=1}^{T} k_t = 0 \tag{3.2b}$$

These constraints make sure that the model has unique solutions. Let both combinations $\mathbf{a}_1$, $\mathbf{b}_1$ and $\mathbf{k}_1$ as well as $\mathbf{a}_2$, $\mathbf{b}_2$ and $\mathbf{k}_2$ be solutions to the model. Then, one can show, that $\mathbf{a}_1 = \mathbf{a}_2$, $\mathbf{b}_1 = \mathbf{b}_2$ and $\mathbf{k}_1 = \mathbf{k}_2$. The combination of the model function and constraints can be used to create equations for $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{k}$. These equations are calibrated on historical data. First, $\mathbf{a}$ is determined by summing Equation 3.1 over t and using Constraint 3.2b. Doings so results in a function for $\mathbf{a}$.

$$\begin{aligned}
\sum_{t=1}^{T} \ln(M_{x,t}) &= \sum_{t=1}^{T} (a_x + b_x k_t + \epsilon_{x,t}) \\
&= T a_x + b_x \sum_{t=1}^{T} k_t \\
&\overset{3.2b}{=} T a_x
\end{aligned}$$

Hence a_x is taken as the average of $\ln(M_{x,t})$ over time. These values can be determined directly and are not dependent on uncertainties besides the standard inaccuracies of the historical data.

$$a_x = \frac{\sum_{t=1}^{T} \ln(M_{x,t})}{T} \tag{3.3}$$

By subtracting every column of matrix $\mathbf{M}$ with vector $\mathbf{a}$, a new matrix $\mathbf{A}$ is obtained as given in Equation 3.4. This matrix can in turn be used to determine both $\mathbf{b}$ and $\mathbf{k}$.

$$\mathbf{A} = \begin{pmatrix} \ln(\mathbf{M}_{0,1}) - \mathbf{a}_0 & \dots & \ln(\mathbf{M}_{0,T}) - \mathbf{a}_0 \\ \vdots & \ddots & \vdots \\ \ln(\mathbf{M}_{X,1}) - \mathbf{a}_X & \dots & \ln(\mathbf{M}_{X,T}) - \mathbf{a}_X \end{pmatrix} \tag{3.4}$$

By applying singular value decomposition (SVD) on real matrix A, matrix A can be written as $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. $\mathbf{U}$ is an $(X+1) \times (X+1)$ orthogonal matrix, $\mathbf{V}$ is an $T \times T$ orthogonal matrix and $\mathbf{\Sigma}$ is an $(X+1) \times T$ diagonal matrix.

$$\mathbf{A} = \begin{pmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_{X+1} \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 & \dots & \mathbf{v}_T \end{pmatrix}^T$$

The Lee-Carter estimation of vectors $\mathbf{k}$ and $\mathbf{b}$ can be defined to be a least squares method. Eckart and Young introduced a theorem of determining the best approximation, with a specific rank, of a given matrix (Eckart & Young, 1936). This theorem states that the best approximation of rank p is given as the first p components of the singular value decomposition. They proved that this method works for the Frobenium norm. Mirsky later extended the theorem to the Eckart-Young-Mirsky Theorem which is applicable for multiple norms (Mirsky, 1960). For this report, the spectral norm is used. The definition of the spectral norm is given in Definition 1.

Definition 1 (Spectral Norm)
The spectral norm $||\cdot||_2$ of a real matrix $\mathbf{A}$ is given as the largest eigenvalue of $\mathbf{A}^T \times \mathbf{A}$ or

$$||\mathbf{A}||_2 = \sigma_{max}(A)$$
$$||\mathbf{A}||_2 = max\frac{|\mathbf{A}\mathbf{x}|_2}{|\mathbf{x}|_2}$$

The proof of Eckart-Young-Mirsky using the spectral norm is given in Theorem 1.

Theorem 1 (Eckart-Young-Mirsky (Mirsky, 1960))
Given real matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $m \leq n$ of rank r with singular value decomposition

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \sum_{i=1}^{r} \mathbf{u}_i \sigma_i \mathbf{v}_i^T,$$

the best rank p approximation of A in the spectral norm, $||\cdot||_2$ is given by

$$\mathbf{A}_p = \sum_{i=1}^{p} \mathbf{u}_i \sigma_i \mathbf{v}_i^T$$

Proof: *Let $\mathbf{A}_p$ be as given above and one can determine the spectral norm of the difference between the real matrix and the rank p approximation.*

$$\begin{aligned}
||\mathbf{A} - \mathbf{A}_p||_2 &= \left|\left| \sum_{i=1}^{r} \mathbf{u}_i \sigma_i \mathbf{v}_i^T - \sum_{i=1}^{p} \mathbf{u}_i \sigma_i \mathbf{v}_i^T \right|\right|_2 \\
&= \left|\left| \sum_{i=p+1}^{r} \mathbf{u}_i \sigma_i \mathbf{v}_i^T \right|\right|_2 \\
&= \left|\left| \sum_{i=p+1}^{r} \mathbf{U}\ \mathrm{diag}(0, \ldots, 0, \sigma_{p+1}, \ldots \sigma_r)\ \mathbf{V}^T \right|\right|_2 \\
&= \sigma_{p+1}
\end{aligned}$$

All left to prove is the fact that matrix $\mathbf{A}_p$ is the best approximation of rank p. To this end, let matrix $\mathbf{B} \in \mathbb{R}^{m \times n}$ be a matrix of rank p. Let $\mathbf{w}$ be a vector of length one such that $\mathbf{Bw} = 0$ and $\mathbf{w} \in \mathrm{span}(\mathbf{v_1}, \ldots, \mathbf{v}_{p+1})$, then

$$\begin{aligned}
||\mathbf{A} - \mathbf{B}||_2^2 &\geq ||(\mathbf{A} - \mathbf{B})\mathbf{w}||_2^2 \\
&= ||\mathbf{Aw}||_2^2 \\
&= (w_1\sigma_1)^2 + \cdots + (w_{p+1}\sigma_{p+1})^2 \\
&\geq \sigma_{p+1}^2
\end{aligned}$$

Taking the square root at both sides implies $||\mathbf{A} - \mathbf{B}||_2 \geq \sigma_{p+1}$. The lower limit is attained at matrix $\mathbf{A}_p$, hence, one can conclude that matrix $\mathbf{A}_p$ is the best rank p matrix approximation of matrix $\mathbf{A}$. □

The Lee-Carter model uses a rank one approximation of matrix $\mathbf{A}$. Hence, by Eckart-Young-Mirsky Theorem, the least square estimate for matrix $\mathbf{A}$ of rank one is given by

$$\hat{\mathbf{A}} = \mathbf{u}_1 \sigma_1 \mathbf{v}_1^T. \tag{3.5}$$

The vectors $\mathbf{u}_1$ and $\mathbf{v}_1$ are the first left and first right eigenvectors given by the first column of matrix $\mathbf{U}$ and $\mathbf{V}$, respectively. σ_1 is the largest singular value given by the first diagonal element in σ. Combining this with Equation 3.1, one can let $\mathbf{b} = \frac{1}{\sum_{x=0}^{X} u_{1,x}} \mathbf{u}_1$ and $\mathbf{k} = \sum_{x=0}^{X} (u_{1,x})\, \sigma_1 \mathbf{v}_1$. Constraint 3.2a is satisfied since the values are divided by the sum of the vector elements of $\mathbf{U}_1$. Similarly, Constraint 3.2b is satisfied as well as is shown in Observation 1. With use of a forecasting method, future values of $\mathbf{k}_t$ will be determined.

Observation 1 (Elements of k sum to zero)
If $\mathbf{k} = \sum_{x=0}^{X} (u_{1,x})\, \sigma_1 \mathbf{v}_1$, then $\sum_{t=1}^{T} k_t = 0$. Both $\mathbf{u}_1$ and $\mathbf{v}_1$ are obtained from the SVD on matrix $\mathbf{A}$.

Proof: *Given matrix $\mathbf{A}$ as in Equation 3.4 whose rows sum up to one $\forall x \in \{1, \ldots, X\}\ \sum_{t=1}^{T} A_{x,t} = 0$. The SVD determines the right eigenvectors of $\mathbf{A}^T\mathbf{A}$. The (i, j)-th element of $\mathbf{A}^T\mathbf{A}$ can be written as $\sum_{x=0}^{X} A_{x,i}A_{x,j}$. Then by summing over t, one obtains the column sum of $\mathbf{A}^T\mathbf{A}$ as*

$$\sum_{t=1}^{T} \sum_{x=0}^{X} A_{x,i}A_{x,j} = \sum_{t=1}^{T} A_{x,i} \sum_{x=0}^{X} A_{x,j} = 0.$$

Let $\mathbf{v} \in \mathbb{R}^T$ be an arbitrary vector and for readability set $\mathbf{B} = \mathbf{A}^T\mathbf{A}$, then one obtains

$$\sum_{t=1}^{T}\sum_{s=1}^{T} B_{t,s}v_s = \sum_{s=1}^{T} v_s \sum_{t=1}^{T} B_{t,s} = 0.$$

Recall that an eigenvector is determined using $\mathbf{Bv} = \lambda\mathbf{v}$. Summing over the elements of both sides, one obtains

$$\sum_{t=1}^{T}\sum_{s=1}^{T} B_{t,s}v_s = \sum_{t=1}^{T} \lambda v_t.$$

The first term is equal to zero and λ is unequal to zero, hence $\sum_{t=1}^{T} \mathbf{v}_t$ must be equal to zero. This implies that $\sum_{t=1}^{T} k_t = 0$ for this choice of $\mathbf{k}$. □

3.1. ARIMA Model

Future values of $\mathbf{k}_t$ can be forecast using different forecasting techniques, which predict future instances based on past observations. Let X_{t-1} denote the observation of a time series at time $t-1$. Based on past observations $X_0, \ldots, X_{t-1}$ the value at time t, given by X_t is estimated.

One of the most straightforward forecasting technique is assuming a random step from the previous value. These steps are independent and identically distributed. This method is known as a random walk, in which the mean of all predictions is equal to the last observation. A drift term, which represents the average increase between time steps, can be introduced as an extension of this model.

This model can be extended by considering more past observations. An autoregressive (AR) model does this exactly. By defining a weight, ϕ for past observations and summing over these, one obtains an estimate for the future value. An AR model incorporates a stochastic term, ϵ_t, representing the error between the estimate and the correct value at time t. The sequence of error term has mean zero and has an autocorrelation of zero. It can be seen as a sequence of random numbers which cannot be predicted. Such processes are called white noise. Combining everything, Equation 3.6 is obtained, which is the general AR model of order p.

$$X_t = \sum_{i=1}^{p} \phi_i X_{t-i} + \epsilon_t \tag{3.6}$$

Instead of summing over some of the past observations, one could sum over the white noise error terms as well. By multiplying the error terms with a weight, θ and combining these, an estimate for the future error is determined. Based on the mean, μ, of the past observations and the future error estimate, predictions can be made, this model is known as the moving average model. The general MA model of order q is shown in Equation 3.7.

$$X_t = \mu + \sum_{i=1}^{q} \theta_i \epsilon_{t-1} + \epsilon_t \tag{3.7}$$

The AR and MA models can be combined into one model which incorporates both the weights for the observations as well as for the error terms. This model is known as the autoregressive moving average (ARMA) model. The general equation is obtained by summing over the weighted past observations and summing over the error terms. The stochastic term at time t is still present. The general ARMA model can be found in Equation 3.8 in which p denotes the order of the AR

part and q the order of the MA part.

$$X_t = \sum_{i=1}^{p} \phi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-1} + \epsilon_t \tag{3.8}$$

All of the above methods work best with stationary data observations. This is however not always realisable. To obtain a more reliable model, the difference between the observations can be taken as input observations. This differencing can happen over multiple time steps at once. This generalization of the ARMA model is given as the autoregressive integrated moving average (ARIMA) model. Based on data observations $X_0, \ldots, X_{t-2}, X_{t-1}$, the ARIMA consists of three parameters p,d, and q. p represents the order of the auto-regressive model. q represents the order of the moving average model. d represents the degree of the difference. If d equals zero, the ARIMA model becomes an ARMA model. The standard form of an ARIMA(p,d,q) model can be found in Equation 3.9.

$$X_t - X_{t-d} - \sum_{i=1}^{p} \left(\phi_i (X_{t-i} + X_{t-i-d})\right) = \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{3.9}$$

Equation 3.9 can be rewritten to Equation 3.10. Only the left side is dependent on the most recent value X_t, since every term on the right side has at least one backward operation. Note ϵ_t is still on the right side.

$$X_t = X_{t-d} + \sum_{i=1}^{p} \left(\phi_i (X_{t-i} + X_{t-i-d})\right) + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{3.10}$$

If one increments the time with value one, an estimate for the future value can be determined. Every observation up till time t is known, as well as the residuals. ϵ_{t+1} is unknown and set equal to zero. All other values of ϵ_t are set equal to the corresponding residuals (e_t). Consecutive increments result in a forecast over h lags.

3.1.1. Prediction Interval

The prediction interval of the ARIMA model can be determined as well. In order to do so, one needs the forecast errors given by the difference between the real value and the predicted value

$$e_t = X_t - \hat{X}_t. \tag{3.11}$$

By increasing the time with one time step, Equation 3.11 can be rewritten to:

$$X_{t+1} = \hat{X}_{t+1} + e_{t+1}. \tag{3.12}$$

$\hat{X}_{t+1}$ is the forecast and e_{t+1} is the unknown future error. By assuming future errors are similar to past errors, one can replace e_{t+1} with one of the past residuals. Depending on the number of past observations, a lot of choices for the current residual can be made. The new simulated data point can be used to repeat the procedure for e_{t+2}. Doing this for h lags into to future results in a h-lag forecast. All h lags have an residual term equal to a past observation. Doing this for multiple combinations of past residual gives a collection of possible future values. Taking the quantile of the collection results in the prediction interval. The obtained interval is denoted as a bootstrapped prediction interval.

If one however assumes that the errors are normally distributed, one could more easily determine the prediction interval. In such a case are the error terms over a h-lag forecast distributed as $\mathcal{N}(0, \hat{\sigma}_h^2)$ and is the prediction interval given by

$$\hat{X}_{t+h} \pm c\hat{\sigma}_h. \tag{3.13}$$

c is a scalar which influence the prediction interval. Setting $c = 1.96$ the 95% confidence interval is obtained. Similar, setting $c = 2.58$ gives the 99% confidence interval.

3.2. Uncertainties of $\mathbf{k}_t$

The Lee-Carter model has to deal with multiple uncertainties. One of the uncertainties is the way how k_t is forecast. The uncertainty of k_t might impact the complete accuracy of the Lee-Carter model. Lee and Carter describe the Lee-Carter model first with forecast using a random walk instead of an ARIMA model (Lee & Carter, 1992). Both the uncertainties of Lee-Carter with a random walk as well as ARIMA model will be discussed. Kleinow and Richards describe both methods (Kleinow & Richards, 2017).

3.2.1. Random Walk With Drift Uncertainties

By applying a random walk with drift (δ, value unknown), one can write k_t as:

$$k_{t+1} = \delta + k_t + \varepsilon_{t+1}, \tag{3.14}$$

in which $\varepsilon_{t+1} \sim N(0, \sigma^2)$, σ^2 is constant finite variance. Extending the random walk to a larger horizon, an h lag forecast can be obtained. By setting ε equal to zero, the central forecast is obtained. This central forecast is used in the remainder of this section. The central forecast is obtained by setting ε equal to zero

$$k_t(h) = h\delta + k_t \tag{3.15}$$

The drift is still unknown, hence an estimate is needed. The drift is estimated by the average of the differences:

$$\hat{\delta} = \frac{1}{T-1}\sum_{i=2}^{T}(k_i - k_{i-1}) = \frac{k_T - k_1}{T-1} \tag{3.16}$$

This estimator can be used in an estimator of h lags in the future for the central forecast, which can be written as

$$\hat{k}_t(h) = h\hat{\delta} + k_t. \tag{3.17}$$

Now, the mean squared predicted error can be calculated as done by Kleinow and Richards. They show that the mean squared predicted error can be divided into parameter uncertainty as well as volatility (Kleinow & Richards, 2017).

$$\begin{aligned}
\mathbb{E}[(\hat{k}_t(h) - k_{t+h})^2] &= \mathbb{E}[(\hat{k}_t(h) - k_t(h) + k_t(h) - k_{t+h})^2] \\
&= \mathbb{E}[(\hat{k}_t(h) - k_t(h))^2] + \mathbb{E}[(k_t(h) - k_{t+h})^2] + 2\mathbb{E}[\hat{k}_t(h) - k_t(h)]\mathbb{E}[k_t(h) - k_{t+h}] \\
&= \mathbb{E}[(\hat{k}_t(h) - k_t(h))^2] + \mathbb{E}[(k_t(h) - k_{t+h})^2] + 2\mathbb{E}[\hat{k}_t(h) - k_t(h)]\mathbb{E}\left[-\sum_{i=1}^{h}\varepsilon_{t+i}\right] \\
&= \mathbb{E}[(\hat{k}_t(h) - k_t(h))^2] + \mathbb{E}[(k_t(h) - k_{t+h})^2] \\
&= \mathrm{Var}(\hat{k}_t(h)) + \mathbb{E}\left[\left(\sum_{i=1}^{h}\varepsilon_{t+i}\right)^2\right] \\
&= h^2\mathrm{Var}(\hat{k}_t(h)) + h\sigma^2 \\
&= h^2\frac{\sigma^2}{t-1} + h\sigma^2 \\
&= \frac{h}{t-1}h\sigma^2 + h\sigma^2
\end{aligned} \tag{3.18}$$

The first part, $\frac{h^2\sigma^2}{t-1}$, is denoted as the parameter uncertainty. The second part, $h\sigma^2$, is denoted as the volatility (Kleinow & Richards, 2017). Hence, taking a smaller historical time horizon T while the projection horizon is larger results in a dominating parameter uncertainty. On the other hand, taking a larger historical time horizon while the projection horizon is small results in a dominating volatility.

3.2.2. ARIMA Model Uncertainties

Similar to the random walk with drift, parameter uncertainty of the ARIMA model can be estimated. Kleinow and Richards describe that ARIMA models with differencing parameters $d = 1$ are only relevant for mortality forecasting. Hence, only ARIMA model with a differencing order of one is considered (Kleinow & Richards, 2017). This suggest that the first difference can be seen as a stationary process. An ARIMA$(p, 1, q)$ model with mean δ can be written as:

$$k_t = k_{t-1} + \sum_{i=1}^{p} \left(\phi_i(k_{t-i} + k_{t-i-1})\right) + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \delta \tag{3.19}$$

This gives a method of forecasting the future values. Continuous applying this results in a forecast over h lags.

$$\begin{aligned} k_{t+h} &= k_{t+h-1} + \sum_{i=1}^{p} \left(\phi_i(k_{t+h-i} + k_{t+h-i-1})\right) + \epsilon_{t+h} + \sum_{i=1}^{q} \theta_i \epsilon_{t+h-i} + \delta \\ &= k_t + \sum_{j=1}^{h} \left(\sum_{i=1}^{p} \left(\phi_i(k_{t+j-i} + k_{t+j-i-1})\right) + \epsilon_{t+j} + \sum_{i=1}^{q} \theta_i \epsilon_{t+j-i} \right) + h\delta \end{aligned} \tag{3.20}$$

An estimate of the mean is determined as the average of the observations:

$$\hat{\delta} = \frac{1}{T} \sum_{i=1}^{T} k_i \tag{3.21}$$

Similar as with the random walk, a forecast for a model can be defined as given in Equation 3.22. The coefficients of the ARIMA model first need to be fitted on past observations.

$$\hat{k}_{t+h} = k_t + \sum_{j=1}^{h} \left(\sum_{i=1}^{p} \left(\phi_i(k_{t+j-i} + k_{t+j-i-1})\right) + \epsilon_{t+j} + \sum_{i=1}^{q} \theta_i \epsilon_{t+j-i} \right) + h\hat{\delta} \tag{3.22}$$

Kleinow and Richards show with a simulations study that the ARIMA model has better performance than the random walk. They observe how $\hat{\delta}$ can dominate the prediction of $\hat{k}_{t+h}$. The closer the coefficient of the ARIMA model are to zero, the quicker $\hat{\delta}$ dominates the forecast (Kleinow & Richards, 2017). Kleinow and Richards conclude that the uncertainty of the ARIMA model for the mortality rates is dominated by uncertainty of the drift and mean.

3.2.3. Consequences of Uncertainties

The uncertainty around k_t is highly dependent on the uncertainty of the drift and mean. Hence, one needs to be careful by estimating the correct ARIMA model. There are however more uncertainties concerning the Lee-Carter model.

Even when the perfect ARIMA model is found, there is still uncertainty in the data used to calibrate this model. Both $\mathbf{b}$ and $\mathbf{k}$ are the results of a low rank approximation. This complexity–accuracy trade-off can result in bad approximations and should be kept in mind for this research. The Lee-Carter model could be extended to incorporate more singular values. An example of this is given by Equation 3.23 in which the first s singular values are used and in which $\mathbf{b}_i$ and $\mathbf{k}_i$ represent the i-th SVD terms.

$$\ln(M_{x,t}) = a_x + b_{1,x}k_{1,t} + b_{2,x}k_{2,t} + \cdots + b_{s,x}k_{s,t} \tag{3.23}$$

All remaining SVD components are relegated to the error term. By taking all components of the SVD, a perfect estimate is obtained. This results in another complexity-accuracy trade-off. The first term of the SVD is dominant for the low rank approximation. The importance of the SVD

term decreases, the later the term is present in the SVD. Because of this, only taking the first SVD term already gives a decent estimate. Because of this, Lee and Carter use a order one low rank approximation, resulting in some uncertainty. The performance of the low rank approximation can however be improved by taking more SVD terms, which slightly complicates the model. Apart from these shortcomings, the structure of Lee-Carter is convenient as input for machine learning as will be discussed in Chapter 5.

4

Machine Learning

Machine learning is a branch of artificial intelligence which tries to train a model based on input data. A trained model in turn tries to predict target variables of an object based on characteristics or features of that same object. Machine learning is often applied to approximate a costly function. A trained model is fast in creating predictions, training the model can however take some time. The data used to train a model is known as the training data and consists of the features as well as observations of the target variables. Performance of a model is determined using test and validation data in which predictions are compared to the observed target values. In this chapter, multiple machine learning methods are described which build upon each other. First, regression trees are introduced which in turn can be used as building blocks for gradient boosting and extreme gradient boosting. Without loss of continuity, one can immediately continue to extreme gradient boosting in Section 4.3 or the multi target variant in Section 4.4.

4.1. Regression Trees

One of the most known machine learning model is a regression tree. Regression trees are mappings of binary decisions which lead to predictions of objects (Hastie et al., 2009). These binary decisions represent a separation in feature values. Regression trees are graphically represented as tree structures in which one branches off a starting node, the root. All internal nodes represent at least one feature, this choice of feature is not unique and the same feature can be relevant for multiple nodes. Every branch, connected to such a node, represents the feature outcome of a binary split. Lastly, terminal nodes (excluding the root) represent target values of the final decision. An example of a regression tree can be found in Figure 4.1. Regression trees are widely used in machine learning. They can be used independent but are however mostly used in combination with other methods as they are known as weak learners, meaning that they not always give a reliable prediction. Combining multiple trees will however gives more reliable method.

