Actuarial Machine Learning - Präsentation

Entscheidungsbaumbasierte Regressionsmodelle zur Risikokapitalmodellierung

S. Brand, T. Cvitan, A. Gaubrich und B. Höfling

Universität zu Köln

Wintersemester 2020/2021

22. Januar 2021

Inhalt

Risikokapitalberechnung unter Solvency II

- Solvency II ist eine EU Richtlinie für Versicherungsunternehmen
- Einer der Schwerpunkte: Solvabilitätsvorschriften zur Eigenmittelausstattung (SCR)
- in Solvency II vorgeschlagene Standardformel ist stark vereinfacht und weist Schwächen bei den Annahman auf
- Versicherungsunternehmen können diese Formel mithilfe eines internen Modells umgehen

SCR

- Erinnerung: SCR entspricht dem Value-at-Risk der Basiseigenmittel eines Vesicherungsunternehmens zu einem Konfidenzniveau von 99,5% über den Zeitraum eines Jahres

$\rightarrow$ benötigen die Verteilung der Eigenmittel am Risikohorizont (1 Jahr)

Wie funktioniert ein internes Modell?

- mit generierten Risikoszenarien (Monte Carlo Simulation) kann die Bilanz am Risikohorizont geschätzt werden
- durch die geschätzten Eigenmittel für jedes Szenario zum Zeitpunkt $t = 1$ erhält man eine Wahrscheinlichkeitsverteilung

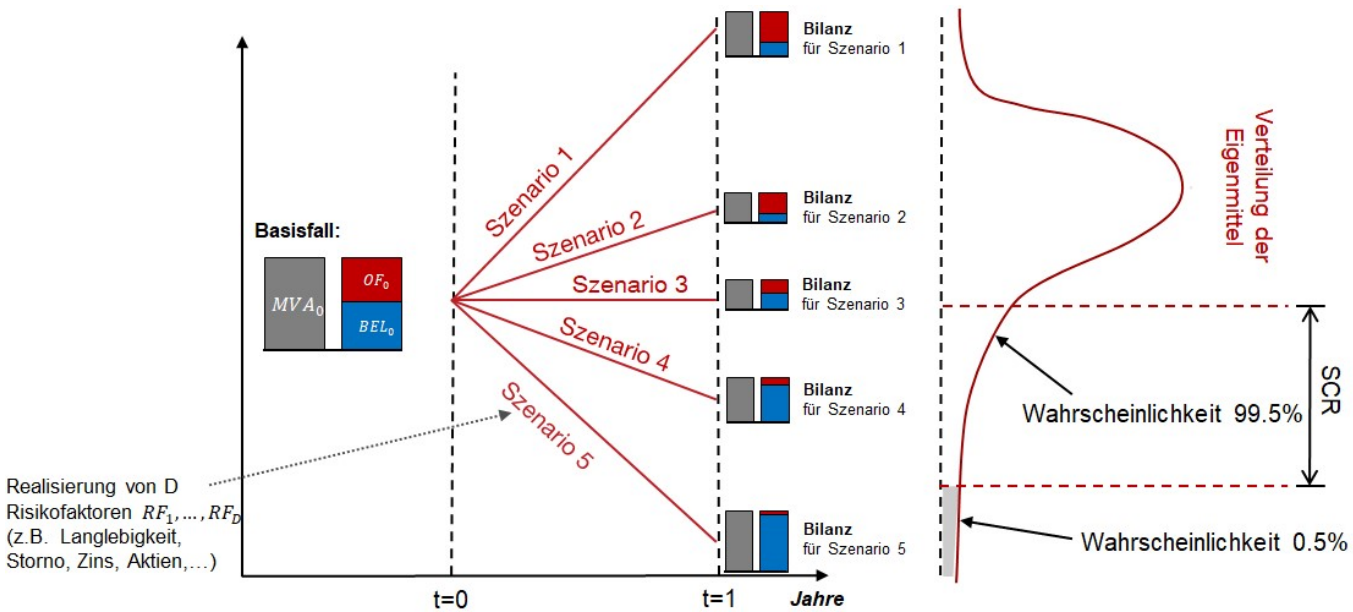


Abbildung: Verteilung der Eigenmittel am Risikohorizont

Nested Stochastics Ansatz

- Bestimmung der Eigenmittel in $t = 1$ Anhand erwarteter zukünftiger Cashflows
- sowohl äußere als auch innere Szenarien durch Monte Carlo simuliert

$\rightarrow$ exponentielle Rechenzeiten

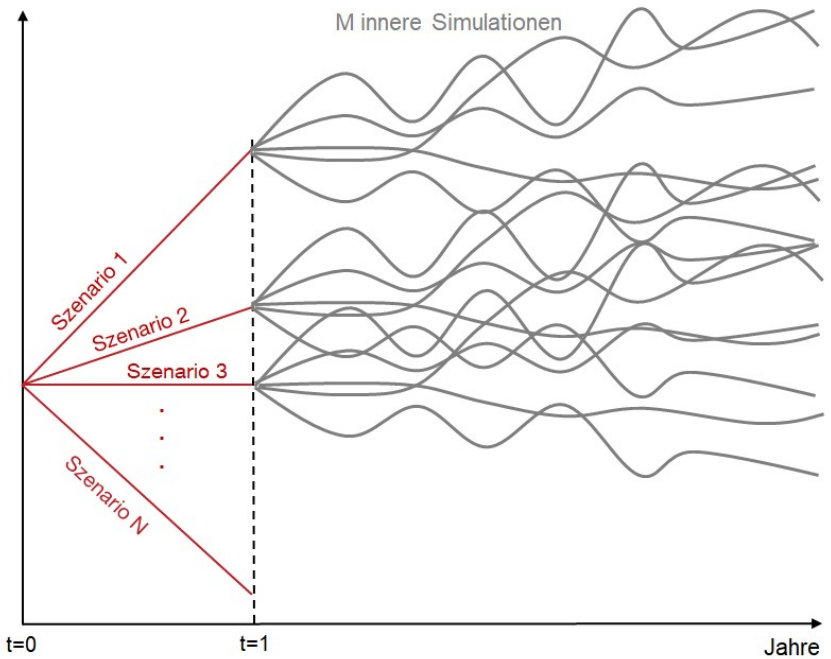


Abbildung: M·N Simulationen (Nested Stochastics)

Lösungsansatz

- Frage: Wie könnte ein effizienteres internes Modell aussehen?
- Idee: Cash Flow Projektkionsmodell könnte durch ein Regressionsmodell ersetzt werden

Lösungsansatz

- Seien $RF_1 \in I_1, \ldots, RF_D \in I_D$ geeignete Risikofaktoren zur Bestimmung der Eigenmittel in $t = 1$
- $I_1 \times \ldots \times I_D$ spannt einen Fittingraum auf
- für $(RF_1, \ldots, RF_D)$ kann mit wenigen inneren Simulationen ein Fittingpunkt PV erzeugt werden
- PV stellt ungenauere Schätzung der Bilanz dar (z.B Eigenmittel)

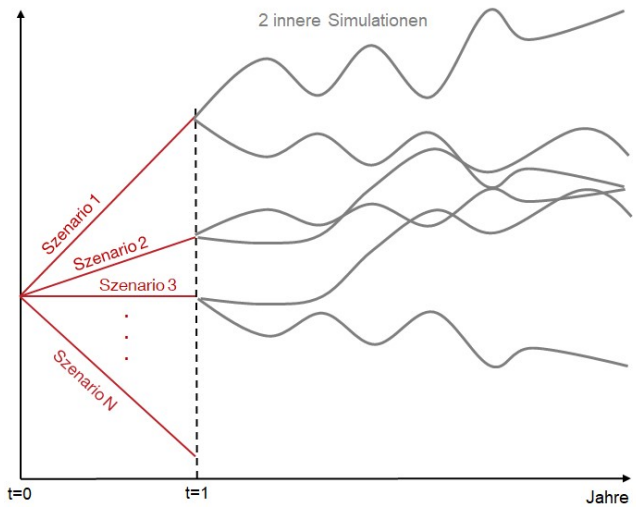


Abbildung: nur zwei innere Szenarien für jedes äußere Szenario

Lösungsansatz

- auf den Punkten $((RF_{1_i}, \ldots, RF_{D_i}), PV_i)$, $i = 1, \ldots, N$ kann durch Regression eine Funktion $f : I_1 \times \ldots \times I_D \longrightarrow \mathbb{R}$ hergeleitet werden, die die Eigenmittel in Abhängigkeit von den Risikofaktoren schätzt