4.1.1. Creating a Regression Tree

The input of a regression tree consist of two parts. First, the feature vector $\mathbf{x}_i$ gives characteristics of an object and is used for decision making within the tree. Second, the target variable y_i represents the the variable to predict. For a given data set $\{(\mathbf{x}_i, y_i) : i = 1, \ldots, n, \mathbf{x}_i \in \mathbb{R}^k\}$ a regression tree can be made. The features used to create a regression tree are generally continuous values. This results in binary splits of the form $x_{1,1} \leq 10$ and $x_{1,1} > 10$. Creating a regression tree is a top-down method as it starts at the root. At every branch the best performing split for that moment is determined, this is known as a greedy machine learning approach. For feature j given by $x_{i,j}$ is a split parameter s defined which splits the values of the features into two regions:

$$R_1(j, s) = \{\mathbf{x}_i | x_{i,j} < s\}$$
$$R_2(j, s) = \{\mathbf{x}_i | x_{i,j} \geq s\}$$

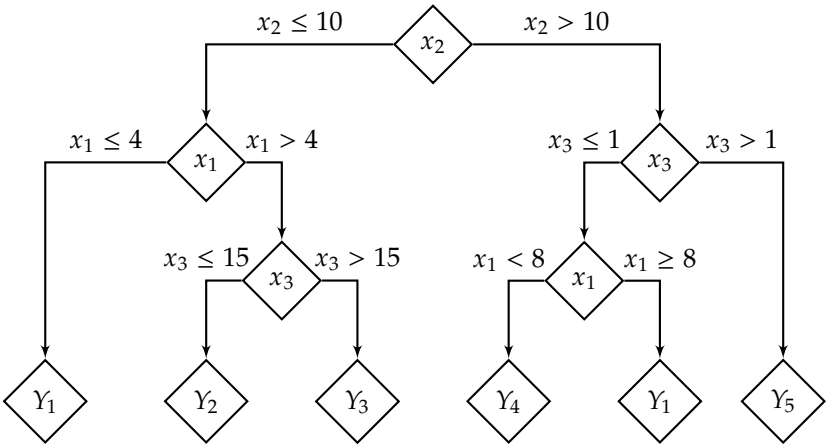


Figure 4.1: A regression tree with three features and multiple possible output values.

The mean target value of region R_1 is denoted by $\hat{y}_{R_1}$, likewise $\hat{y}_{R_2}$ corresponds to region R_2. The best performing split is, for example, determined by minimising the function of squared residuals given in Equation 4.1. Continuing this procedure by creating new regions based on the current regions results in a regression tree.

$$\sum_{j:\mathbf{x}_j \in R_1} (y_j - \hat{y}_{R_1})^2 + \sum_{j:\mathbf{x}_j \in R_2} (y_j - \hat{y}_{R_2})^2 \tag{4.1}$$

Completing this process for the complete set of features might result in a model too specific for the test data, which does not perform well on other data. This is known as overfitting and should be prevented. One possibility to overcome overfitting is stopping when the difference in residual sum of squares (RSS) at each split does not exceed some threshold. This is however not reliable as an extremely good split could have followed up on a bad split. This bad split might not exceed the threshold resulting in the extremely good split never happening. Another method is by building the larger tree (T_0) and then prune to a subtree (T). The performance on a test or validation set can be determined for all subtrees and the one resulting in the lowest errors is chosen. This is however computationally heavy for larger trees as there are a lot of subtrees to consider. Hence, a smaller set of subtrees to consider is needed.

One way to tackle this is by introducing a penalty as a function of the number of leaves ($|T|$). A sequence of indexed trees by penalty parameter α can be obtained. For every value of α there is a tree $T \subset T_0$ such that Equation 4.2 is minimal. This method is known as cost complexity pruning. All R_n's in Equation 4.2 correspond only to terminal nodes. Setting α equal to zero results in the original tree (T) while increasing α results in fewer leaves until only the root note is left.

$$\sum_{n=1}^{|T|} \sum_{i:x_i \in R_n} (y_i - y_{R_n})^2 + \alpha|T| \tag{4.2}$$

The value of α can be chosen by the user, there is however a more reliable method. To determine the optimal value of α, cross validation is used. The data is divided into K subsets or folds. Using $K-1$ folds a new regression tree is created on which cost complexity pruning is applied. The mean squared prediction error for the remaining fold can be determined and written as a function of α. Doing so K times, in which each fold is used as the individual fold once, and taking the mean results in the average mean squared prediction error for a α. The value of α which results in the minimal error is chosen as the value of α for the original tree. This α determines the final regression tree. The complete algorithm can be found in Algorithm 1. Setting α equal to zero, a complete tree is obtained with as much leaves as possible.

Algorithm 1 Creating a Regression Tree (source: (James et al., 2013))

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.
2. Apply cost complexity pruning to the large tree to obtain a sequence of best subtrees, as a function of α.
3. Use K-fold cross-validation to choose α. That is, divide the training observations into K folds. For each $k = 1, \ldots, K$:
   (a) Repeat steps 1 and 2 on all but the k-th fold of the training data.
   (b) Evaluate the mean squared prediction error on the data in the left-out k-th fold, as a function of α.

   Average the results for each value of α and pick α to minimise the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

4.2. Gradient Boosting

After building one regression tree model, there could still be residuals which can be used as input for a second regression tree model. Applying this repeatedly results in a function approximation by summing over of all the individual tree models. Boosting algorithms follow this principle and combine weak learners to a strong learner which better performance (Hastie et al., 2009). Gradient boosting optimises the overall error at every iteration. The error function used to determine the performance at every iteration is denoted as the loss function L. For a given data set $\{(\mathbf{x}_i, y_i) : i = 1, \ldots, N, \mathbf{x}_i \in \mathbb{R}^k\}$, gradient boosting aims to find $\hat{f}$ such that a given loss function $L(\hat{f})$ is minimised. It does so by relying on previous fitted functions in combination with the gradient of the loss function, following gradient descent techniques. At each iteration does the approximation move in the direction of the negative gradient. In this way a local minimum is reached. To ensure approximation of the global minimum, the loss function should be convex. The loss function has to be at least first order differentiable. Some examples of loss functions are given as below, as the sum of squared residuals and the mean squared error, respectively.

$$L(\hat{f}) = \sum_{i=1}^{N} (y_i - \hat{f}(\mathbf{x_i}))^2$$

$$L(\hat{f}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{f}(\mathbf{x_i}))^2$$

The loss function is however defined at object level as well. For example, loss function at object level of the above equations are given by

$$L(y_i, \hat{f}(\mathbf{x}_i)) = (y_i - \hat{f}(\mathbf{x_i}))^2$$

$$L(y_i \hat{f}(\mathbf{x}_i)) = \frac{y_i - \hat{f}(\mathbf{x_i}))^2}{N}.$$

To stay true to the literature (and their notation), function representations of regressions trees are used. Let h_m denote the output function of a regression tree with T_m leaves. The output space of tree h_m is partitioned into T_m disjoint regions $R_1 \ldots R_{T_m}$, which correspond to the leaves of the regression tree. The output function of a tree can be written as

$$h_m(\mathbf{x}_i) = \sum_{j=1}^{T_m} \hat{f}_m(\mathbf{x}_i) \mathbb{1}_{R_j}. \tag{4.3}$$

Gradient boosting creates an additive approximation $\hat{f}(\mathbf{x})$ as the sum of different functions of trees. Or generally, the next best approximation is the current tree added to the previous approximation. In The current tree is fitted on another data set than the previous regression trees, for example on the residuals of the previous approximations.

$$\hat{f}_m = \hat{f}_{m-1} + h_m \tag{4.4}$$

This approximation is an iterative process based on loss function $L(\hat{f})$. Like every iterative process, a starting point needs to be defined. In some cases is this starting point randomly chosen, for example by setting it equal to a constant function. To start with a better initial guess, it is possible to determine the starting point by minimizing the loss function over different regression trees.

$$\hat{f}_0 = \arg\min_h \sum_{i=1}^{n} L(y_i, h(\mathbf{x}_i)) \tag{4.5}$$

Again, one tries to minimise the loss function with respect to the current regression tree. The so far best approximation is known and is used to determine the regression tree minimizing the loss function given at a second stage by solving Equation 4.6.

$$h_m = \arg\min_h \sum_{i=1}^{n} L(y_i, \hat{f}_{m-1}(\mathbf{x}_i) + h(\mathbf{x}_i)) \tag{4.6}$$

Minimising the loss function over all possible regression trees can be a difficult activity. To ease the minimisation one can approximate the loss function with the first order Taylor expansion of the loss function. The Taylor expansion is taken around point $\hat{f}_{m-1}(\mathbf{x}_i)$.

$$L(\hat{f}_m) = L(\hat{f}_{m-1} + h_m) \approx \sum_{i=1}^{N} \left(L(y_i, \hat{f}_{m-1}(\mathbf{x}_i)) + \left[\frac{\partial L(y_i, \hat{f}(\mathbf{x}_i)}{\partial \hat{f}(\mathbf{x}_i)} \right]_{\hat{f}(\mathbf{x}_i)=\hat{f}_{m-1}(\mathbf{x}_i)} h_m(\mathbf{x}_i) \right) \tag{4.7}$$

Minimizing the Taylor expansion with respect to h_m one observes that the best regression tree is given by the negative gradient for each object. This regression tree is multiplied by the learning rate parameter, η. This parameter defines the step size in the direction of the negative gradient. The negative gradient for one feature vector is given in Equation 4.8.

$$h_m(\mathbf{x}_i) = -\eta \left[\frac{\partial L(y_i, \hat{f}(\mathbf{x}_i)}{\partial \hat{f}(\mathbf{x}_i)} \right]_{\hat{f}(\mathbf{x}_i)=\hat{f}_{m-1}(\mathbf{x}_i)} \tag{4.8}$$

The value of η should be chosen carefully. A small value results in a lot of steps to reach the minimum. While a too large value results in large updates which in turn could result in divergent behaviour and the minimum value is never attained. Taking $h_m(\mathbf{x}_i)$ as given in Equation 4.8 gives indeed an decreasing value for the loss function. For every iteration, one observes that the value of the loss function is smaller than in the previous iteration.

$$L(y_i, \hat{f}_m(\mathbf{x}_i)) = L(y_i, \hat{f}_{m-1}(\mathbf{x}_i)) - \eta \left(\left[\frac{\partial L(y_i, \hat{f}(\mathbf{x}_i)}{\partial \hat{f}(\mathbf{x}_i)} \right]_{\hat{f}(\mathbf{x}_i)=\hat{f}_{m-1}(\mathbf{x}_i)} \right)^2 < L(y_i, \hat{f}_{m-1}(\mathbf{x}_i)) \tag{4.9}$$

Every h_m can be seen as a step of the optimisation of approximation $f(\mathbf{x})$. For every object in the training data, the so-called pseudo-residuals given by Equation 4.8 are determined. This results in a new data set $\{(\mathbf{x}_i, r_{m,i}), i = 1, \ldots, n\}$. A weak learner can be applied to this new data set, resulting in a new regression tree. Again, the pseudo-residuals can be determined which implies a new data set. Repeating this process M times results in a final approximation $\hat{f}_M$. The complete

algorithm can be found in Algorithm 2. The pseudo-residuals differ from the standard residuals and are dependent on the loss function. By taking a standard loss function such as the squared error

$$L(\hat{f}_m) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{f}_m(\mathbf{x}_i))^2$$

it becomes clear that the residuals and pseudo-residuals are equal in specific cases.

Algorithm 2 Gradient Boosting for Regression Trees (source: (Bentéjac et al., 2021)

1. Set $\hat{f}_0(\mathbf{x}) = \arg\min_h \sum_{i=1}^{n} L(y_i, h)$
2. For $m = 1, \ldots, M$:
   (a) Compute the pseudo-residuals $r_{m,i} = -\left[\frac{\partial L(y_i, \hat{f}(\mathbf{x}_i))}{\partial \hat{f}(\mathbf{x}_i)}\right]_{\hat{f}(\mathbf{x}_i)=\hat{f}_{m-1}(\mathbf{x}_i)}$
   (b) Fit a regression tree on new data set $\{(\mathbf{x}_i, r_{m,i}), i = 1, \ldots, n\}$
   (c) Determine $h_m = \arg\min_h \sum_{i=1}^{n} L(y_i, \hat{f}_{m-1}(\mathbf{x}_i) + h)$
   (d) Update the function $\hat{f}_m = \hat{f}_{m-1} + h_m$
3. Return approximation $\hat{f}_M$

4.3. Extreme Gradient Boosting

Extreme gradient boosting (XGBoost) is, like gradient boosting, a strong learner based on weak learners (Chen & Guestrin, 2016). It is designed to perform efficient on large data sets. The final approximation is based on an addition of weak learners obtained by minimising a loss function. XGBoost works with the same data set as given for gradient boosting given by $\{(\mathbf{x}_i, y_i) : i = 1, \ldots, N, \mathbf{x}_i \in \mathbb{R}^k\}$. The loss function for extreme gradient boosting is similar to the loss function of gradient boosting in the following way at iteration m

$$L_{xgb}(\hat{f}_m) = \sum_{i=1}^{N} L(y_i, \hat{f}_m(\mathbf{x}_i)) \tag{4.10}$$

$\hat{f}_m(\mathbf{x}_i)$ represent the prediction of object i at iteration m. The loss function given in Equation 4.10 has the observed values as well as the predictions as parameters. The training is an iterative process, in which the previous steps are fixed, and a new regression tree is added to improve the performance of the model. One can rewrite the loss function at iteration m to Equation 4.11 in which the fixed data is combined into one constant.

$$L_{xgb}(\hat{f}_m) = \sum_{i=1}^{N} \Big(L(y_i, \hat{f}_{m-1}(\mathbf{x}_i) + h_m(\mathbf{x}_i))\Big) \tag{4.11}$$

Just like standard gradient boosting, XGBoost uses a Taylor expansion of the loss function around $\hat{f}_m(\mathbf{x})$. XGBoost does consider a second order Taylor polynomial to eventually approximate the minimum.

$$L_{xgb}(\hat{f}_m) \approx \sum_{i=1}^{N} \left(L(y_i, \hat{f}_{m-1}(\mathbf{x}_i)) + G_m(\mathbf{x}_i) \cdot h_m(\mathbf{x}_i) + \frac{1}{2} H_m(\mathbf{x}_i) \cdot h_m^2(\mathbf{x}_i)\right) \tag{4.12}$$

The first en second derivatives are scalars as given in Equations 4.13 and 4.14, respectively. In XGBoost are all first derivatives of the objects combined into one vector, similarly all second

derivatives are combined into one vector.

$$G_m(\mathbf{x}_i) = \left[\frac{\partial L(y_i, \hat{f}(\mathbf{x}_i))}{\partial \hat{f}(\mathbf{x}_i)} \right]_{\hat{f}(\mathbf{x}_i)=\hat{f}_{m-1}(\mathbf{x}_i)} \tag{4.13}$$

$$H_m(\mathbf{x}_i) = \left[\frac{\partial^2 L(y_i, \hat{f}(\mathbf{x}_i))}{\partial \hat{f}(\mathbf{x}_i)^2} \right]_{\hat{f}(\mathbf{x}_i)=\hat{f}_{m-1}(\mathbf{x}_i)} \tag{4.14}$$

Since the constants do not influence the argument corresponding the minimum, one can remove all constants. The loss function after removing all constants results in iteration m is considered.

$$L_{xgb}(\hat{f}_m) = \sum_{i=1}^{N} \left(G_m(\mathbf{x}_i) \cdot h_m(\mathbf{x}_i) + \frac{1}{2} H_m(\mathbf{x}_i) \cdot h_m^2(\mathbf{x}_i) \right) \tag{4.15}$$

Recall the definition of tree structure $h_m(\mathbf{x_i})$ as explained in Section 4.2. Every object has a prediction corresponding to exactly one leaf and multiple objects can correspond to the same leaf. The function defining this behaviour is given by $q(\mathbf{x}_i)$. Every leaf has one value (or prediction), this value is given by in vector w. Combining $q(\mathbf{x}_i)$ and w, one can write (XGBoost, 2022)

$$h_m(\mathbf{x_i}) = w_{q(\mathbf{x}_i)}, w \in \mathbb{R}^M, q : \mathbb{R}^d \rightarrow 1, 2, \ldots, T_m. \tag{4.16}$$

Using this observation, one can rewrite the loss function even further by removing all constants. XGBoost introduces a regularisation term as given in Equation 4.17. This regularisation term implies a penalty for larger tree with lots of leaves.

$$\Omega(h_m) = \gamma T_m + \frac{1}{2} \lambda \sum_{j=1}^{T_m} w_j^2 \tag{4.17}$$

In which γ gives the minimum loss required to create a new split. Higher values of γ results in simpler trees with less leaves. λ gives the L2 regularisation on the scores. Higher values of λ result again in simpler trees. By defining sets $I_j = \{i | q(\mathbf{x}_i = j\}$ as the set of instances assigned to leaf j, the loss function becomes

$$L_{xgb}(\hat{f}_m) = \sum_{i=1}^{N} \left(G_m(\mathbf{x}_i) \cdot w_{q(\mathbf{x}_i)} + \frac{1}{2} H_m(\mathbf{x}_i) \cdot w_{q(\mathbf{x}_i)}^2 \right) + \gamma T_m + \frac{1}{2} \lambda \sum_{j=1}^{T_m} w_j^2 \tag{4.18}$$

$$= \sum_{j=1}^{T_m} \left[\left(\sum_{i \in I_j} G_m(\mathbf{x}_i) \right) \cdot w_j + \frac{1}{2} \left(\sum_{i \in I_j} H_m(\mathbf{x}_i) + \lambda \right) w_j^2 \right] + \gamma T_m \tag{4.19}$$

$$\tag{4.20}$$

Optimising the final equation, one obtains

$$\hat{w}_j = -\frac{\sum_{i \in I_j} G_m(\mathbf{x}_i)}{\sum_{i \in I_j} H_m(\mathbf{x}_i) + \lambda} \tag{4.21}$$

$$\hat{L}_{xgb}(\hat{f}_m) = -\frac{1}{2} \sum_{j=1}^{T_m} \frac{\left(\sum_{i \in I_l} G_m(\mathbf{x}_i) \right)^2}{\sum_{i \in I_j} H_m(\mathbf{x}_i) + \lambda} + \gamma T_m \tag{4.22}$$

The last equation is used as a scoring function to measure how good tree structure h_m is at iteration m. Based on this scoring function, one can determine the best tree, it is, however, computationally heavy to measure all possible tree structures. An alternative method is introduced as a greedy algorithm which starts from a root and add branches the tree which have most gain (Chen &

Guestrin, 2016). Let I_l denoted the instance going to the left at a split and I_r the instances going to the right at the same split. The union of the both sets is given as $I = I_l \cup I_r$. These settings can be used to determine the gain in performance for each split. Applying Equation 4.22 on I_l, one obtains a score for the left leaf of a split as given in Equation 4.23.

$$\frac{\left(\sum_{i\in I_l} G(\mathbf{x}_i)\right)^2}{\sum_{i\in I_l} H(\mathbf{x}_i) + \lambda} \tag{4.23}$$

Similar scoring functions can be made for the right leaf as well as the only node to which both the leaves are connected to. This way, a gain per split is defined. As a split results in one more leaf an the loss function is dependent on the number of leaves, the total gain in performance is obtained by subtracting complexity parameter γ. Note, this is a variant of pruning which already happens as initialization time of the tree. The complete gain of a split can be combined into one complete scoring gain function as given in Equation 4.24.

This scoring function is applied for every split and the best performing one is chosen, resulting in a regression tree. In the perfect world, one would consider every split at every branching step of a tree, resulting in the best tree possible.

$$\text{Gain} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_l} G(\mathbf{x}_i)\right)^2}{\sum_{i\in I_l} H(\mathbf{x}_i) + \lambda} + \frac{\left(\sum_{i\in I_r} G(\mathbf{x}_i)\right)^2}{\sum_{i\in I_r} H(\mathbf{x}_i) + \lambda} - \frac{\left(\sum_{i\in I} G(\mathbf{x}_i)\right)^2}{\sum_{i\in I} H(\mathbf{x}_i) + \lambda}\right] - \gamma \tag{4.24}$$

In a next iteration, one defines a new data set $\left\{\left(\mathbf{x}_i, -\frac{\sum_{i\in I_j} G_m(\mathbf{x}_i)}{\sum_{i\in I_j} H_m(\mathbf{x}_i)+\lambda}\right) : i = 1, \ldots, N\right\}$ and the process is repeated. On top of this, a learning rate η is chosen. The addition of the weak learners is given as the sum of regression trees multiplied by the learning rate, η.

$$\hat{f}_m = \hat{f}_{m-1} + \eta h_m \tag{4.25}$$

The complete process can be found in Algorithm 3.

4.3.1. Split Finding and Parallelisation

Determining the best possible split is, depending on the data, a computationally heavy procedure. XGBoost has the possibility to approximate the best split by creating a histogram of the feature values. Instead of calculating with all the continuous values of the features, XGBoost creates intervals, the bins of the histogram. Split performance is only determined for the bins and not the individual observations. On top of this, there is the possibility to create a new histogram in every iteration or just create only one histogram at the start of the training.

Taking every individual observation into account, results in the best estimates. This takes however a lot of time on larger data sets. The histogram approximation given by XGBoost implies a significant decrease in computation time as a lot of steps can be neglected. Creating a histogram at every split requires more time than using a predefined histogram. One should however consider the loss in performance by using an approximation method.

Moreover, XGBoost has a special implementation which enables a better processing efficiency. 'Normal' machine learning methods relies on a small set of calculators, denoted as cores, able to make challenging equations. The cores are all delegated by the Central Processing Unit (CPU) of a device. A standard device has between eight to thirty-two cores. The CPU can handle a number of tasks equal to the number of cores at the same time. The remaining tasks are queued up and will be awaiting execution. XGBoost has the possibility to use the Graphics Processing Unit (GPU) of the device. The GPU consist of hundred to thousands of cores and can make way more calculations simultaneously. It does, however, have some limits. The calculations done on cores of the GPU are required to be independent and of a simple nature (e.g. the trivial mathematics operations). Determining the scoring for a split is an equation capable to be completed on a GPU core. This fastens the process significantly, while not loosing any accuracy. As of now, the GPU support is only available for the histogram methods.