- uns stehen drei Portfolios mit Datensätzen zur Verfügung bestehend aus
  - Trainingsdaten
  - Validierungsdaten
  - Testdaten(nested)
  - Basisfall
  - Standardfehler Validierungsdaten
  - Standardfehler Testdaten

Der Datensatz

	Portfolio 1	Portfolio 2	Portfolio 3
Anzahl Risikotreiber	13	13	12
Anzahl Trainingsdaten	2^{15}	2^{15}	2^{15}
Anzahl Validierungsdaten	256	256	256
Anzahl Testdaten	129	129	50

Trainingsdaten

- Berechnung des Outputs durch zwei antithetische innere Simulationen für jedes äußere Szenario
  - ungenaue Auswertung der Eigenmittel
  - vergleichsweise wenig Rechenaufwand
- gleichmäßige Verteilung der äußeren Szenarien
- auf dieser Menge wird Funktion f hergeleitet, welche Zusammenhang zwischen Risikotreibern und Eigenmitteln beschreibt

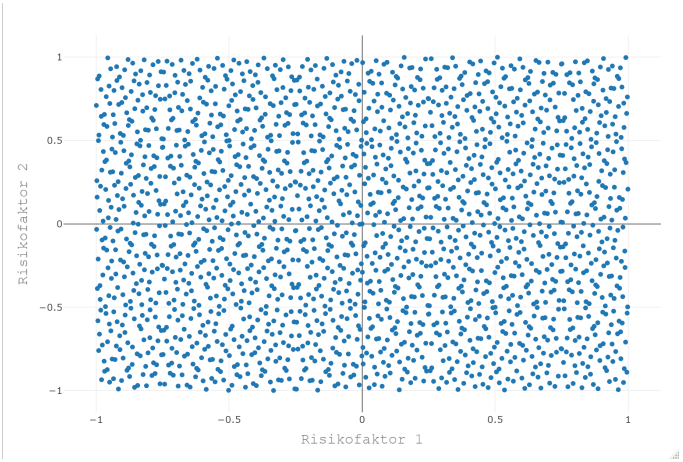


Abbildung: Verteilung der Trainingsdaten für zwei Risikofaktoren (Portfolio 1)

Validierungsdaten

- Berechnung des Outputs durch 1000 innere Simulationen für jedes äußere Szenario
  - genauere Auswertung der Eigenmittel
  - deutlich mehr Rechenaufwand
  - Standardfehler der MC Simulation sind im Datensatz enthalten
- gleichmäßige Verteilung der äußeren Szenarien
- auf dieser Menge wird Funktion f innerhalb des Trainings validiert

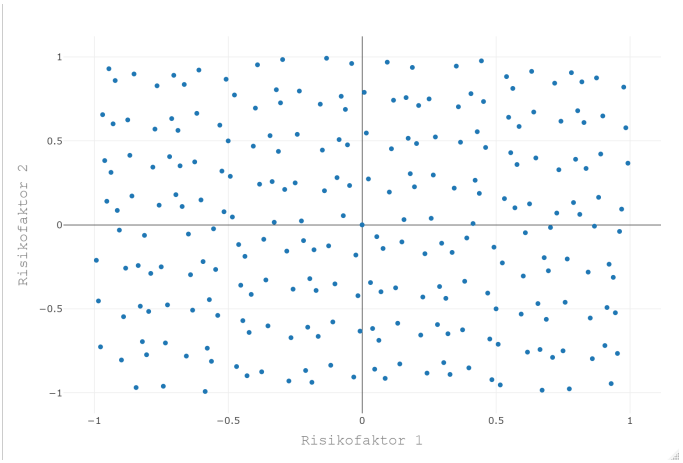


Abbildung: Verteilung der Validierungsdaten für zwei Risikofaktoren (Portfolio 1)

Testdaten

- Berechnung des Outputs durch 4000 innere Simulationen für jedes äußere Szenario
  - genaue Auswertung der Eigenmittel
  - sehr viel Rechenaufwand
  - Standardfehler der MC Simulation sind im Datensatz enthalten
  - geringerer Standardfehler als bei den Validierungsdaten
- Verteilung der Risikofaktoren im Bereich des SCR
  - abhängig von einem vorher gewählten Proxy Ansatz
- auf dieser Menge wird Funktion f nach dem Training entgültig bewertet

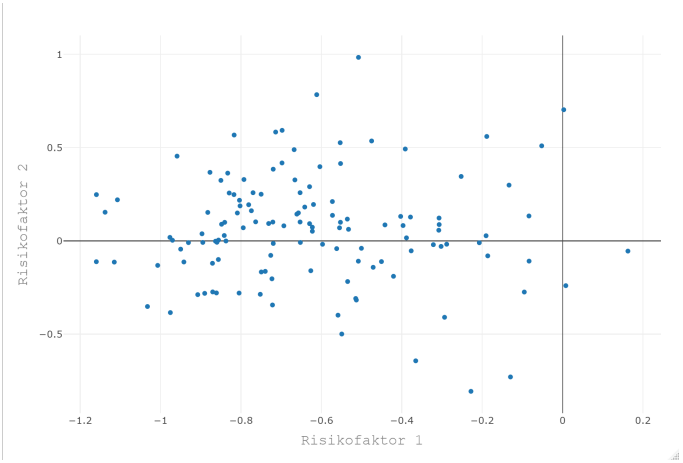


Abbildung: Verteilung der Testdaten für zwei Risikofaktoren (Portfolio 1)

Basisfall

- Referenzpunkt bei Berechnung des SCR
- berechnet mit 16000 inneren Simulationen
- Fittingraum und Outputvariable wurden normiert, sodass Output (Eigenmittel) im Basisfall 1 beträgt
- $((RF_1, \ldots, RF_D), PV)$ mit $RF_1, \ldots, RF_D = 0$ und $PV = 1$
- keine Auslenkung der Risikofaktoren
- kann als Höhe der Eigenmittel zum Zeitpunkt $t = 0$ interpretiert werden
- keine Auslenkung der Risikofaktoren

Vorgehensweise Zusammengefasst

1. Erfassen geeigneter Risikofaktoren
2. Generierung des Datensatzes
3. **Fitten eine Proxy Funktion f auf den Trainingsdaten mittels Regression**
4. **Beurteilung der Approximationsgüte anhand der Validierungsdaten**
5. **Entgültige Bewertung anhand der Testdaten (nested)**

$\longrightarrow$ Wir untersuchen die Punkte 3. bis 5. mit den baumbasierten Regressionsmodellen **Random Forest** und **Gradient Boosting**

Wiederholung Random Forest

- Ensemble aus Entscheidungsbäumen
- fitten von unabhängigen Entscheidungsbäumen f_i $i = 1, \ldots, B$
  - jeder dieser Bäume wird auf einer zufälligen Teilmenge der Trainingsdaten gefitted
  - zufällige Auswahl an Features pro Split
- Endmodell $f = \frac{1}{B} \sum_{i=1}^{B} f_i$

Programmierung in R

- nutzen die Bibliothek *ranger*
- schnelle Implementierung von Random Forest bzw. rekursiver Partitionierung
- mit der Funktion *ranger()* wird ein Random Forest gefitted

```
1  # Laden der Bibliothek ranger
2  library(ranger)
3
4  # fitte ein Random Forest Modell
5  rf.model <- ranger(formula = output~,
6                     data = train,
7                     num.trees = 500
8                     mtry = 3,
9                     min.node.size = 5,
10                    max.depth = 0,
11                    sample.fraction = 1.0
12                    )
```

Abbildung: Beispiel Funktion ranger()

wichtige Parameter

- formula:
- data: Trainingsdaten der Klasse *data.frame*
- num.trees: Anzahl der Bäume
- mtry: Anzahl der möglichen Splitvariablen
- min.node.size: minimale Anzahl der Daten pro Knoten
- max.depth: maximale Tiefe der Bäume
- sample.fraction: Anteil der Daten, die zum fitten der Bäume genutzt werden (bootstraps)