Algorithm 3 Extreme Gradient Boosting of Regression Trees (Based on: (Chen & Guestrin, 2016))

1. Set $\hat{f}_0(\mathbf{x}) = \arg\min_{\beta} \sum_{i=1}^{n} L(y_i, \beta)$
2. For $m = 1, \ldots, M$:
   (a) Compute first and second order derivatives:
   $$G_m(\mathbf{x}_i) = \left[\frac{\partial L(y_i, \hat{f}(\mathbf{x}_i))}{\partial \hat{f}(\mathbf{x}_i)} \right]_{\hat{f}(\mathbf{x}_i)=\hat{f}_{m-1}(\mathbf{x}_i)}$$
   $$H_m(\mathbf{x}_i) = \left[\frac{\partial^2 L(y_i, \hat{f}(\mathbf{x}_i))}{\partial \hat{f}(\mathbf{x}_i)^2} \right]_{\hat{f}(\mathbf{x}_i)=\hat{f}_{m-1}(\mathbf{x}_i)}$$
   (b) Fit a regression tree h_m on new data set $\left\{ \left(\mathbf{x}_i, -\frac{\sum_{i \in I_j} G_m(\mathbf{x}_i)}{\sum_{i \in I_j} H_m(\mathbf{x}_i) + \lambda} \right) : i = 1, \ldots, n \right\}$
      i. Start with a root node
      ii. For $j = 1, \ldots$ Number of leaves:
         A. Determine best performing split:
         $$\text{Gain} = \frac{1}{2} \left[\frac{\left(\sum_{i \in I_l} G(\mathbf{x}_i) \right)^2}{\sum_{i \in I_l} H(\mathbf{x}_i) + \lambda} + \frac{\left(\sum_{i \in I_r} G(\mathbf{x}_i) \right)^2}{\sum_{i \in I_r} H(\mathbf{x}_i) + \lambda} - \frac{(\sum_{i \in I} G(\mathbf{x}_i))^2}{\sum_{i \in I} H(\mathbf{x}_i) + \lambda} \right] - \gamma$$
         B. Add split to regression tree
   (c) Update the function $\hat{f}_m = \hat{f}_{m-1} + \eta h_m$
3. Return approximation $\hat{f}_M$

4.4. Multi-Target Extreme Gradient Boosting

Standard XGBoost only predicts one target variable, while this research focuses on multiple targets. There are two methods which incorporate multiple target variables. The first method is by fitting an individual model on every target, this does however not take dependency between the target variables into account. The second method is using a combined model which trains on all target variables at the same time. In this case one tries to approximate a multi variable function $f : \mathbb{R}^v \to \mathbb{R}^u$. The leaves of the regression trees used to train this model represent vectors of length u instead of scalars.

Similarly, the loss function is required to have a different structure as well. There are two general ideas on the potential loss function. As a first method, one can define the loss function as $L : \mathbb{R}^u \to \mathbb{R}^u$ in which every function has their own individual loss function. As a second method, one can take a loss function with output in one dimension, $L : \mathbb{R}^u \to \mathbb{R}$. To determine behaviour of both methods at the same time, the general loss function is used, $L : \mathbb{R}^u \to \mathbb{R}^u$.

In standard XGBoost, one uses Taylor expansion the approximate the correct leaf scores used to define the data set for the next regression tree. As this problem works with a multi variable function, there is no simple Taylor expansion. Before extending the Taylor expansion for functions from one Euclidean space to another, some notation and new subjects are introduced. To start, every scalar, vector, matrix and any product of higher order can be described using directions. A scalar is just a number and requires zero other information to be described besides its value. A vector on the other hand requires multiple values, say u values are needed. Similarly a matrix in $\mathbb{R}^{u \times u}$ requires u^2 values or u vectors of length u. One can extend this to a object in $\mathbb{R}^{u \times u \times u}$, which can be represented as a vector of matrices. One can extend this indefinitely and describe higher dimensional objects. The objects obtained this way are known as tensors as given in Definition 2. The rank of a tensor is defined as the number of vectors (or arrays) required to describe the object. Hence scalars, vectors and matrices are tensors of rank zero, one and two, respectively. There exists different definitions of tensors, depending on the background of the subject. In machine learning one has the definition as given in Definition 2.

Definition 2 (Tensor (Guo, 2021))
A tensor is a multidimensional array (or matrix).
The rank of a tensor is given by the number of indices required to get one value in the tensor.

Based on the definition of a tensor, a multiplication between two tensors can be defined. This multiplication is known as the tensor product ($\otimes$) and is given as in Definition 3. This is an abstract definition, a more direct example for a vector and a matrix is given in Equation 4.26.

Definition 3 (Tensor product (Guo, 2021))
Let U, V and W be vector spaces with

$$dim(W) = dim(U) \cdot dim(V)$$

And let $\otimes : U \times V \rightarrow W$ be a mapping. This mapping is known as the tensor product.

$$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \otimes \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} = \begin{pmatrix} a_1 \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} \\ a_2 \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} a_1 b_{1,1} & a_1 b_{1,2} \\ a_1 b_{2,1} & a_1 b_{2,2} \\ a_2 b_{1,1} & a_2 b_{1,2} \\ a_2 b_{2,1} & a_2 b_{2,2} \end{pmatrix} \tag{4.26}$$

With the above definitions, the Taylor expansion can be extended to be applicable for multi variable functions. For a given loss function can the Taylor expansion around vector $\mathbf{x}_0$ be written to Equation 4.27 in which $D^k L$ represents differentiation of order k on function L and $R(\mathbf{x_0})$ the remainder.

$$f(\mathbf{x}) = f(\mathbf{x}_0) + (Df(\mathbf{x}_0))(\mathbf{x} - \mathbf{x}_0) + \cdots + \frac{D^k f(\mathbf{x}_0)}{k!}(\mathbf{x} - \mathbf{x}_0)^{\otimes k} + R(\mathbf{x_0}) \tag{4.27}$$

Only considering the first k terms of the Taylor polynomial and neglecting the remainder, one obtains the so-called k-jet as given in Definition 4. As with standard XGBoost, only the terms up to the second order are used, hence a 2-jet is used in the remainder of this research.

Definition 4 (Jet)
The 2-jet of a loss function L is given as the first 2 terms of the Taylor polynomial:

$$L(\hat{\mathbf{f}}_{m-1}) + \mathbf{h}_m \cdot DL(\hat{\mathbf{f}}_{m-1}) + \frac{1}{2}\mathbf{h}_m^{\otimes 2} \cdot D^2 L(\hat{\mathbf{f}}_{m-1})$$

One still has the general additive approach of a gradient boosting, given by $\hat{\mathbf{f}}_m = \mathbf{f}_{m-1} + \mathbf{h}_m$ in which $\mathbf{h}_m$ is the tree with vector values on the leaves. From now on is $\hat{\mathbf{f}}_m(\mathbf{x}_i)$ written as $\hat{\mathbf{y}}_m^i$ to improve readability. Similarly $(\hat{\mathbf{y}}_m^i)_j$ is used to indicate the j-th element of vector $\hat{\mathbf{y}}_m^i$. The loss function can be interpreted as multiple individual functions combined into one vector. The individual functions are given as $L_1 \dots L_u$. The first derivative of one instance is simply the Jacobian matrix and is given as in Equation 4.28. Taking the Jacobian matrix for all n instances, one obtains a tensor

of rank 3. One can visualize this as a vector with matrix elements. $DL(\hat{\mathbf{y}})$ is a matrix whose rows can be seen as the gradients of the components of vector function $L(\hat{\mathbf{y}})$.

$$DL(\hat{\mathbf{y}}) = \begin{pmatrix} \frac{\partial L_1}{\partial(\hat{\mathbf{y}})_1}(\hat{\mathbf{y}}) & \cdots & \frac{\partial L_1}{\partial(\hat{\mathbf{y}})_u}(\hat{\mathbf{y}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial L_u}{\partial(\hat{\mathbf{y}})_1}(\hat{\mathbf{y}}) & \cdots & \frac{\partial L_u}{\partial(\hat{\mathbf{y}})_u}(\hat{\mathbf{y}}) \end{pmatrix} \tag{4.28}$$

The second derivative is more challenging as $D^2L(\mathbf{y}, \hat{\mathbf{y}}) : \mathbb{R}^u \to (\mathbb{R}^{\mathbf{u}}, (\mathbb{R}^{\mathbf{u}}, \mathbb{R}^{\mathbf{u}}))$ is a function whose codomain consists of vectors of matrices. The second order derivative for only one instance of the data set is given as tensor or rank 3 as given in Equation 4.29. By combining all N data points, the problem becomes even more difficult and turns into a 4-tensor. This 4-tensor can be visualized as a matrix in which the elements itself are given by matrices. This significantly complicated the problem.

$$D^2L(\hat{\mathbf{y}}) = \left(\begin{pmatrix} \frac{\partial^2 L_1}{\partial(\hat{\mathbf{y}})_1^2}(\hat{\mathbf{y}}) & \cdots & \frac{\partial^2 L_1}{\partial(\hat{\mathbf{y}})_1 \partial(\hat{\mathbf{y}})_u}(\hat{\mathbf{y}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 L_1}{\partial(\hat{\mathbf{y}})_u \partial(\hat{\mathbf{y}})_1}(\hat{\mathbf{y}}) & \cdots & \frac{\partial^2 L_1}{\partial(\hat{\mathbf{y}})_u^2}(\hat{\mathbf{y}}) \end{pmatrix}, \ldots, \begin{pmatrix} \frac{\partial^2 L_u}{\partial(\hat{\mathbf{y}})_1^2}(\hat{\mathbf{y}}) & \cdots & \frac{\partial^2 L_u}{\partial(\hat{\mathbf{y}})_1 \partial(\hat{\mathbf{y}})_u}(\hat{\mathbf{y}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 L_u}{\partial(\hat{\mathbf{y}})_u \partial(\hat{\mathbf{y}})_1}(\hat{\mathbf{y}}) & \cdots & \frac{\partial^2 L_u}{\partial(\hat{\mathbf{y}})_u^2}(\hat{\mathbf{y}}) \end{pmatrix} \right) \tag{4.29}$$

Determining this tensor is computationally heavy and should be prevented. Hence, using a multi output loss function is not desirable. A single output loss function, however, simplifies the problem significantly. The gradient of one instance is no longer a matrix, but a vector as shown in Equation 4.30. Similarly, the second derivative of the loss function of one instance forms a matrix instead of a 3-tensor. The second order derivative, $D^2L(\hat{\mathbf{y}})$, is given as matrix $\mathbf{H}$ in Equation 4.31. By considering all N point in the data set, the first order derivative transforms into a matrix, while the second order derivative transforms into a tensor or rank 3. At this moment is the focus only on one feature vector as well as one target vector of the data set, later the complete data set is discussed.

$$\mathbf{G}(\hat{\mathbf{y}}) = \left(\frac{\partial L}{\partial(\hat{\mathbf{y}})_1}(\hat{\mathbf{y}}), \ldots, \frac{\partial L}{\partial(\hat{\mathbf{y}})_u}(\hat{\mathbf{y}}) \right) \tag{4.30}$$

$$\mathbf{H}(\hat{\mathbf{y}}) = \begin{pmatrix} \frac{\partial^2 L}{\partial(\hat{\mathbf{y}})_1^2}(\hat{\mathbf{y}}) & \cdots & \frac{\partial^2 L}{\partial(\hat{\mathbf{y}})_1 \partial(\hat{\mathbf{y}})_u}(\hat{\mathbf{y}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial(\hat{\mathbf{y}})_u \partial(\hat{\mathbf{y}})_1}(\hat{\mathbf{y}}) & \cdots & \frac{\partial^2 L}{\partial(\hat{\mathbf{y}})_u^2}(\hat{\mathbf{y}}) \end{pmatrix} \tag{4.31}$$

For readability, some notation is introduced. $\mathbf{G}_m^i$ corresponds to the gradient vector with respect to approximation $\hat{\mathbf{y}}_m^i$, given as $\mathbf{G}_m^i = \mathbf{G}(\hat{\mathbf{y}}_m^i)$. Similarly, $\mathbf{H}_m^i$ corresponds to the matrix given in Equation 4.31 as $\mathbf{H}(\hat{\mathbf{y}}_m^i)$. At the same time is a new tree structure defined based on Equation 4.16:

$$\mathbf{h_m}(\mathbf{x}_i) = \mathbf{w}_q(\mathbf{x}_i), q : \mathbb{R}^M \to 1, 2, \ldots, T_m, \mathbf{w} \in \mathbb{R}^{T_m \times u} \tag{4.32}$$

There is still regularization required for this multi output model. The same regularization term as in standard XGBoost can be used, since the complexity of the model is still dependent on the number of leaves. By introduction this regularization term, one can write an approximation of the loss function for the entire data set as in Equation 4.33.

$$L(\hat{\mathbf{y}}_m) = \sum_{i=1}^{N} \left(L(\mathbf{y}, \hat{\mathbf{y}}_{m-1}^i) + \mathbf{G}_m^i \mathbf{w} + \frac{1}{2} \mathbf{w}^T \mathbf{H}_m^i \mathbf{w} \right) + \gamma T_m + \frac{1}{2} \lambda ||\mathbf{w}||_2^2 \tag{4.33}$$

This function can in turn be optimized to determine the global minimum. The loss function is differentiated with respect to $\mathbf{w}$ and set equal to zero, resulting in an optimal leaf and corresponding

loss function value as given in Equations 4.34 and 4.35, respectively.

$$\hat{\mathbf{w}}_m = -\left(\sum_i \mathbf{H}_m^i + \lambda I\right)^{-1} \left(\sum_i \mathbf{G}_m^i\right) \tag{4.34}$$

$$\hat{L}(\hat{\mathbf{y}}_m) = -\frac{1}{2}\left(\sum_i \mathbf{G}_m^i\right)^T \cdot \left(\sum_i \mathbf{H}_m^i + \lambda I\right)^{-1} \cdot \left(\sum_i \mathbf{G}_m^i\right) + \gamma T_m \tag{4.35}$$

A similar split finding algorithm can be created shown in 4.24. This results in a cumbersome algorithm as for every split, the inverse of a $u \times u$ matrix needs to be determined. This problem solves itself when matrix $\mathbf{H}(\hat{\mathbf{y}})$ is a diagonal matrix, let $\hat{\mathbf{H}}(\hat{\mathbf{y}})$ be the diagonal of the matrix. In this case one can write the i-th element of leaf $\hat{\mathbf{w}}_m$.

$$\hat{\mathbf{w}}_m^i = -\frac{\sum_j (\mathbf{G}_m^i)_j}{\sum_j (\hat{\mathbf{H}}_m^i)_j + \lambda} \tag{4.36}$$

The final loss function can be found by summing over all dimensions of the leaves.

$$\hat{L}(\hat{\mathbf{y}}_m) = -\frac{1}{2}\sum_{i=1}^{u}\left(\frac{\sum_j \left((\mathbf{G}_m^i)_j\right)^2}{\sum_j \left((\hat{\mathbf{H}}_m^i)_j\right) + \lambda}\right) + \gamma T_m \tag{4.37}$$

Hence, the functions obtained for the multi-dimensional with diagonal matrix behaves similar to standard XGBoost. This does however not hold for the general case, as given in Equations 4.34 and 4.35. Chen et al. show that a diagonal approximation can be used as an upper bound for the first and order derivatives (Chen et al., 2015). This simplifies the problem as for all matrices only the diagonal elements needs to be considered, which results in Equation 4.36. As stated, by taking the complete data set, is the second order derivative on the complete data set given as an tensor of rank 3. This tensor can however be simplified to a matrix by taking the diagonal elements of the matrices as the rows of a new matrix. As for the implementation, one can find how this matrix is defined in an XGBoost application A.3. The first order derivative is left as a matrix in order to incorporate all information of this derivative.

The system is defined and only creation of the tree is still required. The procedure is similar to the standard XGBoost tree creation with the main difference in the vector leaves. An optimal split is searched in every tree branching step. This is, like the standard cases, done by defining scoring for every leaf obtained afters branching. Let I_l denote the instances going to the left at the split, I_r the instances going tot the right and I the union of both. I is the only node to which both the leaves are connected to. Using Equation 4.37, one can describe the scoring of the left instances as in Equation 4.38.

$$\sum_{i=1}^{u}\left(\frac{\sum_{j\in I_l} \left((\mathbf{G}_m^i)_j\right)^2}{\sum_{j\in I_l} \left((\hat{\mathbf{H}}_m^i)_j\right) + \lambda}\right) \tag{4.38}$$

The gain of a split can be defined as the sum of the scores on I_r and I_r, subtracted by I and complexity parameter y. This results in the gain function as given in Equation 4.39. This a more complicated function than in standard XGBoost. It is however still easy enough to be done in parallel on a GPU, resulting in a faster calculation than only working with the CPU. Besides, the histogram methods, as described in Section 4.3, can be applied in this situation as well, resulting in less splits to consider. This in turn results in less computation time. Combining both the histogram

method and the GPU usage a significant decrease in computation time is possible.

$$\text{Gain} = \frac{1}{2}\left[\sum_{i=1}^{u}\left(\frac{\sum_{j\in I_l}\left((\mathbf{G}_m^i)_j\right)^2}{\sum_{j\in I_l}\left((\hat{\mathbf{H}}_m^i)_j\right)+\lambda}\right)+\sum_{i=1}^{u}\left(\frac{\sum_{j\in I_r}\left((\mathbf{G}_m^i)_j\right)^2}{\sum_{j\in I_r}\left((\hat{\mathbf{H}}_m^i)_j\right)+\lambda}\right)-\sum_{i=1}^{u}\left(\frac{\sum_{j\in I}\left((\mathbf{G}_m^i)_j\right)^2}{\sum_{j\in I}\left((\hat{\mathbf{H}}_m^i)_j\right)+\lambda}\right)\right]-\gamma \tag{4.39}$$

The algorithm is similar to the standard case with some alterations, the complete algorithm can be found in Algorithm 4.

Algorithm 4 Extreme Gradient Boosting of Multi Output Regression Trees

1. Set $\hat{f}_0(\mathbf{x}) = \arg\min_{\beta} \sum_{i=1}^{n} L(y_i, \beta)$
2. For $m = 1, \ldots, M$:
   (a) Compute Gradients and Hessian's for every instance of the data set:

$$\mathbf{G}(\hat{\mathbf{y}}) = \left(\frac{\partial L}{\partial(\hat{\mathbf{y}})_1}(\hat{\mathbf{y}}), \ldots, \frac{\partial L}{\partial(\hat{\mathbf{y}})_u}(\hat{\mathbf{y}})\right)$$

$$\mathbf{H}(\hat{\mathbf{y}}) = \begin{pmatrix} \frac{\partial^2 L}{\partial(\hat{\mathbf{y}})_1^2}(\hat{\mathbf{y}}) & \cdots & \frac{\partial^2 L}{\partial(\hat{\mathbf{y}})_1 \partial(\hat{\mathbf{y}})_u}(\hat{\mathbf{y}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial(\hat{\mathbf{y}})_u \partial(\hat{\mathbf{y}})_1}(\hat{\mathbf{y}}) & \cdots & \frac{\partial^2 L}{\partial(\hat{\mathbf{y}})_u^2}(\hat{\mathbf{y}}) \end{pmatrix}$$

   (b) Fit a regression tree h_m on new data set $\left\{\left(\mathbf{x}_i, -\frac{\sum_j (\mathbf{G}_m^i)_j}{\sum_j (\hat{\mathbf{H}}_m^i)_j + \lambda}\right) : i = 1, \ldots, n\right\}$
      i. Start with a root node
      ii. For $j = 1, \ldots$ Number of leaves:
         A. Determine best performing split:

$$\text{Gain} = \tfrac{1}{2}\left[\sum_{i=1}^{u}\left(\frac{\sum_{j\in I_l}\left((\mathbf{G}_m^i)_j\right)^2}{\sum_{j\in I_l}\left((\hat{\mathbf{H}}_m^i)_j\right)+\lambda}\right)+\sum_{i=1}^{u}\left(\frac{\sum_{j\in I_r}\left((\mathbf{G}_m^i)_j\right)^2}{\sum_{j\in I_r}\left((\hat{\mathbf{H}}_m^i)_j\right)+\lambda}\right)-\sum_{i=1}^{u}\left(\frac{\sum_{j\in I}\left((\mathbf{G}_m^i)_j\right)^2}{\sum_{j\in I}\left((\hat{\mathbf{H}}_m^i)_j\right)+\lambda}\right)\right]-\gamma$$

         B. Add split to regression tree
   (c) Update the function $\hat{f}_m = \hat{f}_{m-1} + \eta h_m$
3. Return approximation $\hat{f}_M$

4.5. Performance Metrics and Loss Function

In Chapter 1 requirements of the approximation are given. These requirements state that 99.5% of the predictions should deviate within at most 1% or 0.5% for management and reporting purposes, respectively. A metric to check this can be defined. This metric determines the relative error of a prediction and if the prediction deviates within the allowed interval, a value of zero is given. If the prediction deviates more than allowed, value one is given. The sum over all prediction is taken and the mean is taken, resulting in Equation 4.40 for management requirements. This metric gives an estimate of the probability of prediction deviating more than allowed, and is thus required to be smaller than 0.05. Hence, this metric should be minimized. This function is however not (twice) differentiable and thus cannot be used as the loss function in extreme gradient boosting. Moreover, even if this metric was differentiable, it would not be the wanted loss function. The training of an algorithm would stop when every prediction deviates within the allowed intervals. This does however not mean that the best predictions are obtained. In the worst situation every prediction could deviate exactly below 1% while better predictions are within reach. This metric is

still used to determine whether the requirements are satisfied.

$$\frac{1}{N}\sum_{i}^{N}\mathbb{1}_{\frac{|y_i-\hat{y}_i|}{y_i}>1} \tag{4.40}$$

To get more insight in the performance of the machine learning method another metric needs to be defined. One of the most used metrics is the root mean squared error (RMSE). The RMSE, as given in Equation 4.41, gives a measure of the absolute error. The RMSE gets large for large differences between prediction $\hat{y}$ and y. This function is twice differentiable and forms a possible loss function for extreme gradient boosting. Opposite to Equation 4.40, the RMSE does aim to obtain the best approximations. Even if the approximations are close to their corresponding real values, the RMSE is unequal to zero and the training can continue to even further improve the model. This is however more susceptible for overfitting, as stricter predictions are required.