Programmierung in R

- mit der Funktion *predict()* kann ein trainiertes Modell die Outputs einer Inputmenge vorhersagen

```
13
14  prediction <- predict(rf.model, data = new_data)
15
```

Abbildung: Beispiel prediction mit Random Forest

- *prediction* ist ein Objekt des Datentyps Liste, welcher unter anderem die Vorhersagen des Modells enthält
- mit diesen kann ein Fehlermaß wie z.B der Mean Squared Error berechnet werden

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

```
17
18  # Berechnung des MSE
19  mse_new_data <- mean((prediction$predictions - new_data[,"output"])^2)
20
```

Abbildung: Berechnung MSE

Hyperparametersuche

- Fitting des Modells auf den Trainingsdaten
- Bewertung des Modells mittels Fehlermaß auf der Vorhersage der Validierungsdaten
- Welche Kombination der Parameter liefert ein gutes Modell?

$\longrightarrow$ Hyperparametersuche via **Grid Search**

- Überprüfe die Modelle zahlreicher Kombinationen von Parametern auf ihre Güte

Programmierung in R

Code

Ergebnisse Grid Search

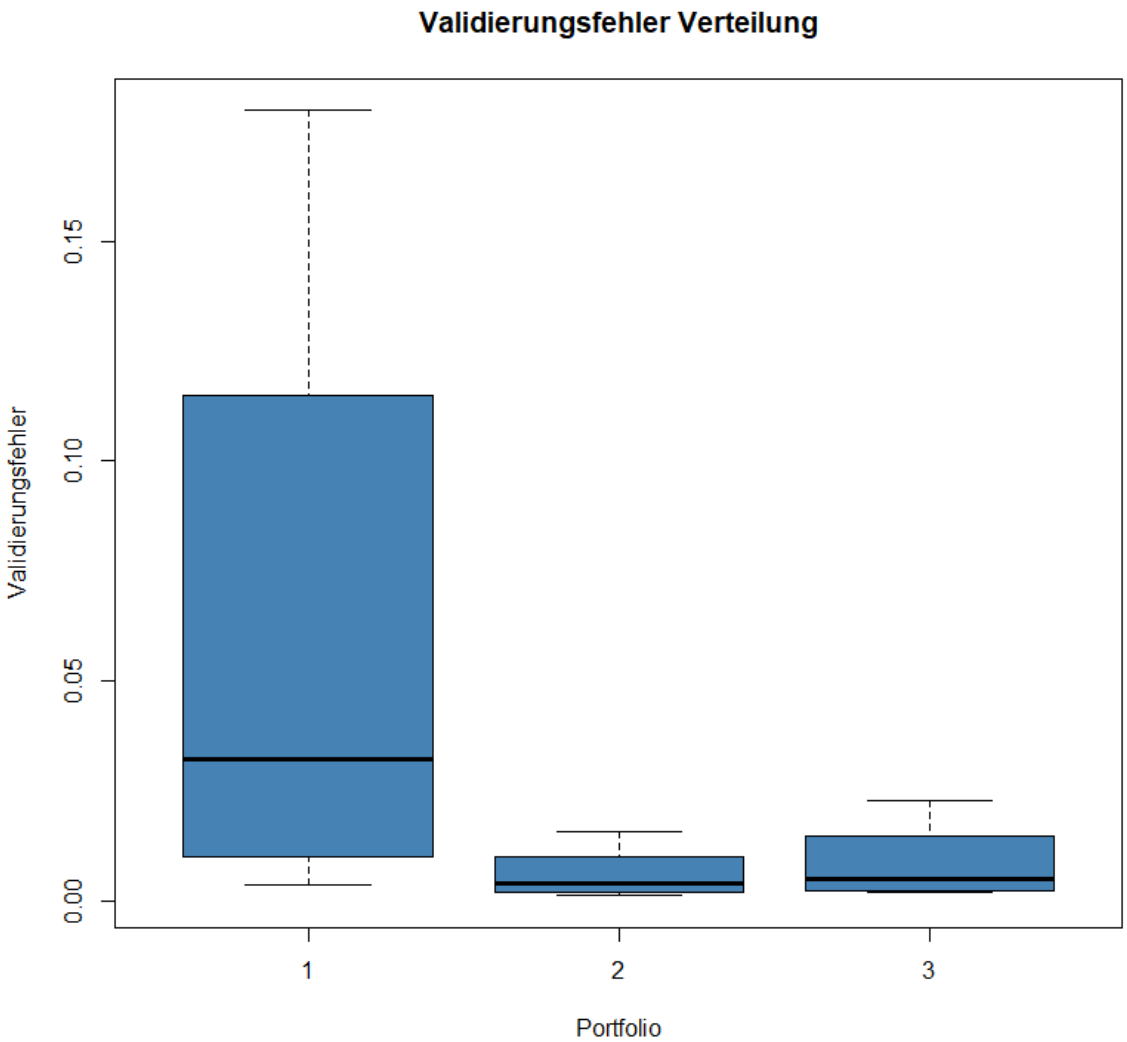


Abbildung: Verteilung der Fehler nach Grid Search

Fehler beste Modelle

	Portfolio 1		Portfolio 2		Portfolio 3	
	MAE	MSE	MAE	MSE	MAE	MSE
train	6.161e-2	7.933e-3	4.998e-2	6.570e-3	1.652e-1	6.992e-2
validation	4.455e-2	3.821e-3	2.403e-2	1.196e-3	3.525e-2	2.061e-3
nested	7.528e-2	7.864e-3	5.133e-2	3.431e-3	5.053e-2	8.950e-3
base	2.520e-2	6.350e-4	3.557e-2	1.265e-3	3.111e-3	9.680e-6

Wiederholung Gradient Boosting

- ebenfalls baumbasierte Ensemblemethode
- Baum wird auf Residuum der vorherigen Iteration gefitted
- einzelne Entscheidungsbäume nicht mehr unabhängig

$$\text{fitte } h_m \text{ auf } y - F_m(x)$$

$$F_{m+1}(x) = F_m(x) + h_m(x)$$

Programmierung in R

- Bibliothek *XGBoost* (extreme Gradient Boosting) beinhaltet effiziente Implementierung von Gradient Boosting