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2} \tag{4.41}$$

A more general loss function as given in Equation 4.42 is however more convenient. Both the first and second order derivatives of this loss function have a simple form. As well as a loss function which always tries to improve the results, similar to the RMSE. This loss function is used in Chapter 5 for an one dimensional extreme gradient boosting model.

$$\frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \tag{4.42}$$

4.5.1. Multi Output Regression

In this research are multiple target variable defined, which means that all metrics and the loss function need to be extended to be applicable to vectors. Equation 4.40 can be extended such that the highest difference between two corresponding vector elements is bigger than allowed. This way, all target variables of one prediction are required to satisfy the requirements on all variables at the same time. One obtains the metric as given in Equation 4.43 in which $||\mathbf{x}||_\infty$ represents the ∞-norm of vector $\mathbf{x}$ given as the largest absolute element of x and element wise division is applied.

$$\frac{1}{N}\sum_{i}^{N}\mathbb{1}_{\left\|\frac{\mathbf{y}_i-\hat{\mathbf{y}}_i}{\mathbf{y}_i}\right\|_\infty>1} \tag{4.43}$$

The RMSE can be extended to a vector function as well. One needs to subtract the observation from the approximation and take the 2-norm over the obtained vector. Taking the mean of all the results, one obtains the RMSE for a vector problem. This metric can be found in Equation 4.44

$$\frac{1}{N}\sum_{i=1}^{n}||\hat{\mathbf{y}}_i - \mathbf{y}_i||_2 \tag{4.44}$$

In a similar fashion one can extend the loss function to be capable of vector calculations. Again, taking the 2-norm over the difference results in the desired results as given in Equation 4.45. This results in convenient first and second order derivatives. This vector loss function is used in Chapter 5 for a multi target XGBoost model.

$$\frac{1}{2}\sum_{i=1}^{n}||\hat{\mathbf{y}}_i - \mathbf{y}_i||_2^2 \tag{4.45}$$

5

Results

With a calibrated Lee-Carter model, one can simulate future mortality tables and in turn simulate the future liability cash flow. This cash flow is constructed out of roughly two parts, the best estimate liability (BEL) and the solvency capital requirements (SCRs). The SCRs are again divided into different parts as discussed in Chapter 2. For this research, only three SCRs are considered, given as mortality, longevity and catastrophe. Based on these values in combination with the forecast mortality table from the Lee-Carter, input data for extreme gradient boosting (XGBoost) is obtained. Two different XGBoost approaches are trained on this data set, in order to determine the approach with the better performance. The first approach trains a model for every target variable individually. The second approach makes use of a vector target which trains the model for all target variables at once. Since the main time is spent in creating a training set, the size of such a training set is minimized.

5.1. Lee-Carter Forecasting

As described in Chapter 3, the Lee-Carter model is calibrated on historical data. Only the Dutch historical data is relevant for this research as the portfolio consist of products on the Dutch market. Historical mortality data is publicly available in the Human Mortality Database (HMD) (Human Mortality Database et al., 2022). HMD is a collaboration between various agencies that collect mortality data around the world. Not all data, given by this database, is correctly available. Higher age groups imply debatable values or no values at all, which is a consequence of a small sample size. For example, observations concerning ages above a hundred years old do not give a realistic view of the real mortality rates as the group size is relatively small. In most practises are these undefined rates set equal to one. In this research, this is applied for all ages 110 and above as these are absent in the HMD. The HMD has collected data of the Dutch population varying from ages 0 to 110 starting from year 1850 up till 2019. To incorporate the correct behaviour of the mortality rates, this complete data set is taken as input data.

5.1.1. Calibrating the Lee-Carter Model on the Dutch Population

Based on the structure of the data from the HMD, the program in Appendix A.1 is created, which imports and alters the data as described. There are already (outdated) packages created which apply Lee-Carter calibrations, the backlog information is however unclear or they apply some illogical assumptions. Because of this are these packages not used. A simple implementation of the Lee-Carter model is given in A.1. As given in Chapter 3, the Lee-Carter model is given by:

$$\ln(\mathbf{M}_{x,t}) = \mathbf{a}_x + \mathbf{b}_x \mathbf{k}_t + \epsilon_{x,t} \qquad x \in [0, 120] \text{ and } t \in [1850, 2019] \tag{5.1}$$

The individual components of Equation 5.1 are calibrated on the data. The derivation of the individual components is described in Chapter 3.

Mean Logarithmic Mortality Rates
First, vector **a** of Equation 5.1 is determined. As given in Equation 3.3, vector **a** is given as the average of the logarithmic mortality rates. The most natural deaths happen at higher ages, hence higher ages are expected to have a higher mortality rate than lower ages. Setting age against the mean logarithmic mortality rate, an increasing line is expected. This is exactly as observed in Figure 5.1 in which the every point corresponds the mean logarithmic mortality rate for that age. One can observe the decrease of the logarithmic mortality rates for ages until sixteen. After the age of sixteen an increase of the logarithmic values is observed, which is as expected. For ages above one hundred strange behaviour is observed. This is a direct consequence of the data at the HMD which is unreliable for higher ages. As one observes, male mortality rates are in general higher than female mortality rates. The values visualised in this figure form vector **a** of the Lee-Carter model calibration.

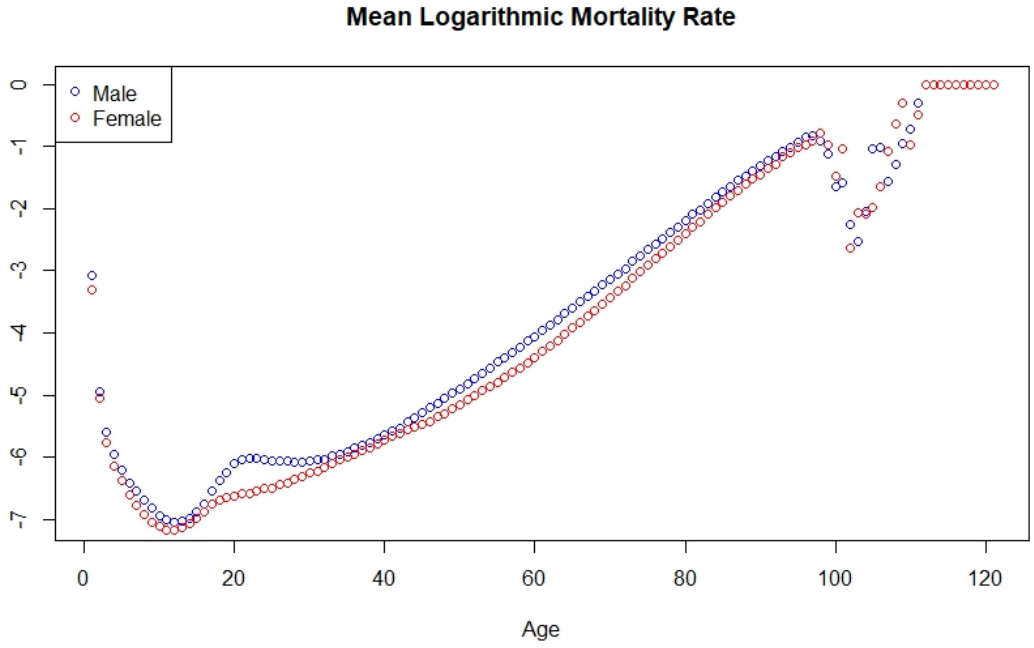


Figure 5.1: The mean logarithmic mortality rates based on the Dutch population. Every point corresponds to the average mortality rate over the years at that age. The inconsistency around age one hundred follows from insufficient data. A overall increasing series is observed which is a direct result of improvements in the medical world.

Relative Mortality
Second, vector **b** of Equation 5.1 is determined. As discussed in Chapter 3 is vector **b** determined as the result of using the singular value decomposition as a low rank approximation. It follows that **b** is part of an approximation of the original mortality matrix. This implies some uncertainties in the use of vector **b**. Elements of vector **b** describe the impulse of mortality at that age when the general level of mortality **k** changes, known as the relative mortality. Hence, it influences the mortality trend (**k**) according to whether change at a given age is faster or slower than the original mortality trend. For a large time horizon, it is expected that all elements are positive due to medical improvements. Similar as with **a** can the age be set against the value relative mortality of that age. The relative mortality for lower ages is expected to be higher than the relative mortality for higher ages. This is a result from the mortality rates changing less fiercely for higher ages. The obtained values of **b** are visualised in Figure 5.2, in which one indeed observes a decreasing series. Again, inconsistency is observed for ten ages starting at age one hundred. This is a consequence of the fiercely changing mortality rates at these ages given in the data from the HMD.

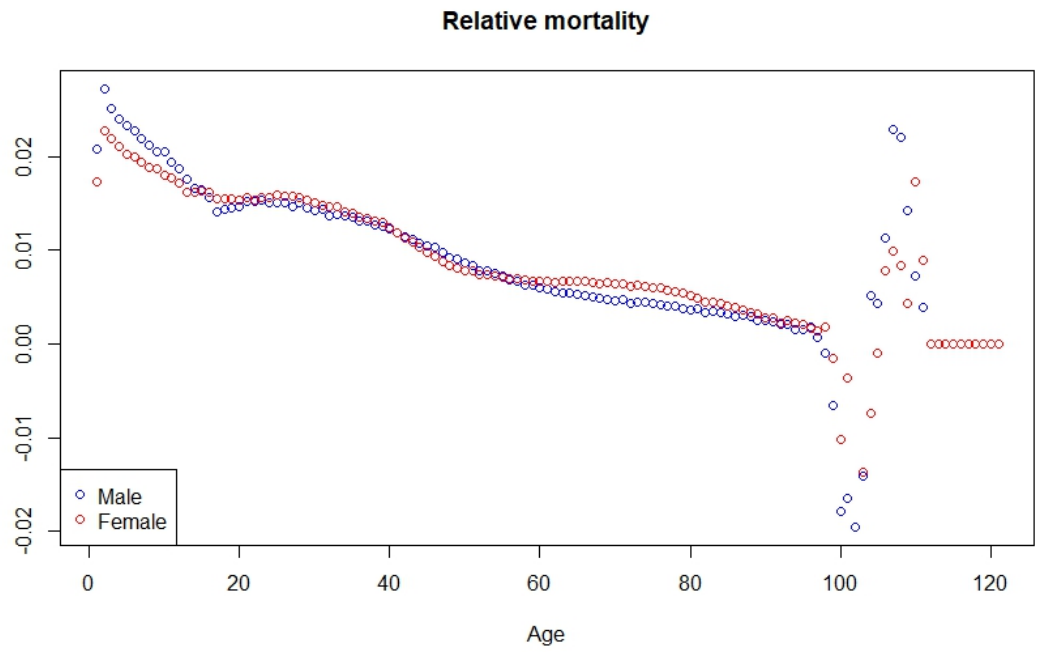


Figure 5.2: Values of relative mortality calibrated on Dutch population data. Every point corresponds to the relative mortality at that age for both males and females. The values are almost all positive, meaning that the mortality trend underestimates the real mortality and is corrected by these values. The values for both male and female sum op to one.

Mortality Trend

Lastly, vector **k** of Equation 5.1 is determined. Similar to vector **b**, vector **k** is the results of the singular value decomposition in combination with a rank one approximation. **k** has to deal with the same uncertainties a **b**. Unlike **a** and **b** is **k** dependent on the year instead of an age. Vector **k** represents the mortality trend over the past years. As the medical world has been improved over the past years, a decreasing mortality trend is expected. A visualisation of the mortality trend is shown in Figure 5.3, which confirms the expectation. Multiple spikes can be observed. The fist spike is observed around 1890 in which only the male mortality trend shows a spike. Around 1920 and 1942, one observes spikes in both the male and female mortality trend. These spikes are direct results of the first and second world war, respectively.

The obtained mortality trends form the basis of the forecast of the mortality rates. The trends are fitted as ARIMA models using R-package *Forecast*. *Forecast* has a method, *auto.arima* which aims to find the best fitted ARIMA model for a given time series. This method is used to determine the best ARIMA model fits for both the male and female mortality trend. Male mortality trend is given by an ARIMA(0,1,1) model, while the female mortality trend is given by an ARIMA(1,1,1) model, both have a drift term. The coefficients of the fitted ARIMA models can be found in Table 5.1. As discussed in Chapter 3, the parameter uncertainty of the ARIMA model approximation for the mortality trend is dominated by the drift. The drift can be interpreted as the slope of the time series. A larger time horizon results in a more reliable estimate of the drift. A decently large time horizon is taken as part of the calibration, hence the obtained drift is assumed to be a reliable estimate.

	Ar1	Ma1	Drift
Male		-0.4788	-1.1346
Female	0.4342	-0.8532	-1.4856

Table 5.1: ARIMA coefficients for the male and female mortality trend. Both are given as ARIMA(1,1,1) which is as explained in Chapter 3.

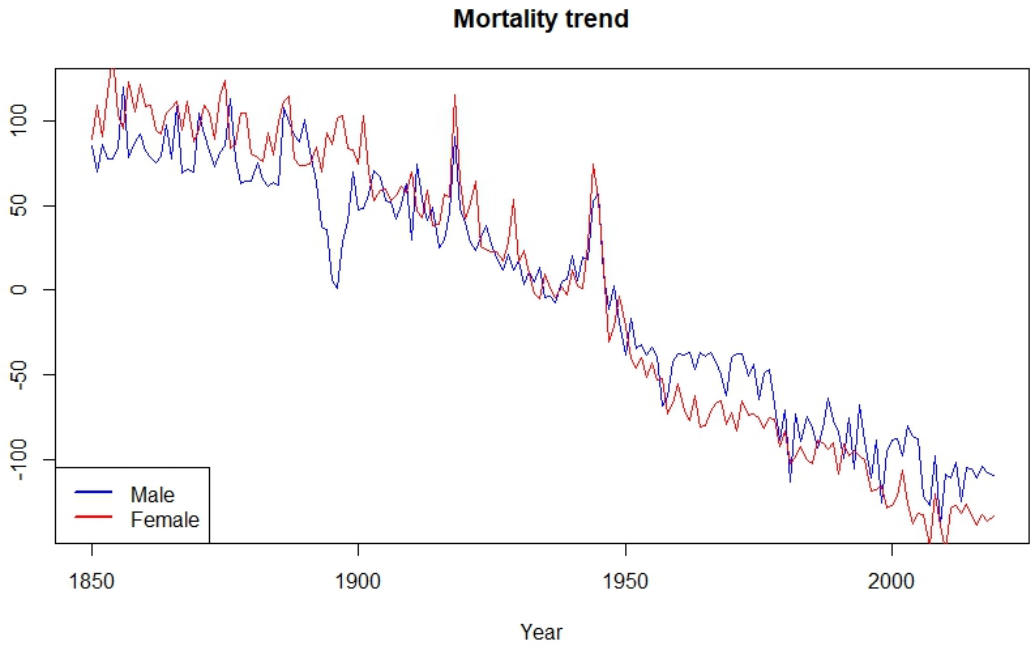


Figure 5.3: The mortality trend over a period from 1850 until 2019. Both male and female mortality trend are given as a decreasing series since the medical world has improved over the years. Negatively oriented spikes result from mass deaths while positively oriented spikes result from developments in medical world. The values of both male and female sum op to zero.

5.1.2. Forecasting of Mortality Trends

The fitted ARIMA models are used to forecast the mortality trends. First some confidence intervals of the simulations are determined. The confidence intervals are determined using the bootstrapping method described in Section 3.1. The obtained confidence intervals are shown in Figure 5.4, in which both the eighty and ninety percent confidence interval are shown. The confidence interval of the forecast female mortality trend is small resulting in more accurate predictions. This follows from a more stable decrease in mortality trend over the latest years. Because of this, the trend is more easily predictable.

Based on the forecast mortality trend, the relative mortality and the mean mortality rate a forecast mortality table can be determined. For every forecast mortality table, the value of the relative mortality as well as the mean mortality rates stays the same. Hence only the forecast mortality trend results in differences in the mortality tables. Because of this will only vector $\mathbf{k}$, the mortality trend, be used as input data in the XGBoost models. A total of 1400 simulations are created in which 1000 are used to train the algorithm, 200 serve as a testing set and 200 serve as a validation set. The simulation forecast till year 2191, which gives a reasonable estimate for the future. A sample of the forecast male mortality trends is shown in Figure 5.5. The further the prediction horizon increases, the more diverse the predictions become.

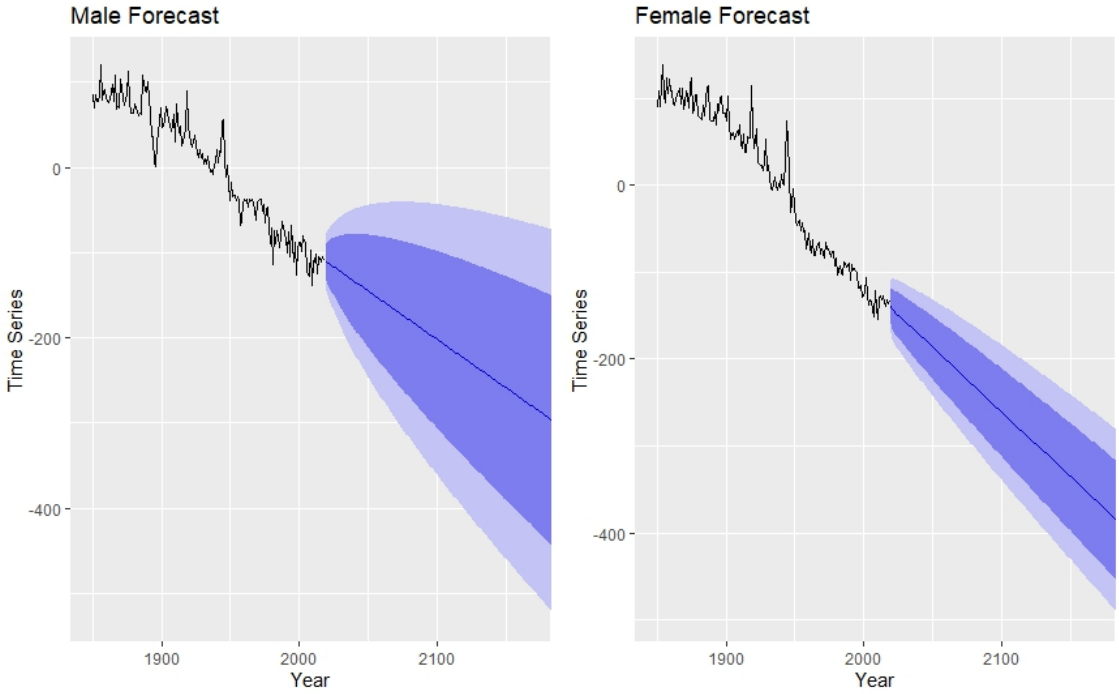


Figure 5.4: Confidence intervals of the forecast of mortality trend **k**. The left shows the male forecast, and the right shows the female forecast. Dark blue denotes the 80% confidence interval and light blue gives the 90% confidence interval.

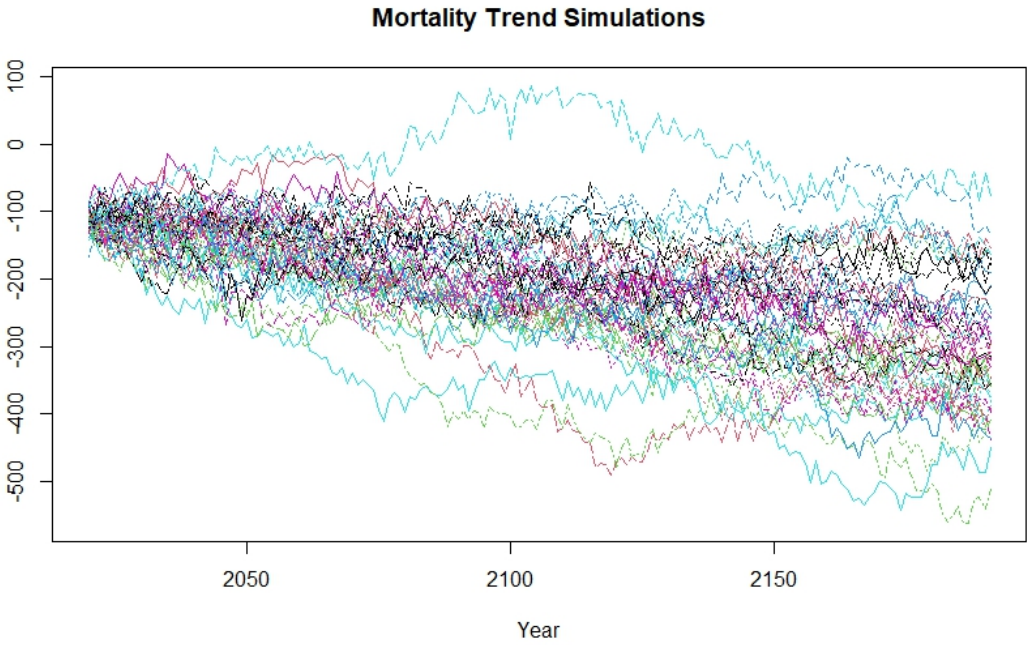


Figure 5.5: Fifty simulations of the male mortality trend. The dispersion becomes more apparent at larger horizons. Most simulations are decreasing which is a result of the decrease in mortality trend for past years.

5.2. Extreme Gradient Boosting

The Lee-Carter model is calibrated and simulations can be made based upon the ARIMA model. The simulations can be again be used to determine one realisation of the BEL and SCRs. These objects combined form the input for a machine learning method. XGBoost is the machine learning method used in this report and the working of it can be found in Chapter 4. The features used in the model consists of a simulation of the male and female mortality trend over a horizon of 172 years. An example of simulations for the male mortality trend is given in Figure 5.5. Based on this features, one aims to find approximations of the BEL and The SCRs, the target variables. Instead of predicting the SCRs, the BEL of the stressed situation is predicted. The standard portfolio used in this research consists of hundred thousand Dutch policy holders of a life insurance. The male to female ratio is almost one to one with ages varying between 0 and 110. All target variables are of the same order, namely 10^7. This observations holds for almost all combinations of mortality tables used in this research. This plays an important role in the loss function. If the target variables would be of different order, the target variables with the highest order would be considered first as the gain is generally bigger for these variables. This could result in good prediction for these target variables, while the smaller target variables have bad predictions. Since all the target variables are of the same order, they all have a similar gain and good predictions for all variables are expected. Since multiple target variables are defined this problem is defined as a multi-output regression problem.

The standard implementation of XGBoost can, however, only predict one target variable. There are different modifications of XGBoost possible such that multiple target values can be predicted. The most straightforward method is by building an XGBoost model for every target variable individually. Doing so results in loss of coherent characteristics of the target variables. It does, however, imply a simple approach to solve the problem. Creating a model for every target variable can however become quite cumbersome. As an alternative to incorporate some of the coherency, a hierarchy can be defined. Based on this hierarchy, the individual models can be trained in order in which one includes aspects of the past target variable models. This could result in a more accurate prediction, especially for the target variables later in the hierarchy. This way, more coherent aspects are kept to train the model. As performance of this approach is highly dependent on the order of training, it is not used in this report.

To incorporate all dependency between target variables, independent of order, one can decide to use a multi-target regression tree. This model is discussed in depth in Section 4.4. As the name suggest, a multi-target regression tree has leaves which represent multiple values at once, represented as a vector. This does, however, complicates the machine learning. A standard objective function is not realisable anymore as more dimensions are introduced. Creating a higher dimensional objective function implies possibilities. In turn, both the gradient and hessian used in XGBoost change as well. Instead of working with a gradient vector in XGBoost one uses a matrix in which every column consists of one gradient vector for that target value. Similarly, the hessian becomes a matrix in which every column consists to the diagonal of the hessian matrix of a target variable. First, the performance metrics as well as the chosen loss function are discussed. Followed by the method of parameter estimation for XGBoost. Next, both the individual model approach as well as the multi-target approach are discussed.

5.2.1. Performance Metrics and Loss Function

Determining the performance of a machine learning method is done by a metric. At least one metric is required to determine the performance. In this research, there are two metrics defined. The first metrics determines the percentage of predictions with an error greater than a given allowed deviation. This is based upon two separate requirements: the management and reporting requirement, as introduced in Chapter 1. In order for a model to satisfy the management requirement, the predictions must in 99.5% of the cases deviate within 1%. Similar, to satisfy the reporting requirement, the predictions must in 99.5% of the cases deviate within 0.5%. As

discussed in Section 4.5, this metric cannot be used as loss function since it is not differentiable. Besides, it will stop minimizing when the requirements are satisfied, while even better predictions might be within reach. This metric is only determined for a already trained model to determine fulfillment of the requirements.

Another metric which monitors the performance after every regression tree step needs to be defined. This method shows the improvements made by adding more regression trees. The standard root mean squared error (RMSE) is used for this purpose. The RMSE implies more specific information than the requirement metric as it is only equal to zero if the predictions are equal to the observations. Hence, the full performance increase can be observed by this metric. The RMSE is twice differentiable and can thus be used a the loss function. There is however a more convenient choice. As discussed in Section 4.5, the squared error (SE) multiplied with a constant has clever first and second order derivatives.

It is possible that only one of the target variables has decent performance, while the others have worse of even bad performance. To observe such behaviour there is a distinction made between individual performance and combined performance. The metrics are applied to the individual target variables predictions as well as the combined predictions. For the management and reporting requirements this means that all target variable must deviate within the allowed interval all at the same time for one prediction. The value of the requirement metric on the combined prediction is always of worse or equal accuracy compared to the individual target variables. Combining all the metrics allows the performance of the XGBoost models to be evaluated as well as determining the fulfillment of the requirements.

5.2.2. Parameter Calibration of XGBoost

The performance of a machine learning is highly dependent on the the choice of parameters. A standard regression tree has multiple parameters such as the maximum depth of the three, maximum number of leaves etc. XGBoost adds a choice of three other parameters λ, η and γ . λ is the regularisation term on weights and is generally taken equal to one. λ adds a penalty the the loss function which helps to reduce overfitting. η is the learning parameter, which determines how much every tree weights in the final approximation. Default value of η is 0.3 and is in most cases sufficient. A larger eta implies faster and less accurate improvements while smaller eta implies slower and more accurate improvements. γ determines the shallowness of all regression trees in which higher values result in shallower trees. γ is used in the split finding to find the best split which has a least an improvement greater than γ, see Section 4.3. γ has a default value of zero, resulting in a regression trees as large as possible. γ is connected to both maximum depth and maximum number of leaves as they all regulate the number of splits.

On top of this, one need to specify the maximum number of trees to train. These regression trees are combined, using η, into one predictive model. A small η is generally accompanied with a large number of trees and vice-versa. Besides, XGBoost has multi process capabilities. This value gives the number of CPU cores available for the training. This value is always set equal to number maximum available cores on the device.

To determine the optimal combination between all parameters, *GridSearchCV* is used (Pedregosa et al., 2011). *GridSearchCV* applies cross validation to the input data, while trying different combinations of parameters. The options for the parameters are user input and can be defined by some interval. This method of parameter calibration is used to determine the correct combination of parameters for an XGBoost model. *GridSearchCV* is, however, a computationally heavy method, depending on the number of times cross validation is applied as well as the number of parameter combinations. The calculation time increases as the number of folds increases. Instead of applying k-fold cross validation parallel with *GridSearchCV*, it is possible to do delay the cross validation. First, the optimal parameter combination on the complete training set is determined using the different combinations considering only a 2-fold cross validation. Second, the optimal parameter

combination is in turn used for k-fold cross validation to determine the performance on unseen data. This method requires a lot less computation time, while not altering the outputs significantly.

When determining the optimal combination of parameters for the multi-target regression approach, one needs a similar structure. Parameter calibration of the multi-target approach can be found in Algorithm 5. As this is a self created implementation, cross validation is completely separated and not applied within the parameter calibration. First the best performing parameter combination is determined. k-fold cross validation is in turn applied on the machine leaning model with this parameter combination.

Algorithm 5 Parameter Calibration for Multi-Target XGBoost (Based on *GridSearchCV*: (Pedregosa et al., 2011))

1 Define possible values for λ, γ, η and number of trees given in arrays Λ, Γ, Ω and N, respectively.
2 For $\lambda \in \Lambda$:
  a For $\gamma \in \Gamma$:
    For $\eta \in \Omega$:
      For $n \in N$:
        i Train Multi-Target XGBoost model on data set
        ii Determine performance:
          (1) RMSE
          (2) Management requirement
          (3) Reporting requirement
3 Return best performing parameter combination

5.2.3. Individual XGBoost Models

The first prediction approach is by defining individual models for every target variable. As stated, this does not take the dependency between target variables into account. Four independent models need to be trained, one for the BEL and three for the stressed BEL corresponding to the three different SCRs. Every model needs at least one metric as well as one loss function. As discussed in Section 4.5, the standard squared error of the residuals is a convenient loss functions. Furthermore, this function is a build-in loss function in the XGBoost package. The loss function is chosen independent of the target variable and can thus be applied to every individual model.

$$L : \frac{1}{2} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Both the first derivative and the diagonal of second derivative matrix are easily determinable and have a convenient form. The loss function is convex and can thus be minimized to approach the global minimum.

$$\frac{\partial L}{\partial \hat{y}_i} = \hat{y}_i - y_i$$
$$\frac{\partial^2 L}{\partial \hat{y}_i^2} = 1$$

As stated in 4.5 is the RMSE a good choice for the metric and will therefore be used for every model. The reporting and management metrics are in turn only used to determine the satisfaction of the requirements.

Parameter Choice

Every model needs to be individually treated for parameter calibration. *GridSearchCV* is applied on every target variable in combination with all training features. The possible values for the

parameters are given in Table 5.2, in which the best performing parameter combination is given as well. Every individual model turns out to have the same parameter combination as given in Table 5.2. This combination of parameters is used to train the individual models. The value of γ is not of significant size for the starting situations as the starting data set consists of target variables of order 10^7. Hence in the starting phase, every tree is not limited by γ. Later on in the process, the improvements become smaller and γ start to play a more important role in creating the tree. The max depth of the tree turns out the be relative small. The possibility of overfitting is decreased in this way as the threes do not have that many leaves. Since the maximal depth of the trees is relative small, more threes are required to get a decent prediction, which is exactly as observed in Table 5.2. This does however influence the optimal choice op η. As there are many small threes to train on, the learning parameter needs to be smaller than the standard case, resulting in a more conservative approach. The value of λ is taken as the standard value given by XGBoost. On top of this parameter combination, an early stopping criterion is added. After 25 iterations without improvements, training is terminated and the so far calibrated model is returned. Besides, split finding takes place using the histogram method described in Chapter 4.

Parameter	**Parameter Options**	**Value**
γ	25, 50, 100, 500	50
λ	0.5, 1	1
η	0.01, 0.02, 0.03, 0.3	0.02
Max depth	2, 3, 4, 5, 6	4
Number of trees	800, 900, 1000, 1100	900

Table 5.2: Best performing parameters for all individual trained models. These values are obtained using *GridSearchCV* where all parameter options are as given.

Prediction

As stated in Chapter 1, obtaining the input data for machine learning is a computationally heavy and above all time consuming process. Hence to get the best time improvements, one needs to know the minimal size of the training set required to satisfy the management and reporting requirements as given in Chapter 1. To get an estimate of a decent size of the training set, the training size is taken as a variable. For every training size all individual models are trained and the performance is determined. The RMSE estimated on the training set of different training sizes is shown in Figure 5.6 for all target variables. The training RMSE is an increasing series, which is a direct result of overfitting at the smaller training sizes. If the size of the training set is small, the performance on the training set is biases as it can simply 'remember' the data set. This does however changes around 650, in which the RMSE stagnates. Increasing the size does not significantly change the performance. As there is still some randomness within XGBoost, there is a lot of fluctuation visible in Figure 5.6. Repeating this setup multiple times and taking the average implies more stable lines with smaller variance.

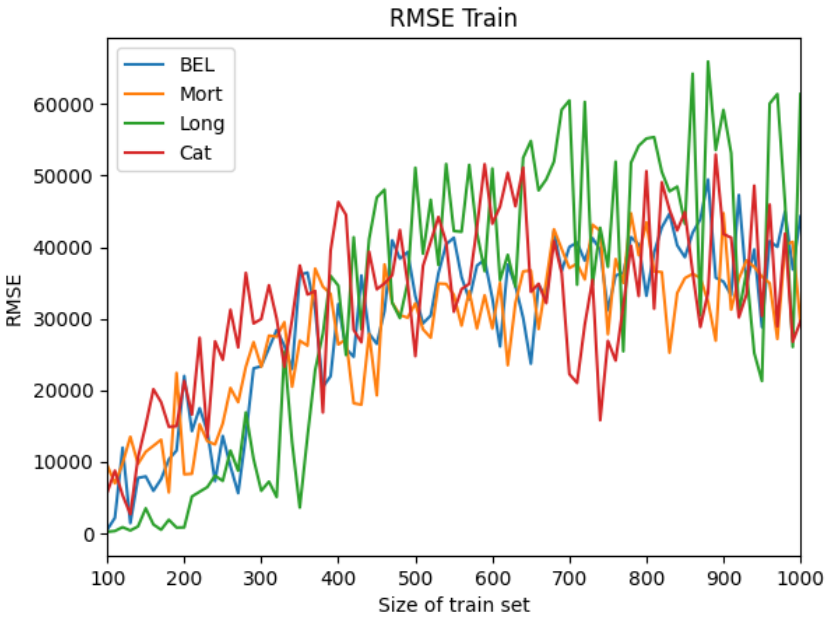


Figure 5.6: The RMSE of training data plotted versus training set size (multiples of 10 only) in which every target variable has an individual model. For the BEL and all three SCRs one observes an increase up till a size of 650. After 650 stagnates the RMSE for all target variables.

The real performance should, however, be estimated on the test set instead of the training set. This test set is taken equal for every model and is independent on the size of the training set. The test set consists of 200 observations which are never used in the training phase. Because of this, it is expected that the RMSE on the test set is higher than the RMSE on the training set. Besides, it is to expect that the performance on the test set increases as the size of the training data set increases. One should however be careful not to underfit, it the data becomes too diverse, XGBoost is not able to create a generic approximation. Instead it will give a general value such as the mean. In Figure 5.7, one observes indeed a decreasing RMSE as the size of the training set increases. The same stagnation is observed starting from 650.

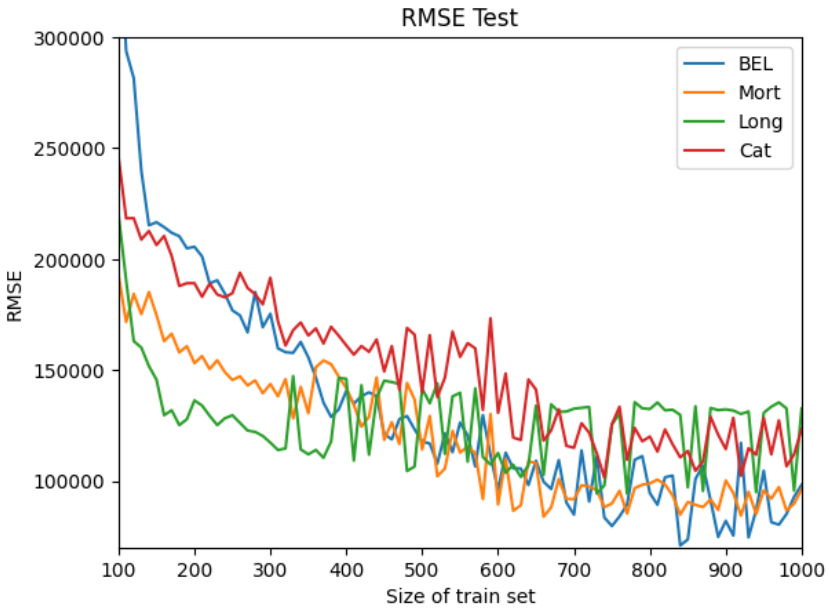


Figure 5.7: The RMSE of test data plotted versus training set size (multiples of 10 only) in which every target variable has an individual model. For the BEL and all three SCRs one observes an decreasing series up till a size of 650.

Both the management and reporting requirements needs to be checked to accept or decline the model. The deviation of every data point in the test data set is determined. As it is required that all predicted target variables of once instance should satisfy the requirements at the same time, the maximum of the individual deviations is taken. The percentage of errors deviating more than the allowed inaccuracy is shown in Figure 5.8. The management requirements (1%) are satisfied at a training set of size 650, which is where the RMSE stagnates. The reporting requirements (0.5%) do however not reach the desired error of half percent. Instead, it converges to two and a half percent and no further improvement in observed. One could increase the training set to observe possible further improvements. As the improvements are not that big anymore after a size of 650, there is no direct need to do so.

Cross Validation

Cross validation is applied in order to test the ability of XGBoost to predict unseen data. A k-fold cross validation is used. In general, k is chosen between 5 and 10. As lower value have higher bias, a 10-fold cross validation is applied. For every training size, 10 times a training set is defined consisting of 9 folds, while the performance is determine on the remaining fold. The performances of the individual folds are shown in Figure 5.9a, in which similar behaviour is observed for every fold. Moreover, in terms of the RMSE, the individual folds exhibits behavior similar to the entire training set, whose RMSE is plotted in Figure 5.7 Hence, according to the RMSE, the model performance is independent of the choice of train and test data. Figure 5.9a only shows the RMSE of the BEL, the other target variable imply similar figures and are therefore omitted. A similar test can be constructed for the management and reporting metric as shown in Figure 5.9b for the management metric. It shows that the percentage of of predictions deviating more than one percent behave similar for every fold. Hence, based on these metrics one can confirm that the model performance is independent of the unseen data. Moreover, the performance is equally as good for all of the k folds.

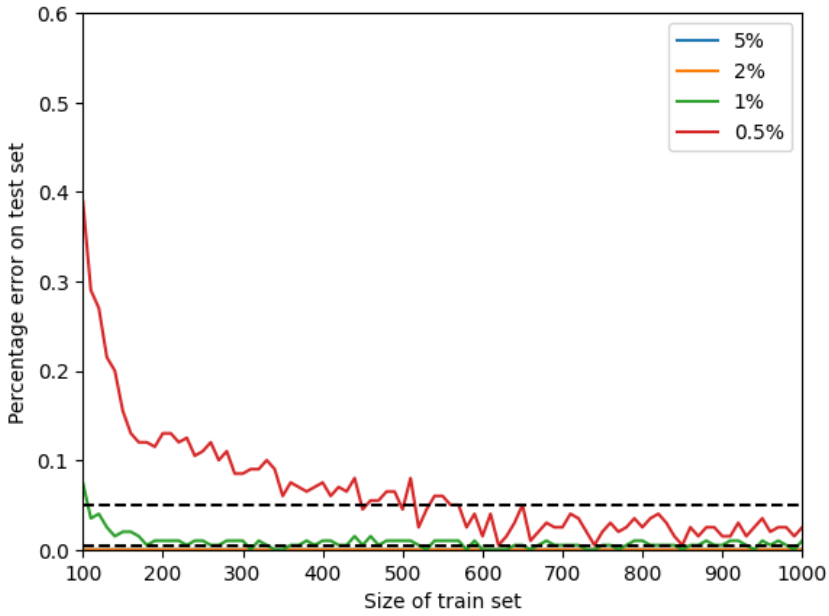
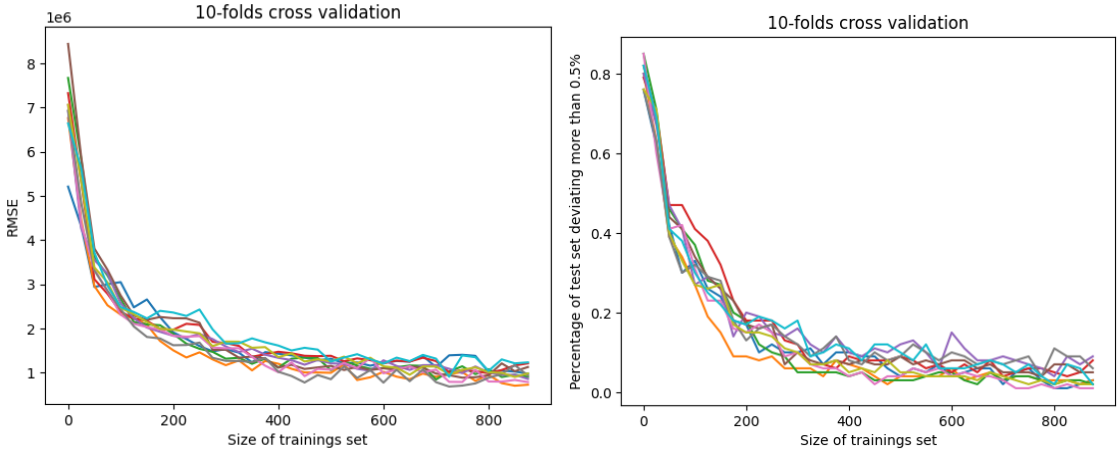


Figure 5.8: Percentage of test set predictions deviating more than 5%, 2%, 1% and 0.5%, respectively. For every target variable is one model trained and these are combined to one final predictor. Two black dotted lines are shown, one corresponding to 5% and the other to 0.5%. When the lines stay below the black line at 5%, 95% of the predictions varies within either 5%, 2%, 1% and 0.5%.



(a) RMSE metric on target variable BEL

(b) Reporting metric on all target variables combined

Figure 5.9: 10-fold cross validation of the individual trained model. Similar behaviour is observed for every of the fold. Only BEL is shown as the other target variables imply similar results.

5.2.4. Combined XGBoost Model

As a second modelling approach, a combined model is defined. This model tries to predicts all target variables at the same time while only training the model once. This method does take possible dependencies between target values into account. XGBoost does not have a standard implementation for multi-target regression, hence a completely new design is created in Section 4.4. It turns out that, in order to be solvable, the loss function must have a one-dimensional codomain. Since, the leaves of a tree are vectors, a multi-dimension loss function is needed to optimize as well as a multi-dimensional metric. The loss function is taken as a generalization of the one-dimensional standard squared of residuals. This loss function is minimized to train the model.

$$L : \frac{1}{2} \sum_{i=1}^{n} ||\hat{\mathbf{y}}_i - \mathbf{y}_i||_2^2$$

As discussed in Section 4.4 the second order derivative has an upper boundary defined by the diagonal of the matrices. Hence only the diagonal elements are considered. Just like standard gradient boosting, the first and second order derivative are convenient. Note, both the first and second derivative are vectors in this case.

$$\frac{\partial L}{\partial \hat{\mathbf{y}}_i} = \hat{\mathbf{y}}_i - \mathbf{y}_i$$
$$\frac{\partial^2 L}{\partial \hat{\mathbf{y}}_i^2} = \mathbf{1}$$

On top of the higher dimensional loss function, higher dimensional metrics are required. As discussed in Section 4.5 the RMSE can easily be extended to a multi-dimensional variant. Similarly, for the management and reporting requirements, the vector leaves are considered as input instead of the individual target variables. It is expected that the combined model has lower RMSE and a better performance according to the management and reporting metrics.

Parameter Choice

Similar to the individual models are the XGBoost parameters determined by testing different combinations, following Algorithm 5. The options for the parameters are the taken the same in the individual model case, as can be seen in Table 5.3. The optimal performing combination of parameters is almost identical to the individual models in which only the number of trees for the combined model is larger than in the individual models. This increase implies a slower convergence than the individual models. Thus the combined model needs more iterations to approach the global minimum in comparison to the individual models. It is observed, later in this chapter, that the later trees do not add that much information to the approximation. Because of this, the same early stopping criterion in introduced as used in the individual models. If the performance does not improve in 25 consecutive trees, training is terminated, and the current best iteration is returned. Besides, split finding takes place using the histogram method described in Chapter 4.

Parameter	Parameter Options	Value
γ	25, 50, 100, 500	50
λ	0.5, 1	1
η	0.01, 0.02, 0.03, 0.3	0.02
Max depth	2, 3, 4, 5, 6	4
Number of trees	800, 900, 1000, 1100	1000

Table 5.3: Best performing parameters for all individual trained models. These values are obtained using Algorithm 5 where all parameter options are as given.

Prediction
Just as with the individual model is one interested in the minimal training size which satisfies the requirements. To do so, the influence of the training size on the performance is determined. For every size of the training set, ranging from 100 to 1000, is the RMSE on the training set determined. In contrast to the individual model approach, only one RMSE value is determine for every training size as the vector output is taken as one prediction, see Section 4.5. The results are visualized in Figure 5.10, in which similar value are observed as with the individual models. Again, overfitting is visible for too small training sets. The process stagnates around 700, which is later than the observed in the individual models. The fluctuations are again the result of randomness of XGBoost and can be eliminated by repeating this procedure and taking the average value for every training set size.

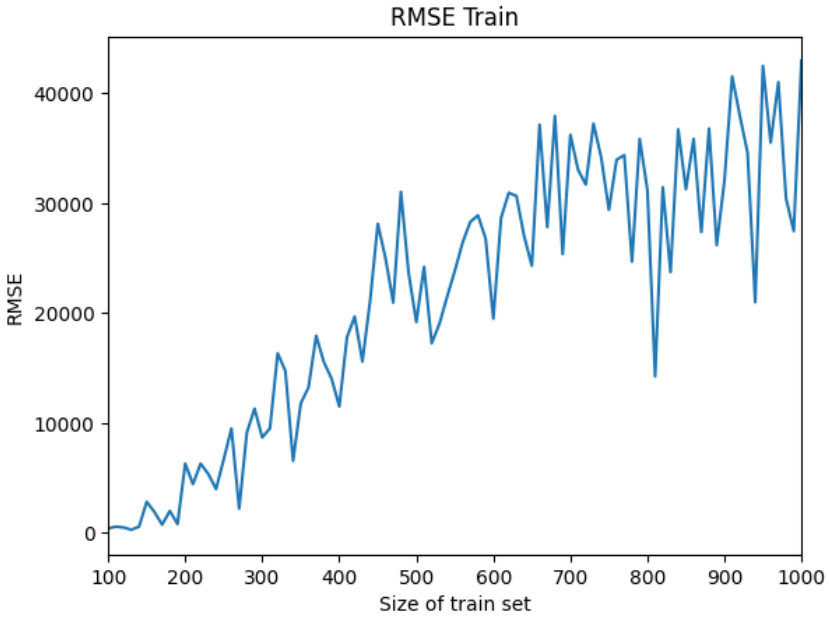


Figure 5.10: The RMSE of training data plotted versus training set size (multiples of 10 only) in which all target variable are considered at once. For the BEL and all three SCRs one observes an increase up till a size of 650. After 650 stagnates the RMSE for all target variables.

The performance on unseen data gives, however, more insight in the performance of the model. An independent test set is defined, consisting of 200 observations. This test set consist of the same observations used for the individual models. The RMSE values of this test set are determined for all training sizes and are shown in Figure 5.11. In this figure is a decreasing series observed, which stagnates around 650. The values of the combined model are, in most cases, smaller than the RMSE of the individual models. This is consequence of the dependence between the target variables. The target variables are highly correlated as they are all calculated in a similar manner with the same input. For example, the stressed BEL corresponding to the mortality SCR is determined using the same mortality table multiplied with a constant, see Chapter 2. Hence, one can conclude that the RMSE performance of the combined model is better than the the performance of the individual models on the unseen data.

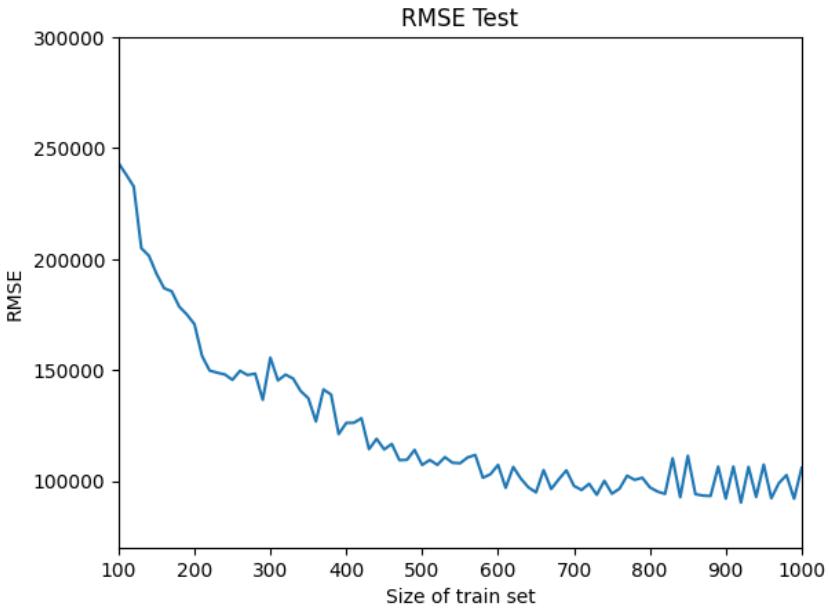


Figure 5.11: The RMSE of test data plotted versus training set size (multiples of 10 only) in which all target variable are considered at once. An decreasing series up till a size of 650 is observed. After 650 stagnates the RMSE.

While the RMSE metric shows promising results, the requirement metrics show otherwise. The management requirements are already satisfied for a training size of 450, which is an improvements on the individual models. The reporting requirement are, however, never satisfied as seen in Figure 5.12. Starting from a training size of 600 does 95% of the test set vary within 0.5%. Similar to the individual models do the reporting metric converge to an error of 2.5%. At this point, 97.5% of the test set varies within 0.5%, which is not accurate enough. The convergence to this point is faster than with the individual model approach. Later on, the size of the training set is increased further to observed possible fulfillment of the reporting requirements. The combined model has a better performance for a smaller training set in contrast to the individual trained models. As the training size increases, the difference are no significant anymore. One can, however, observe a slightly better performance of the combined model.

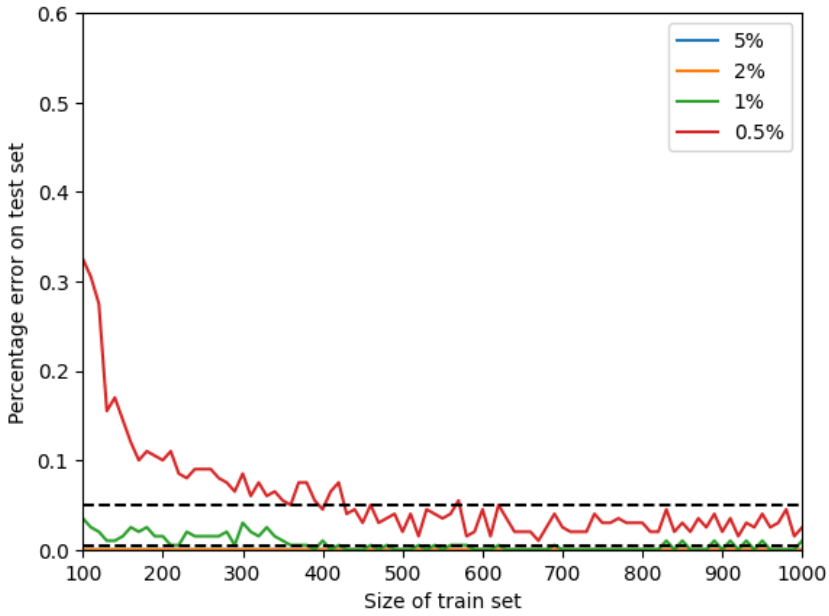
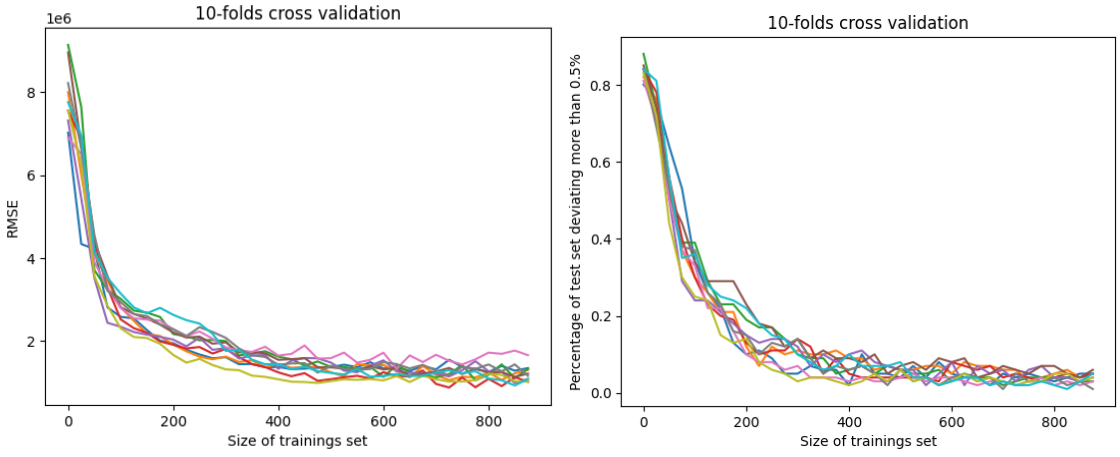


Figure 5.12: Percentage of test set prediction deviating more than 5%, 2%, 1% and 0.5%, respectively. All target variables are trained in one model. Two black dotted lines are shown, one corresponding to 5% and the other to 0.5%. When the lines stay below the black line at 5%, 95% of the predictions varies within either 5%, 2%, 1% and 0.5%.

A 10-fold cross validation is applied to ensure correctness of the model. For every fold is the performance on the test set shown in Figure 5.13. Only the RMSE of the stressed BEL corresponding to catastrophe SCR is shown as the other target variables show similar results. It can be seen that behaviour is independent of the fold. The reporting requirement shown in Figure 5.13b have even better performance. Taking the average of all the folds, one even fulfils the reporting requirement. Based on these metrics one can confirm that this model performance is independent of the unseen data.



(a) RMSE metric on SCR catastrophe

(b) Reporting metric on all target variables combined

Figure 5.13: 10-fold cross validation of the combined model. Similar behaviour is observed for every of the fold. Only SCR catastrophe is shown as the other target variables imply similar results.

5.3. Model Comparison

Both modelling approaches are analysed and based on the performances a better approach can be determined. Even though the errors are of equal order does the combined model have a better performance for smaller training sets and a slightly better performance overall. Hence, this model is so far preferred over training an individual model for every target variable. The main purpose of machine learning is a decrease in calculation time, hence calculation time is another import aspect. Creating the complete training set, for a portfolio consisting of roughly hundred thousand objects, took about fifteen minutes per calculation. A total time of ten days are required to create the train and test data. Hence, if one needs even more calculations, it becomes even more time consuming. By considering only 650 tables, only halve the initial time is required, already resulting in a decrease by a factor two. Training all the individual models to create Figure 5.7 took well above one hour, in which the management requirement are satisfied around a training set of size 650. The partition of time over the individual models can be found in Table 5.4. Besides this time estimate, it might be more important to get the duration of only one time training on a complete training set. For this purpose, a training size of 650 is chosen, as it is observed that management requirements are satisfied from this point. Hence, training the model on a set of 650 observations has a total duration of five days, this includes the time to obtain the data.

Model	Complete Process Duration in Seconds	Duration With Training Size 650
BEL	1138	11.0
SCR Mortality	1123	11.0
SCR Longevity	1032	9.8
SCR Catastrophe	1010	10.0
Total	4301	41.7
Average	1075	10.4

Table 5.4: Duration of every model used in the individual model approach. The values are rounded to nearest second or tenth of a second to ensure a good signal to noise ratio.

A similar time observation can be formed for the the combined model. Creating Figure 5.11 took a little less than an hour, which is already an improvement. The total duration as well as a duration for only one training set is can be found in Table 5.5. Again, training the model on a set of 650 objects, has a total duration of five days, including the time to obtain the data. There is a reduction of fifteen minutes, which is a significant different on one hour, but not on five days. Based on this observation and the slightly better performance, the combined model is the preferred modelling approach. This model is highly dependent on the portfolio used to determine the input BEL and SCRs. To determine whether the reporting requirements are satisfied at any training size, the size of training size needs to be extended. On top of this, to get insight in the performance on other portfolio's, the model should be applied to other portfolio's as well.

Complete Process Duration in Seconds	Duration With Training Size 650
3384	30.1

Table 5.5: Duration of the combined model approach. The values are rounded to nearest second or tenth of a second to ensure a good signal to noise ratio.

5.4. Extending The Model

The model can be extended in multiple ways. One of the extensions is by using different portfolios, this is however difficult as the model is trained on the BEL and the stressed BELs, corresponding to SCRs, specific to one portfolio. Another way is testing the model with a larger training set. In this way, the performance on the reporting requirements can be determined. The calculation time can be further reduced by using the GPU of a device in the machine leaning step. This is nice to see, it is, however not relevant on a total time of five days.

5.4.1. Different Portfolios

The discussed models are optimized for a specific portfolio and can therefore not one to one be applied on other portfolios. This portfolio results in values of the BEL and SCRs, which in turn are used as training material for XGBoost, hence the dependence on the portfolio. Portfolios consisting of similar products and quantities, do however behave well with a predictive model trained on each other. There is a thin line between similar portfolios as there are a lot of things to consider. If two portfolio consists of the same products, while the characteristics of the policy holders differ a lot, the model does not perform well on the new portfolio. If the characteristics however have a lot of overlap, the model performances with a similar accuracy on both portfolios.

Over a small time interval there a not that many changes in a portfolio. The main changes consists of some introductions of new policyholder and some terminations of contracts. Depending on the kinds of products present in the portfolio, this is more realisable than others. Hence, a portfolio of only one insurer can be used while only training the model once. An example of three different portfolios can be found in Figure 5.14. One of the lines represents the standard portfolio, while the other two have an increase or decrease of two percent of the policy holders. The removed policy holders from the smaller portfolio are randomly chosen. Similarly, to create the bigger portfolio, a random sample is taken and some randomness is added to the characteristics of the policy holder. These 'new' policy holder are added to the standard portfolio to create the bigger portfolio. It is observed that in this case, the performance is similar and the management requirements are always satisfied for training set of size 800 and larger. The standard portfolio as well as the smaller portfolio satisfy the management requirements from 600. The reporting metrics behave similar for all three portfolio's.

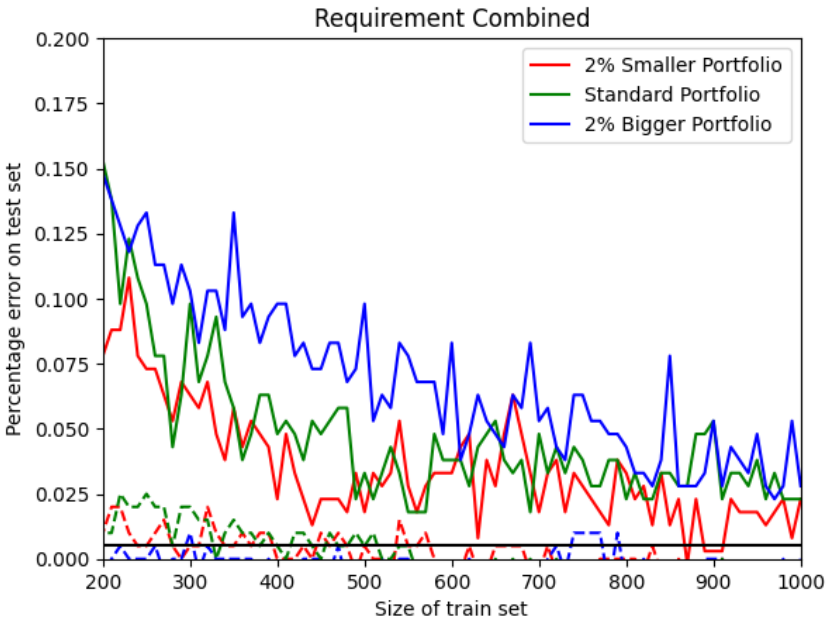


Figure 5.14: Management and reporting requirements on three different portfolios. The model is trained only on the standard portfolio (green), while the other two portfolios differ from the standard portfolio by 2%. The dashed lined give the management requirements and the full lines give the reporting requirements.

If one has large differences in portfolios, this becomes more complicated. A model, calibrated on largely different portfolio, cannot be used to make decent predictions. A completely independent and different portfolio is used to show this. This different portfolio consists of the same kind of insurance products, the premium holder characteristics are however changed drastically as well the occurrences of some products. For an example of bad performance on a different portfolio, one can consider Figure 5.15 in which performance is assessed on a completely different portfolio

than the portfolio used to train the model. If one needs to calculate with a different, independent portfolio, one needs to fit a new model based on that portfolio. On top of that, it might be beneficial to re-estimate the parameters of XGBoost to obtain a more accurate fit. A new training set needs to be defined, consisting of some observation of the BEL and SCRs. Although, creating new training data is time consuming, it is less time consuming than determining the BEL and SCR for an even bigger set. Creating a more generic model would make the training more intensive as there are more features to work with. On top of this, to create a decent performing model, a larger training size is needed.

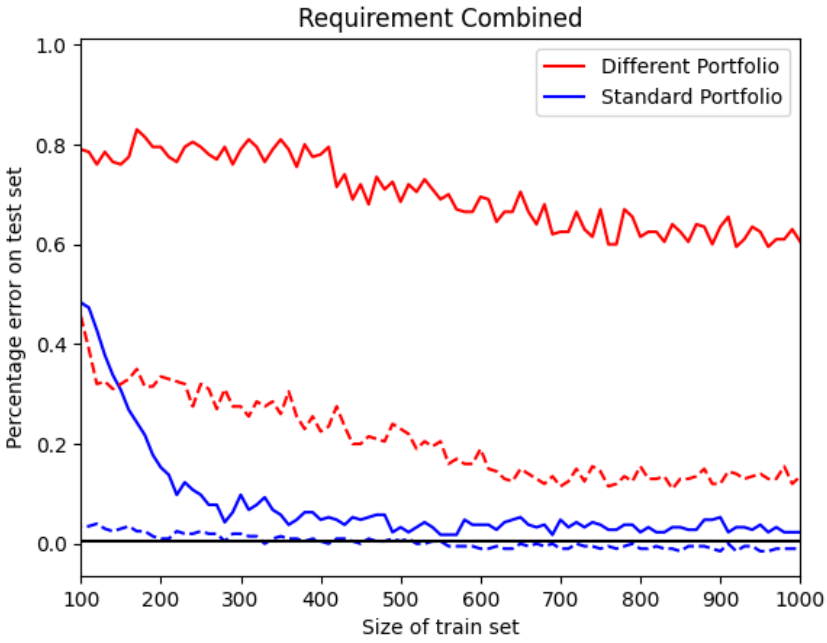


Figure 5.15: Performance of trained model on an different portfolio. While the management of the standard portfolio are satisfied at some time, does other portfolio never fulfill the management requirements. Even worse performance is observed for the reporting requirements.

5.4.2. Larger Training Set

The reporting requirement states that in 99.5% of the cases, the prediction should deviate within 0.5% of the observed value. As observed is this requirements not satisfied for training size smaller or equal than one thousand. To check if the reporting requirements are ever satisfied, a larger training set is needed. Determining more BEL and SCRs is computationally heavy but should be considered. To this end, a smaller portfolio is defined, consisting of half of the original portfolio. Determining the BEL and SCRs for this smaller portfolio takes about halve as long and the time required for creating a larger training set is therefore greatly reduced. By creating the new training set with size three thousand and only considering the reporting requirements, one obtains Figure 5.16. It can be seen that the reporting requirement is satisfied starting with a training size of 1750. This is almost three times as much as for the management requirements.

A similar study is applied on the individually trained models. Separately, the satisfy the reporting requirements around 1750 as well. Combining the individual models to one predictor implies however way less accurate predictions. Just a the training size reaches size 3000 are the reporting requirements satisfied. Considering this, one observes the real power of the combined model.

5.4.3. GPU Calculations

The calculation time of XGBoost can be further reduced by using the GPU of a device. As stated in Chapter 4, XGBoost has a built in method which allows use of the GPU. The GPU is used in split

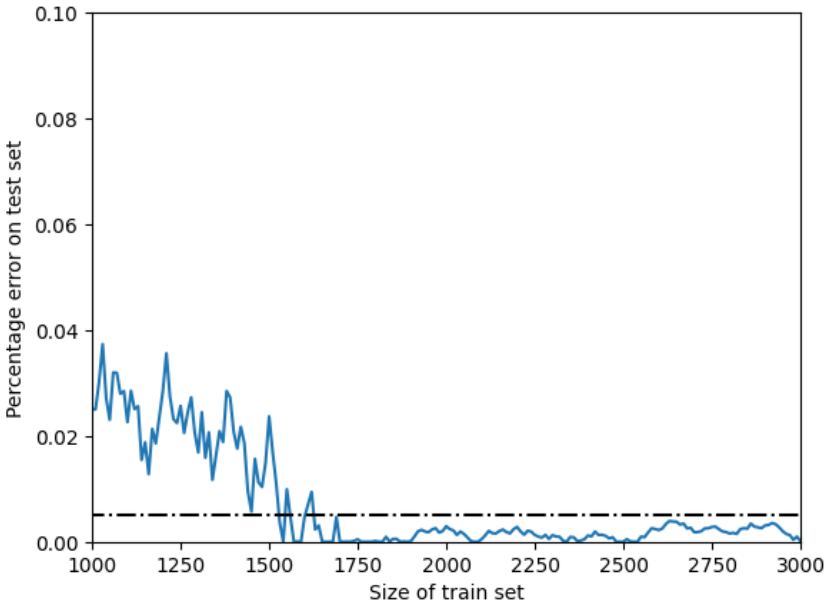


Figure 5.16: Combined model trained on different training sizes versus the reporting performances. At a training size of 1750 are the reporting requirements always satisfied.

finding for the regression tree as this consists of simple operations. Using a NVDIA GEFORCE RTX 3080, the calculation time is reduced by a factor ten, which is a decent improvement. Increasing the number of trees, while simultaneously decreasing learning rate η results in a more conservative approach. This in turn requires more split findings and the GPU could have even more impact. For this research, the GPU is not used as training the model and predicting with this model takes less than a minute, which is already a huge improvement. Besides, the XGBoost GPU usage is only allowed for NVDIA GPU's as the code is written in CUDA, which only communicates with NVDIA GPU's.

5.5. Performance on Optimal Size

As discussed in Section 5.3, the better performing model is chosen as the model which predicts every target variable at once. It is seen in Figure 5.12 that starting from a training set of size 600, the management requirements are satisfied. To ensure correctness, a larger training set of size 650 is taken as the correct size for management purposes. Similar, one observes in Figure 5.16 that a training size of 2000 is large enough for reporting requirements. From now on, the focus is on management requirements. Reporting requirements can be discussed in similar fashion, it is however more computationally heavy.

There is already a model trained with a training size of 650, as can be seen in Section 5.2.4. Here are different training sizes compared based on their performance. The parameters used in Section 5.2.4 are however biased as they are determined using the complete training set of 1000 observations. If an insurer only creates a training size of 650, one can observe different parameters used in XGBoost. Based on the training set, correct parameters can be determined which are not biased. For the main portfolio used in this research, there is a slight difference in the best XGBoost parameters. The best performing parameters combination is given as in Table 5.6. The number of trees is decreased, which is a direct result of the smaller training set. As there are less observations to train on, a smaller error is more easily obtained. Since the smaller error does not improve much anymore after some time, a smaller number of trees is allowed.

Parameter	Value
γ	50
λ	1
η	0.02
Max depth	4
Number of trees	800

Table 5.6: Best performing parameters for a training size of 650. These values are obtained using Algorithm 5.

Based on this parameter choice, the model can be trained. It turns out that after 717 iterations, the error does not decrease significantly any further and the model terminates. All the remaining observations are combined into one test set of which a scatter plot is shown in Figure 5.17. This figure only shows the performance of the BEL as the other target variables have similar behaviour. Of the 550 test sample are there 549 within the management requirements, resulting in a correctness of 99.8% which is as allowed. The reporting requirements are, however, only satisfied by 539 of predictions, which is a correctness of 98%. Hence, as expected are the management requirements perfectly satisfied at a training size of 650 while the reporting requirement are not satisfied. Training this model and prediction the complete test data took in total around 30 seconds.

A similar test can be applied to a training set of size 1750. The optimal XGBoost parameters differ in this case only in the number of trees. The number of trees increases to 1250, while the other parameters stay the same. The management requirements give in this case a more narrow scatter plot. The reporting requirement are satisfied as well at this training size and a similar scatter plot as in Figure 5.17 can be created.

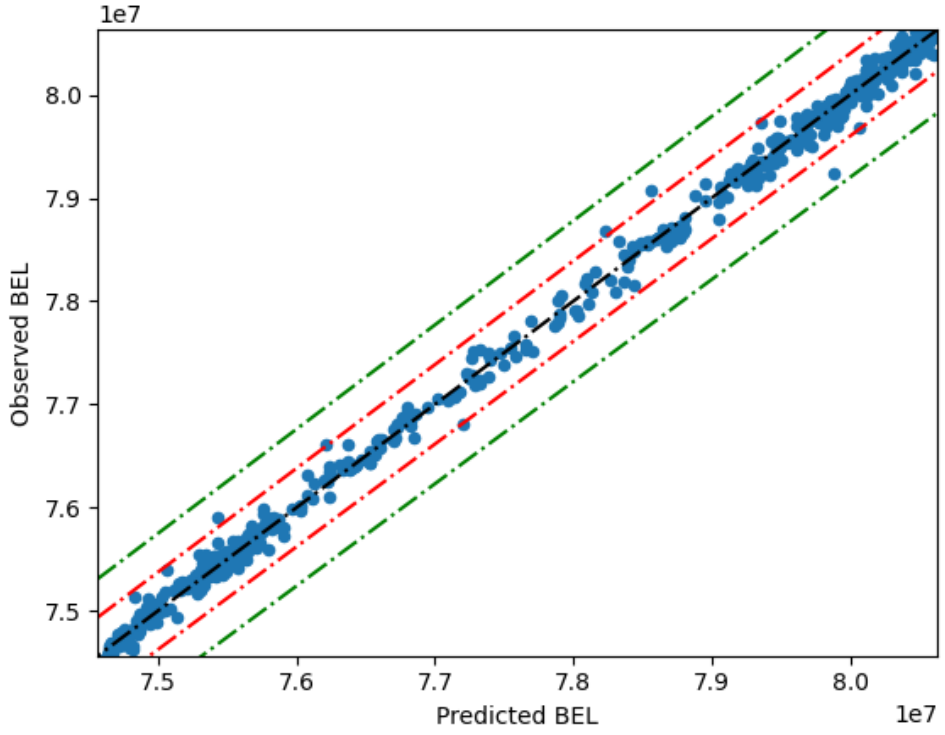


Figure 5.17: Scatter plot of the BEL predictions of the model trained on a training set of size 650. The management and reporting requirements are depicted as red and green lines, respectively. Every point corresponds to a predicted BEL as well as the observed BEL. Almost all point are within the management and reporting requirements. Similar plots can be made for SCR mortality, longevity and catastrophe.

6

Discussion

Insurers are required to have sufficient buffers to be able to meet financial obligations that result from their portfolios. The minimal value of these buffers can be determined using one of two methods. Insurers can use the standard model and one is allowed to create an internal model as discussed in Chapter 2. An internal model can be made more specific for the current situation of the insurer. Creating an internal model is, however, difficult due to all the regulations of the European Insurance and Occupational Pensions Authority (EIOPA). Hence, applying the prescribed standard model is a choice often made by insurers. The standard model becomes computationally heavy as a portfolio increases in size, resulting in a calculation time om tens of minutes. The standard model calculates with only two mortality tables, one for the male policy holders and one for the female policy holders. Hence, for every combination of forecast mortality tables is only one possible future cash flow obtained. More combinations of mortality tables are required get a reliable estimate of the future cash flow and a corresponding variance. Computing the cash flow for multiple combination of tables results in a even more computationally heavy and above all computationally time consuming process. Computing the standard model for thousand different combination of mortality tables on a portfolio of size one hundred thousand takes roughly five whole days.

In this report, the possibility to use XGBoost as an alternative to determining all cash flows is explored. Instead of considering a complete cash flow, only the parts dependent on a mortality table are considered, given as the best estimate liability (BEL) and the solvency capital requirements (SCR) for mortality, longevity and catastrophe risk. Forecast mortality tables are needed to determine future values of the BEL and SCRs. Mortality rates are forecast using the Lee-Carter model calibrated on historical mortality data. The forecast mortality tables are used to determine the corresponding BEL and SCR. The combination of this data is in turn used as input of an XGBoost model. Two different modelling approaches of XGBoost are compared. First an individual model is trained on each target variable. Secondly, a combined model is trained, which trains on all target variables at the same time. This second approach takes possible dependence between target variables into account. The main reason to consider machine learning is the reduction of computation time. As the most time is lost by creating a training set, one tries to minimize the number of training samples.

In Chapter 5 the size of the training set is taken as a variable and for each size is the root mean squared error (RMSE) determined. The increase in performance of the individual models stagnates around a training size of 650. Similar behaviour of the combined model is observed, in which again around 650 the performance increase stagnates. The combined model approach, however, attains a better performance. This is especially the case for smaller training sizes. Hence, the combined approach is taken as the better performing model. If one is, however, only interested in only one target variable at a time, creating one individual model is faster. The performance of

one such an individual model is almost identical to the combined model for only one target.

Multiple requirements need to be satisfied in order to call the XGBoost model application a success. First, the management requirement state that in 99.5% of the cases, the prediction should be within 1% of the real value. Similar, the reporting requirements state that in 99.5% of the cases, the prediction should deviate within 0.5% of the real value. The management requirements are already satisfied for a training set of size 650 as seen in Chapter 5. This model takes only 30 seconds in which the model is trained as well as the predictions on the remaining data are made. Hence, with a training size of only 650 observations, thousands of cash flows can be predicted. By training of 650 observations and predicting 650 instances, the calculation time is decreased by roughly 50%, as (only) 650 cash flows are determined. Insurers could extend the instances to predict even further to for example 6500, in which the training set still consists of 650 observations. In this case a time reduction of 90% is achieved while still satisfying the management requirements. Hence, by introducing more instances to predict, a higher time reduction is achieved. This is a huge decrease and is worth considering for insurers.

In order to satisfy the reporting requirements for the combined model, one has to use a training set of at least size 1750 as seen in Chapter 5. This is almost three times a large as for management requirements which means creating the training set takes three times as long. This still results in a huge decrease in computation time for a large number of instances to predict. Determining the cash flow for reporting purposes is happening at lower frequencies than management purposes, for example yearly against monthly. Because of this, the higher number of standard model calculations is partly justified and acceptable.

To conclude, one can conclude that XGBoost in combination with Lee-Carter simulations results in good approximations of the cash flows while greatly reducing the computation time. Based on the higher number of predictions obtained this way, a better estimate of the future cash flow can be formed.

6.1. Recommendations for future research

In order to be usable for reporting of insurers, this model has to be approved by the regulatory authorities. The regulatory authorities are, however, sceptical of randomness within machine learning. To be used as an alternative to the standard model, one does need more and extensive testing. This research is trained on only one portfolio of a decent size. It is however unclear what would happen if the portfolio becomes twice as big or even bigger. This makes a nice follow up research. One could for example in which one could consider different kinds of machine learning such as a neural network as well.

The data used in this research only considers Dutch population data. In real life applications, one might want to consider immigration and emigration as well as mortality data of neighbouring countries. Based on this data a more reliable mortality table can be created resulting in better estimates. Besides, event such as World War I, World War II and corona should be researched to get an even better mortality trend. Considering all these aspects, the mortality tables created are closer to reality.

Creating a more generic model which works for multiple portfolios would be another nice follow up research. A portfolio can be characterize based on the characteristics of the policy holders. For example, one could create groups based on gender, age, premium, frequency etc. These characteristics can be added as features in the training data. This way a generic model is created, which can predict for all portfolios. This is however an extremely computationally heavy problem as characterizing a portfolio in small enough parts will result in lots of features. Besides, way more training data needs to be available to get a well performing model.

In this research are only the BEL and three SCRs considered, there are however more SCRs defined by the standard model. The models can be extended in such a way that more SCRs

are considered as target variables. This does however require more features, such as equity information. This complicates the model further but also creates more possibilities for research.

The GPU capabilities of XGBoost are only available on histogram feature selection as discussed in Chapter 4. The histogram method is an approximation method introduced to be time efficient, while the exact method is time inefficient but creates a better model. The GPU capabilities of XGBoost can be extended to use for the exact method, which is a research subject on its own.

References

Bentéjac, C., Csörgő, A., & Martınez-Muñoz, G. (2021). A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, *54*, 1937–1967.

Börger, M. (2010). Deterministic shock vs. stochastic value-at-risk an analysis of the solvency II standard model approach to longevity risk. *Blätter der DGVFM*, *31*, 225–259. https://doi.org/10.1007/s11857-010-0125-z

Castellani, G., Fiore, U., Marino, Z., Passalacqua, L., Perla, F., Scognamiglio, S., & Zanetti, P. (2018). An investigation of machine learning approaches in the solvency II valuation framework. *Available at SSRN 3303296*.

CEIOPS. (2009). *CEIOPS' advice for level 2 implementing measures on solvency II: Standard formula SCR - article 109 c life underwriting risk*. https://register.eiopa.europa.eu/CEIOPS-Archive/Documents/Advices/CEIOPS-L2-Final-Advice-on-Standard-Formula-Life-underwriting-risk.pdf

CEIOPS. (2010). CEIOPS' advice for level 2 implementing measures on solvency II: SCR standard formula article 111(d) correlations. *CEIOPS*, 1–57. https://register.eiopa.europa.eu/CEIOPS-Archive/Documents/Advices/CEIOPS-L2-Advice-Correlation-Parameters.pdf

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794. https://doi.org/10.1145/2939672.2939785

Chen, T., Singh, S., Taskar, B., & Guestrin, C. (2015). Efficient Second-Order Gradient Boosting for Conditional Random Fields. In G. Lebanon & S. V. N. Vishwanathan (Eds.), *Proceedings of the eighteenth international conference on artificial intelligence and statistics* (pp. 147–155). PMLR.

DNB. (2022). *Solvency II: General notes*. Retrieved April 22, 2022, from https://www.dnb.nl

Eckart, C., & Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, *1*(3), 211–218.

EIOPA. (2009). *Directive 138/2009/ec (solvency II directive)* [https://www.eiopa.europa.eu/rulebook-categories/directive-1382009ec-solvency-ii-directive_en].

EIOPA. (2021). Insurance stress test 2021 technical specifications. *EIOPA*, 1–39. https://www.eiopa.europa.eu/system/files/2021-05/2021-stress-test-technical-specifications-v1.1.pdf

Fiore, U., Marino, Z., Passalacqua, L., Perla, F., Scognamiglio, S., & Zanetti, P. (2018). Tuning a deep learning network for solvency II: Preliminary results. *Mathematical and Statistical Methods for Actuarial Sciences and Finance: MAF 2018*, 351–355.

Guo, H. (2021). *What are tensors exactly?* World Scientific.

Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (Vol. 2). Springer.

Human Mortality Database, Max Planck Institute for Demographic Research (Germany), University of California Berkeley (USA), & French Institute for Demographic Studies (France). (2022). Retrieved June 8, 2022, from https://www.mortality.org

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning: With applications in R*. Springer.

Kleinow, T., & Richards, S. J. (2017). Parameter risk in time-series mortality forecasts. *Scandinavian Actuarial Journal*, *2017*(9), 804–828. https://doi.org/10.1080/03461238.2016.1255655

Lee, R. D., & Carter, L. R. (1992). Modeling and forecasting us mortality. *Journal of the American statistical association*, *87*(419), 659–671.

Mirsky, L. (1960). Symmetric gauge functions and unitarily invariant norms. *The quarterly journal of mathematics*, *11*(1), 50–59.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

XGBoost. (2022). *DMLC XGBoost documentation*. Retrieved May 30, 2023, from https://xgboost.readthedocs.io/en/stable/index.html

A

Code Appendices

A.1. R code generating mortality tables

```
```{r Packages}
library(readr)
library(forecast)
library(openxlsx)
```

```{r Functions R file source}
ApplyLeeCarter <- function(dataFrame){
  logDataFrame <- log(t(dataFrame))
  a <- apply(logDataFrame, 2, mean)
  dfMinusMean <- sweep(logDataFrame, 2, a)
  USV = svd(dfMinusMean)
  sumv <- sum(USV$v[,1])
  b <- USV$v[,1]/sumv
  k <- USV$d[1] * USV$u[,1] * sumv
  return(list(a = a, b = b, k = k))
}

KToArimaModel <- function(k, startYear, plotTitle){
  timeSeries <- ts(k, start = startYear)
  par(mfrow=c(1,2))
  plot(timeSeries, xlab = "Year", ylab ="Mortality Trend k", main = plotTitle)
  plot(diff(timeSeries), xlab="Year", ylab='Differenced Logarithmic Mortality Trend',
      main = plotTitle)
  par(mfrow=c(1,1))
  arimaFit <- auto.arima(timeSeries)
  return(arimaFit)
}

ArimaToMortalityTable <- function(a, b, k){
  return(exp(sweep(outer(k,b), 2, a, "+")))
}

GenerateTables <- function(dataFrame, numberOfSimulations, numberOfYearsForecast, folder
    ){
  LeeCarter <- ApplyLeeCarter(dataFrame)

  #Writing the fit to Excel
  wbFit = createWorkbook()
  addWorksheet(wbFit, "a")
  writeData(wbFit, "a", LeeCarter$a)
  addWorksheet(wbFit, "b")
  writeData(wbFit, "b", LeeCarter$b)
  addWorksheet(wbFit, "k")
```
```

```
  writeData(wbFit, "k", LeeCarter$k)
  saveWorkbook(wbFit, paste(folder, "LeeCarterFit.xlsx", sep = "/"),overwrite = TRUE)

  startYear <- min(colnames(dataFrame))
  endYear <- max(colnames(dataFrame))
  ArimaModel <- KToArimaModel(LeeCarter$k, startYear, endYear)
  Simulations <- replicate(numberOfSimulations, simulate(ArimaModel,
      numberOfYearsForecast))
  for(i in 1:numberOfSimulations){
    ForeCastMortalityTable <- data.frame(t(ArimaToMortalityTable(LeeCarter$a, LeeCarter$
        b, Simulations[,i])))
    colnames(ForeCastMortalityTable) <- seq(as.numeric(endYear)+1, as.numeric(endYear)+
        numberOfYearsForecast)
    rownames(ForeCastMortalityTable) <- rownames(dataFrame)

    #Write table to Excel
    wbTable = createWorkbook()
    addWorksheet(wbTable, "MortalityTable")
    writeDataTable(wbTable, "MortalityTable", ForeCastMortalityTable, rowNames = TRUE)
    addWorksheet(wbTable, "Vector_k")
    writeData(wbTable, "Vector_k", Simulations[,i])
    saveWorkbook(wbTable, paste(folder, paste(toString(i), ... = "ForeCastMortalityTable
        .xlsx", sep = "_"), sep = "/"), overwrite = TRUE)
  }
}

GenerateTablesCombined <- function(dataFrameMale, dataFrameFemale, numberOfSimulations,
    numberOfYearsForecast, folder){
  LeeCarterMale <- ApplyLeeCarter(dataFrameMale)
  LeeCarterFemale <- ApplyLeeCarter(dataFrameFemale)

  #Writing the fit to Excel
  wbFit = createWorkbook()

  #Male fit
  addWorksheet(wbFit, "aMale")
  writeData(wbFit, "aMale", LeeCarterMale$a)
  addWorksheet(wbFit, "bMale")
  writeData(wbFit, "bMale", LeeCarterMale$b)
  addWorksheet(wbFit, "kMale")
  writeData(wbFit, "kMale", LeeCarterMale$k)

  #Female fit
  addWorksheet(wbFit, "aFemale")
  writeData(wbFit, "aFemale", LeeCarterFemale$a)
  addWorksheet(wbFit, "bFemale")
  writeData(wbFit, "bFemale", LeeCarterFemale$b)
  addWorksheet(wbFit, "kFemale")
  writeData(wbFit, "kFemale", LeeCarterFemale$k)

  #Save workbook
  saveWorkbook(wbFit, paste(folder, "LeeCarterFit.xlsx", sep = "/"),overwrite = TRUE)

  startYear <- min(colnames(dataFrameMale))
  endYear <- max(colnames(dataFrameMale))

  #Fitting ARIMA model
  ArimaModelMale <- KToArimaModel(LeeCarterMale$k, startYear, "Male mortality")
  ArimaModelFemale <- KToArimaModel(LeeCarterFemale$k, startYear, "Female mortality")

  #Simulating
  SimulationsMale <- replicate(numberOfSimulations, simulate(ArimaModelMale,
      numberOfYearsForecast))
  SimulationsFemale <- replicate(numberOfSimulations, simulate(ArimaModelFemale,
      numberOfYearsForecast))

  #Create mortality tables
```

```
  for(i in 1:numberOfSimulations){
    ForeCastMortalityTableMale <- data.frame(t(ArimaToMortalityTable(LeeCarterMale$a,
        LeeCarterMale$b, SimulationsMale[,i])))
    ForeCastMortalityTableFemale <- data.frame(t(ArimaToMortalityTable(LeeCarterFemale$a
        , LeeCarterFemale$b, SimulationsFemale[,i])))
    colnames(ForeCastMortalityTableMale) <- seq(as.numeric(endYear)+1, as.numeric(
        endYear)+numberOfYearsForecast)
    colnames(ForeCastMortalityTableFemale) <- seq(as.numeric(endYear)+1, as.numeric(
        endYear)+numberOfYearsForecast)
    rownames(ForeCastMortalityTableMale) <- rownames(dataFrameMale)
    rownames(ForeCastMortalityTableFemale) <- rownames(dataFrameFemale)

    #Write table and k to Excel
    wbTable = createWorkbook()
    addWorksheet(wbTable, "MortalityTableMale")
    writeDataTable(wbTable, "MortalityTableMale", ForeCastMortalityTableMale, rowNames =
         TRUE)
    addWorksheet(wbTable, "MortalityTableFemale")
    writeDataTable(wbTable, "MortalityTableFemale", ForeCastMortalityTableFemale,
        rowNames = TRUE)
    addWorksheet(wbTable, "Vector_k_Male")
    writeData(wbTable, "Vector_k_Male", SimulationsMale[,i])
    addWorksheet(wbTable, "Vector_k_Female")
    writeData(wbTable, "Vector_k_Female", SimulationsFemale[,i])

    #Save workbook
    saveWorkbook(wbTable, paste(folder, paste(toString(i), ... = "ForeCastMortalityTable
        .xlsx", sep = "_"), sep = "/"), overwrite = TRUE)
  }
}
```

```{r Importing data}
X210722_DeathRatesNL <- read_table("210722_DeathRatesNL.txt", skip = 2)
X210722_DeathRatesNL[X210722_DeathRatesNL == "110+"] <- "110" #Set equal to one for age
    over 100
X210722_DeathRatesNL[X210722_DeathRatesNL == "."] <- "1" #Some data is undefined and is
    set equal to one
X210722_DeathRatesNL[X210722_DeathRatesNL == "0.000000"] <- "0.000001" #Some data is
    undefined and is set equal to one

ReadDeathRates <- function(txtData){
  years <- max(txtData$Year) - min(txtData$Year) + 1
  ages <- 120
  dfMale <- data.frame(matrix(ncol = years, nrow = ages + 1))
  dfFemale <- data.frame(matrix(ncol = years, nrow = ages + 1))
  columnNames <- unique(txtData$Year)
  colnames(dfMale) <- columnNames
  colnames(dfFemale) <- columnNames
  rowNames <- 0:ages
  rownames(dfMale) <- rowNames
  rownames(dfFemale) <- rowNames
  for(i in 1:nrow(txtData)){
      rowData <- txtData[i,]
      dfMale[toString(rowData$Age), toString(rowData$Year)] = as.numeric(rowData$Male)
      dfFemale[toString(rowData$Age), toString(rowData$Year)] = as.numeric(rowData$
          Female)
  }

  #Adding data for ages above 110 equal to rate at age 110
  dfMale[is.na(dfMale)] <- 1
  dfFemale[is.na(dfFemale)] <- 1

  return(list(Male = dfMale, Female = dfFemale))
}

DeathRatesNL <- ReadDeathRates(X210722_DeathRatesNL)
```

```
DeathRatesMale <- DeathRatesNL$Male
DeathRatesFemale <- DeathRatesNL$Female
```

```{r Generate Tables}
set.seed(468)

GenerateTablesCombined(DeathRatesMale, DeathRatesFemale, 1000, 172, "ForecastMortalityTables")
```

A.2. Python Code Standard XGBoost

```
import pandas as pd
import xgboost as xgb
import matplotlib.pyplot as plt
import numpy as np
from typing import Dict, Tuple, List
from sklearn.model_selection import GridSearchCV
import time

np.random.seed(4679)
dfData = pd.read_excel(dataFileName)

def StandardModel(feature, value, featureValidation, valueValidation):
    X, y = feature, value
    Xval, yval = featureValidation, valueValidation
    params = {
            "tree_method": "hist",
            "num_target": y.shape[1],
            "gamma": "50",
            "max_depth": "4",
            "lambda":"1",
            "eta":"0.02",
            "nthread":"-1"
        }
    reg = xgb.XGBRegressor(tree_method="hist", n_estimators=900, early_stopping_rounds = 25, gamma= 50, eta = 0.02, max_depth = 4, reg_lambda = 1, verbosity=0)
    reg.fit(X, y, eval_set=[(Xval, yval)])
    return reg

def RMSE(observed, predicted):
    return np.sqrt(np.sum(np.power((observed - predicted),2)))

def RequirementCheck(observed, predicted, requirement):
    accuracy = np.divide(abs((observed - predicted)), observed)
    return len([i for i in accuracy if i <= requirement])/len(observed)

def DeviateCheck(observed, predicted):
    accuracy = np.divide((observed - predicted), observed)
    return accuracy

def HowManyWithinRequirement(observed, predicted, requirement):
    accuracy = np.divide(abs((observed - predicted)), observed)
    return len([i for i in accuracy if i <= requirement])

def RequirementCheckCombined(observed, predicted, requirement):
    accuracy = np.zeros((4,len(observed)))
    for i in range(4):
        accuracy[i] = np.divide(abs((observed.iloc[:,i] - predicted[:,i])), observed.iloc[:,i])
    combinedAccuracy = np.max(accuracy, axis = 0)
    return len([i for i in combinedAccuracy if i <= requirement])/len(observed)

def TrainPerModelAllTestSizes(stepsize: int, totalTrainIndices: list, testIndices: list, valdiationIndices: list, printPlots: bool):
```

```
    numberOfTargetVariables = 4
    numberOfTrain = len(totalTrainIndices)
    numberOfTest = len(testIndices)
    length = int(np.floor(numberOfTrain/stepsize))
    rmse = np.empty(shape = (numberOfTargetVariables,length))
    managementrequirement = np.empty(shape = (numberOfTargetVariables,length))
    managementrequirementCombined = np.empty(length)
    reportingrequirement = np.empty(shape = (numberOfTargetVariables,length))
    reportingrequirementCombined = np.empty(length)
    requirement5 = np.empty(shape = (numberOfTargetVariables,length))
    requirementCombined5 = np.empty(length)
    requirement2 = np.empty(shape = (numberOfTargetVariables,length))
    requirementCombined2 = np.empty(length)
    requirement01 = np.empty(shape = (numberOfTargetVariables,length))
    requirementCombined01 = np.empty(length)
    trainRMSE = np.empty(shape = (numberOfTargetVariables,length))
    testRMSE = np.empty(shape = (numberOfTargetVariables,length))
    prediction = np.empty(shape = (numberOfTargetVariables,numberOfTest))
    deviations = np.empty(shape = (numberOfTargetVariables, length, numberOfTest))
    startTime = time.time()
    j=0
    for i in range(stepsize,numberOfTrain + 1,stepsize):
        j+=1
        trainIndices = totalTrainIndices[0:i]
        train = dfData.iloc[trainIndices]
        test = dfData.iloc[testIndices]
        validation = dfData.iloc[validationIndices]
        print("Starting iteration with training size " + str(len(train)))
        partFeaturesTrain = train.iloc[:,:-4]
        partFeaturesTest = test.iloc[:,:-4]
        partFeaturesValidation = validation.iloc[:,:-4]
        partValuesTestCombinded = test.iloc[:,-4:]
        for index in range(4):
            partValuesTrain = train.iloc[:,-4 + index]
            partValuesTest = test.iloc[:,-4 + index]
            partValuesValidation = test.iloc[:,-4 + index]
            model, results = custom_model(partFeaturesTrain, partValuesTrain,
                partFeaturesValidation, partValuesValidation)
            prediction[index] = model.predict(partFeaturesTest)
            rmse[index][j-1] = RMSE(partValuesTest, prediction[index])
            trainRMSE[index][j-1] = results["Train"]["rmse"][-1]
            testRMSE[index][j-1] = results["Test"]["rmse"][-1]
            managementrequirement[index][j-1] = RequirementCheck(partValuesTest,
                prediction[index], 0.01)
            reportingrequirement[index][j-1] = RequirementCheck(partValuesTest,
                prediction[index], 0.005)
            requirement5[index][j-1] = RequirementCheck(partValuesTest, prediction[index
                ], 0.05)
            requirement2[index][j-1] = RequirementCheck(partValuesTest, prediction[index
                ], 0.02)
            requirement01[index][j-1] = RequirementCheck(partValuesTest, prediction[
                index], 0.001)
            deviations[index][j-1] = DeviateCheck(partValuesTest, prediction[index])
        reportingrequirementCombined[j-1] = RequirementCheckCombined(
            partValuesTestCombinded, prediction.T, 0.005)
        managementrequirementCombined[j-1] = RequirementCheckCombined(
            partValuesTestCombinded, prediction.T, 0.01)
        requirementCombined5[j-1] = RequirementCheckCombined(partValuesTestCombinded,
            prediction.T, 0.05)
        requirementCombined2[j-1] = RequirementCheckCombined(partValuesTestCombinded,
            prediction.T, 0.02)
        requirementCombined01[j-1] = RequirementCheckCombined(partValuesTestCombinded,
            prediction.T, 0.001)
    duration = time.time() - startTime
    if(printPlots):
        # Code to generate the plots
    print(duration)
```

```
    return rmse, managementrequirementCombined, reportingrequirementCombined, trainRMSE,
         testRMSE, requirementCombined5, requirementCombined2, requirementCombined01,
        reportingrequirement, managementrequirement, requirement5, requirement2,
        requirement01, deviations

numberOfTrain = 1000
numberOfTest = 200
numberOfValidation = 200
stepsize = 10
trainIndices = list(range(numberOfTrain))
testIndices = list(range(numberOfTrain, numberOfTrain + numberOfTest))
valiationIndices = list(range(numberofTrain + numberOfTest, numberofTrain + numberOfTest
     + numberOfValidation))

rmse, managementrequirementCombined, reportingrequirementCombined, trainRMSE, testRMSE,
    requirementCombined5, requirementCombined2, requirementCombined01,
    managementrequirement, reportingrequirement, requirement5, requirement2,
    requirement01, deviations \
    =TrainPerModelAllTestSizes(stepsize=stepsize, totalTrainIndices=trainIndices,
        testIndices=testIndices, validationIndices=validationIndices, printPlots=True)
```

A.3. Python code Multi Output

```
import pandas as pd
import xgboost as xgb
import matplotlib.pyplot as plt
import numpy as np
from typing import Dict, Tuple, List
import time

np.random.seed(467)
dfData = pd.read_excel(dataFileName)

def multi_model(feature, value, featureValidation, valueValidation):
    def rmse(prediction: np.ndarray, observed: xgb.DMatrix):
        y = observed.get_label().reshape(prediction.shape)
        return np.sqrt(np.mean(np.power(prediction - y,2)))

    def gradient(prediction: np.ndarray, observed: xgb.DMatrix):
        """Gradient squared error in multiple dimensions"""
        y = observed.get_label().reshape(prediction.shape)
        g = (prediction - y).reshape(prediction.size)
        return g

    def hessian(prediction: np.ndarray, observed: xgb.DMatrix):
        """Hessian squared error in multiple dimensions"""
        h = np.ones(prediction.shape).reshape(prediction.size)
        return  h

    def se_total(predt: np.ndarray, dtrain: xgb.DMatrix ):
        grad = gradient(predt, dtrain)
        hess = hessian(predt, dtrain)
        return grad, hess

    def rmse_output(prediction: np.ndarray, observed: xgb.DMatrix):
        v = rmse(prediction, observed)
        return "rmse", v

    X, y = feature, value
    Xy = xgb.DMatrix(X, y)
    XyTest = xgb.DMatrix(featureValidation, valueValidation)
    results: Dict[str, Dict[str, List[float]]] = {}
    reg = xgb.train(
        {
            "tree_method": "hist",
            "num_target": y.shape[1],
```

```
            "gamma": "50",
            "max_depth": "4",
            "lambda":"1",
            "eta":"0.02",
            "nthread":"-1"
        },
        early_stopping_rounds = 25,
        dtrain=Xy,
        num_boost_round=1000,
        obj=se_total,
        evals=[(Xy, "Train"),(XyTest, "Test")],
        evals_result=results,
        custom_metric=rmse_output,
        verbose_eval=False,
        multi_strategy = "multi_output_tree",
    )
    return reg, results

def RMSE(observed, predicted):
    return np.sqrt(np.sum(np.power((observed - predicted),2)))

def RequirementCheck(observed, predicted, requirement):
    accuracy = np.divide(abs((observed - predicted)), observed)
    return len([i for i in accuracy if i <= requirement])/len(observed)

def DeviateCheck(observed, predicted):
    accuracy = np.divide((observed - predicted), observed)
    return accuracy

def HowManyWithinRequirement(observed, predicted, requirement):
    accuracy = np.divide(abs((observed - predicted)), observed)
    return len([i for i in accuracy if i <= requirement])

def RequirementCheckCombined(observed, predicted, requirement):
    accuracy = np.zeros((4,len(observed)))
    for i in range(4):
        accuracy[i] = np.divide(abs((observed.iloc[:,i] - predicted[:,i])), observed.
            iloc[:,i])
    combinedAccuracy = np.max(accuracy, axis = 0)
    return len([i for i in combinedAccuracy if i <= requirement])/len(observed)

def RunAllTestSizes(stepsize: int, totalTrainIndices: list, testIndices: list,
    validationIndices:list, printPlots: bool):
    numberOfTargetVariables = 4
    numberOfTest = len(testIndices)
    numberOfTrain = len(totalTrainIndices)
    numberOfValdiation = len(validationIndices)
    length = int(np.floor(numberOfTrain/stepsize))
    rmseBEL = np.empty(length)
    rmseMort = np.empty(length)
    rmseLong = np.empty(length)
    rmseCat = np.empty(length)
    managementrequirement =  np.empty(shape = (numberOfTargetVariables,length))
    managementrequirementCombined = np.empty(length)
    reportingrequirement =  np.empty(shape = (numberOfTargetVariables,length))
    reportingrequirementCombined = np.empty(length)
    requirement5 = np.empty(shape = (numberOfTargetVariables,length))
    requirementCombined5 = np.empty(length)
    requirement2 = np.empty(shape = (numberOfTargetVariables,length))
    requirementCombined2 = np.empty(length)
    requirement01 = np.empty(shape = (numberOfTargetVariables,length))
    requirementCombined01 = np.empty(length)
    numManagementrequirementBEL = np.empty(length)
    numManagementrequirementMort = np.empty(length)
    numManagementrequirementLong = np.empty(length)
    numManagementrequirementCat = np.empty(length)
    numReportingrequirementBEL = np.empty(length)
```

```
    numReportingrequirementMort = np.empty(length)
    numReportingrequirementLong = np.empty(length)
    numReportingrequirementCat = np.empty(length)
    trainRMSE = np.empty(length)
    testRMSE = np.empty(length)
    deviations = np.empty(shape = (numberOfTargetVariables, length, numberOfTest))
    startTime = time.time()
    j=0
    test = dfData.iloc[testIndices]
    validation = dfData.iloc[validationIndices]
    partFeaturesTest = test.iloc[:,:-4]
    partValuesTest = test.iloc[:,-4:]
    partFeaturesValidation = valdiation.iloc[:,:-4]
    partValuesValidation = validation.iloc[:,-4:]
    for i in range(stepsize,numberOfTrain + 1,stepsize):
        j+=1
        trainIndices = totalTrainIndices[0:i]
        train = dfData.iloc[trainIndices]
        print("Starting␣iteration␣with␣training␣size␣" + str(len(train)))
        partFeaturesTrain = train.iloc[:,:-4]
        partValuesTrain = train.iloc[:,-4:]
        model,results = custom_model(partFeaturesTrain, partValuesTrain,
            partFeaturesTest, partValuesTest)
        prediction = model.inplace_predict(partFeaturesTest)
        rmseBEL[j-1] = RMSE(partValuesTest.iloc[:,0], prediction[:,0])
        rmseMort[j-1] = RMSE(partValuesTest.iloc[:,1], prediction[:,1])
        rmseLong[j-1] = RMSE(partValuesTest.iloc[:,2], prediction[:,2])
        rmseCat[j-1] = RMSE(partValuesTest.iloc[:,3], prediction[:,3])
        trainRMSE[j-1] = results["Train"]["rmse"][-1]
        testRMSE[j-1] = results["Test"]["rmse"][-1]
        managementrequirement[0][j-1] = RequirementCheck(partValuesTest.iloc[:,0],
            prediction[:,0], 0.01)
        managementrequirement[1][j-1] = RequirementCheck(partValuesTest.iloc[:,1],
            prediction[:,1], 0.01)
        managementrequirement[2][j-1] = RequirementCheck(partValuesTest.iloc[:,2],
            prediction[:,2], 0.01)
        managementrequirement[3][j-1] = RequirementCheck(partValuesTest.iloc[:,3],
            prediction[:,3], 0.01)
        managementrequirementCombined[j-1] = RequirementCheckCombined(partValuesTest,
            prediction, 0.01)
        reportingrequirement[0][j-1] = RequirementCheck(partValuesTest.iloc[:,0],
            prediction[:,0], 0.005)
        reportingrequirement[1][j-1] = RequirementCheck(partValuesTest.iloc[:,1],
            prediction[:,1], 0.005)
        reportingrequirement[2][j-1] = RequirementCheck(partValuesTest.iloc[:,2],
            prediction[:,2], 0.005)
        reportingrequirement[3][j-1] = RequirementCheck(partValuesTest.iloc[:,3],
            prediction[:,3], 0.005)
        reportingrequirementCombined[j-1] = RequirementCheckCombined(partValuesTest,
            prediction, 0.005)
        requirement5[0][j-1] = RequirementCheck(partValuesTest.iloc[:,0], prediction
            [:,0], 0.05)
        requirement5[1][j-1] = RequirementCheck(partValuesTest.iloc[:,1], prediction
            [:,1], 0.05)
        requirement5[2][j-1] = RequirementCheck(partValuesTest.iloc[:,2], prediction
            [:,2], 0.05)
        requirement5[3][j-1] = RequirementCheck(partValuesTest.iloc[:,3], prediction
            [:,3], 0.05)
        requirementCombined5[j-1] = RequirementCheckCombined(partValuesTest, prediction,
             0.05)
        requirement2[0][j-1] = RequirementCheck(partValuesTest.iloc[:,0], prediction
            [:,0], 0.02)
        requirement2[1][j-1] = RequirementCheck(partValuesTest.iloc[:,1], prediction
            [:,1], 0.02)
        requirement2[2][j-1] = RequirementCheck(partValuesTest.iloc[:,2], prediction
            [:,2], 0.02)
        requirement2[3][j-1] = RequirementCheck(partValuesTest.iloc[:,3], prediction
```

```
            [:,3], 0.02)
157         requirementCombined2[j-1] = RequirementCheckCombined(partValuesTest, prediction,
                0.02)
158         requirement01[0][j-1] = RequirementCheck(partValuesTest.iloc[:,0], prediction
                [:,0], 0.001)
159         requirement01[1][j-1] = RequirementCheck(partValuesTest.iloc[:,1], prediction
                [:,1], 0.001)
160         requirement01[2][j-1] = RequirementCheck(partValuesTest.iloc[:,2], prediction
                [:,2], 0.001)
161         requirement01[3][j-1] = RequirementCheck(partValuesTest.iloc[:,3], prediction
                [:,3], 0.001)
162         requirementCombined01[j-1] = RequirementCheckCombined(partValuesTest, prediction
                , 0.001)
163         deviations[0][j-1] = DeviateCheck(partValuesTest.iloc[:,0], prediction[:,0])
164         deviations[1][j-1] = DeviateCheck(partValuesTest.iloc[:,1], prediction[:,1])
165         deviations[2][j-1] = DeviateCheck(partValuesTest.iloc[:,2], prediction[:,2])
166         deviations[3][j-1] = DeviateCheck(partValuesTest.iloc[:,3], prediction[:,3])
167         numManagementrequirementBEL[j-1] = HowManyWithinRequirement(partValuesTest.iloc
                [:,0], prediction[:,0], 0.01)
168         numManagementrequirementMort[j-1] = HowManyWithinRequirement(partValuesTest.iloc
                [:,1], prediction[:,1], 0.01)
169         numManagementrequirementLong[j-1] = HowManyWithinRequirement(partValuesTest.iloc
                [:,2], prediction[:,2], 0.01)
170         numManagementrequirementCat[j-1] = HowManyWithinRequirement(partValuesTest.iloc
                [:,3], prediction[:,3], 0.01)
171         numReportingrequirementBEL[j-1] = HowManyWithinRequirement(partValuesTest.iloc
                [:,0], prediction[:,0], 0.005)
172         numReportingrequirementMort[j-1] = HowManyWithinRequirement(partValuesTest.iloc
                [:,1], prediction[:,1], 0.005)
173         numReportingrequirementLong[j-1] = HowManyWithinRequirement(partValuesTest.iloc
                [:,2], prediction[:,2], 0.005)
174         numReportingrequirementCat[j-1] = HowManyWithinRequirement(partValuesTest.iloc
                [:,3], prediction[:,3], 0.005)
175     duration = time.time()-startTime
176     if(printPlots):
177         # Code to generate the plots
178     print(duration)
179     return rmseBEL, rmseMort, rmseLong, rmseCat, managementrequirementCombined,
            reportingrequirementCombined, trainRMSE, testRMSE, requirementCombined5,
            requirementCombined2, requirementCombined01, managementrequirement,
            reportingrequirement, requirement5, requirement2, requirement01, deviations
180
181 numberOfTrain = 1000
182 numberOfTest = 200
183 numberOfValidation = 200
184 stepsize = 10
185 trainIndices = list(range(numberOfTrain))
186 testIndices = list(range(numberOfTrain, numberOfTrain + numberOfTest))
187 valiationIndices = list(range(numberofTrain + numberOfTest, numberofTrain + numberOfTest
        + numberOfValidation))
188
189 rmse, managementrequirementCombined, reportingrequirementCombined, trainRMSE, testRMSE,
        requirementCombined5, requirementCombined2, requirementCombined01,
        managementrequirement, reportingrequirement, requirement5, requirement2,
        requirement01, deviations
190     =RunAllTestSizes(stepsize=stepsize, totalTrainIndices=trainIndices, testIndices=
            testIndices, validationIndices=validationIndices, printPlots=True)
```

B

Data Appendices

B.1. Mortality data

The following data is retrieved from the human mortality database July 21, 2022 and consists of the mortality rates $M_{x,t}$(Human Mortality Database et al., 2022). The data for higher ages is not always accurate and/or available as can be seen in the bottom of Table B.1. These values are set equal to one in the calibration of the Lee-Carter model.

Year	Age	Female	Male		Year	Age	Female	Male
1850	0	0.203847	0.243150		2019	0	0.003335	0.003956
1850	1	0.064087	0.067497		2019	1	0.000300	0.000286
1850	2	0.032135	0.033367		2019	2	0.000107	0.000090
1850	3	0.019046	0.020285		2019	3	0.000059	0.000145
1850	4	0.013872	0.015508		2019	4	0.000058	0.000089
1850	5	0.011730	0.011918		2019	5	0.000058	0.000099
1850	6	0.009213	0.009483		2019	6	0.000035	0.000066
1850	7	0.007921	0.008106		2019	7	0.000068	0.000053
1850	8	0.007204	0.007176		2019	8	0.000044	0.000021
1850	9	0.006389	0.006047		2019	9	0.000065	0.000031
1850	10	0.005727	0.005357		2019	10	0.000054	0.000061
1850	11	0.005209	0.004770		2019	11	0.000033	0.000073
1850	12	0.004603	0.004166		2019	12	0.000065	0.000083
1850	13	0.004506	0.003670		2019	13	0.000097	0.000072
1850	14	0.005514	0.004189		2019	14	0.000052	0.000100
1850	15	0.005784	0.004149		2019	15	0.000112	0.000195
1850	16	0.005162	0.003975		2019	16	0.000139	0.000104
1850	17	0.006000	0.005253		2019	17	0.000198	0.000245
1850	18	0.006605	0.007206		2019	18	0.000171	0.000282
1850	19	0.006643	0.007521		2019	19	0.000194	0.000390
1850	20	0.007028	0.008264		2019	20	0.000138	0.000427
1850	21	0.006626	0.009019		2019	21	0.000178	0.000375
1850	22	0.008292	0.009530		2019	22	0.000244	0.000465
1850	23	0.007697	0.009175		2019	23	0.000257	0.000377
1850	24	0.007348	0.008191		2019	24	0.000204	0.000350
1850	25	0.008861	0.009223		2019	25	0.000136	0.000370
1850	26	0.008866	0.008935		2019	26	0.000227	0.000405
1850	27	0.009828	0.009374		2019	27	0.000268	0.000441
1850	28	0.011152	0.009039		2019	28	0.000301	0.000429

1850	29	0.011208	0.009223	2019	29	0.000296	0.000384
1850	30	0.010112	0.008878	2019	30	0.000284	0.000405
1850	31	0.010737	0.008227	2019	31	0.000258	0.000512
1850	32	0.012897	0.009910	2019	32	0.000360	0.000506
1850	33	0.012623	0.010496	2019	33	0.000389	0.000521
1850	34	0.012617	0.009647	2019	34	0.000419	0.000627
1850	35	0.011591	0.009782	2019	35	0.000444	0.000714
1850	36	0.011902	0.010171	2019	36	0.000374	0.000739
1850	37	0.012864	0.010108	2019	37	0.000368	0.000607
1850	38	0.014377	0.011823	2019	38	0.000520	0.000850
1850	39	0.013879	0.012175	2019	39	0.000512	0.000850
1850	40	0.013901	0.011914	2019	40	0.000725	0.000875
1850	41	0.013732	0.013014	2019	41	0.000635	0.001134
1850	42	0.014632	0.014421	2019	42	0.000761	0.001142
1850	43	0.013325	0.013737	2019	43	0.000807	0.001238
1850	44	0.011980	0.013943	2019	44	0.000879	0.001283
1850	45	0.013888	0.015531	2019	45	0.001035	0.001406
1850	46	0.013476	0.015329	2019	46	0.001222	0.001450
1850	47	0.013500	0.015137	2019	47	0.001261	0.001706
1850	48	0.014054	0.017326	2019	48	0.001178	0.001772
1850	49	0.013730	0.015625	2019	49	0.001568	0.002160
1850	50	0.013711	0.018107	2019	50	0.001628	0.002329
1850	51	0.015389	0.019265	2019	51	0.001981	0.002636
1850	52	0.017882	0.023255	2019	52	0.002319	0.002743
1850	53	0.017307	0.020869	2019	53	0.002405	0.003175
1850	54	0.017747	0.023529	2019	54	0.002540	0.003287
1850	55	0.018074	0.023407	2019	55	0.002973	0.003896
1850	56	0.021458	0.024708	2019	56	0.003261	0.004249
1850	57	0.021944	0.025659	2019	57	0.003407	0.004998
1850	58	0.022429	0.029368	2019	58	0.004084	0.005290
1850	59	0.023779	0.029192	2019	59	0.004434	0.006059
1850	60	0.026211	0.033625	2019	60	0.004842	0.006607
1850	61	0.026767	0.035299	2019	61	0.005491	0.007072
1850	62	0.031705	0.035348	2019	62	0.006056	0.007876
1850	63	0.034722	0.037977	2019	63	0.006507	0.008543
1850	64	0.039809	0.042448	2019	64	0.007179	0.010070
1850	65	0.041059	0.049604	2019	65	0.007835	0.011093
1850	66	0.045389	0.051472	2019	66	0.008180	0.011870
1850	67	0.044407	0.049359	2019	67	0.009150	0.012803
1850	68	0.056751	0.055334	2019	68	0.010470	0.014287
1850	69	0.057481	0.060647	2019	69	0.010899	0.016233
1850	70	0.059301	0.060814	2019	70	0.012372	0.017541
1850	71	0.068057	0.066501	2019	71	0.013135	0.018771
1850	72	0.084577	0.075505	2019	72	0.014506	0.021363
1850	73	0.078684	0.088735	2019	73	0.016362	0.023904
1850	74	0.092105	0.096549	2019	74	0.017944	0.026768
1850	75	0.098805	0.103775	2019	75	0.021112	0.031212
1850	76	0.105420	0.107384	2019	76	0.022464	0.033719
1850	77	0.110821	0.117041	2019	77	0.024737	0.037069
1850	78	0.125451	0.126288	2019	78	0.028776	0.041538
1850	79	0.129575	0.137091	2019	79	0.032407	0.048624
1850	80	0.137721	0.164324	2019	80	0.036057	0.056085
1850	81	0.136639	0.159163	2019	81	0.041843	0.060627
1850	82	0.163119	0.178506	2019	82	0.048704	0.069988

1850	83	0.198675	0.189053	2019	83	0.055684	0.079678
1850	84	0.209581	0.205209	2019	84	0.064955	0.091254
1850	85	0.230187	0.220946	2019	85	0.071934	0.106815
1850	86	0.242329	0.245658	2019	86	0.086783	0.117019
1850	87	0.250703	0.271114	2019	87	0.098011	0.139010
1850	88	0.282353	0.254464	2019	88	0.115408	0.156985
1850	89	0.247423	0.314312	2019	89	0.131933	0.174437
1850	90	0.280734	0.264352	2019	90	0.157984	0.197304
1850	91	0.346154	0.196721	2019	91	0.174325	0.221977
1850	92	0.342145	0.412607	2019	92	0.202296	0.242956
1850	93	0.318584	0.526074	2019	93	0.227886	0.285989
1850	94	0.552037	0.434211	2019	94	0.245357	0.300283
1850	95	0.335852	0.366844	2019	95	0.270890	0.322701
1850	96	0.475128	0.461538	2019	96	0.316842	0.378495
1850	97	0.317965	0.382979	2019	97	0.354615	0.379677
1850	98	0.430828	0.461538	2019	98	0.354124	0.435297
1850	99	0.545455	2.000.000	2019	99	0.404258	0.479533
1850	100	1.114.120	0.000000	2019	100	0.472270	0.533858
1850	101	2.445.141	.	2019	101	0.502967	0.580333
1850	102	4.178.571	.	2019	102	0.548145	0.641386
1850	103	.	.	2019	103	0.584441	0.699850
1850	104	.	.	2019	104	0.617820	0.772336
1850	105	.	.	2019	105	0.660964	0.911892
1850	106	.	.	2019	106	0.708861	1.385.705
1850	107	.	.	2019	107	0.779246	2.717.839
1850	108	.	.	2019	108	0.901401	5.768.455
1850	109	.	.	2019	109	1.347.922	.
1850	110+	.	.	2019	110+	5.814.632	.

Table B.1: Mortality data obtained from the HMD for year 1850 and 2019