- Funktion *xgb.train()* ermöglicht das Trainieren eines GB Modells
- Parameter werden hier als Liste übergeben

```
# Laden der Bibliothek
library(xgboost)

# Bestimmung der Hyperparameter
parameters = list(eta = 0.3,
                  max_depth = 6,
                  min_child_weight = 1,
                  subsample = 1,
                  colsample_bytree = 1,
                  gamma = 0,
                  lambda = 0,
                  alpha = 0
                  )

# fitte ein xgb Modells
xgb.model <- xgb.train(
                       params = parameters,
                       data = train,
                       nrounds = 5000,
                       objective = "reg:squarederror",
                       )
```

Abbildung: Beispiel xgb.train()

Argumente

- params: Liste der Parameter
  - eta: Lernrate
  - max_depth: maximale Tiefe der Bäume
  - min_child_weight: minimale Anzahl an Daten in einem Knoten
  - subsample: Anteil der Daten, die zur Konstruktion der Bäume verwendet werden (zufällige Auswahl)
  - colsample_bytree: Anteil der Features die zur Konstruktion der Bäume verwendet werden
  - gamma: minimale Fehlerreduktion, die erforderlich ist um einen weiteren Split zu vollziehen
  - lambda und alpha: Regularisierungsparameter
- data: Trainingsdaten der Klasse *xgb.DMatrix*
- objective: spezifiziert das Lernziel (bei uns Regression mit quadratischem Fehler als Loss)

Argumente

- außerdem kann das Argument *watchlist* bestehend aus Trainingsdaten und einer weiteren Datenmenge (z.B Validierung) hinzugefügt werden
- das Modell wird nach jeder Iteration Baum auf diesen Mengen evaluiert
- dies kann in Kombination mit dem Argument *early_stopping_rounds* verwendet werden
- das Training wird gestoppt sobald keine Verringerung des Validierungsfehlers in den nächsten k Iterationen stattfindet

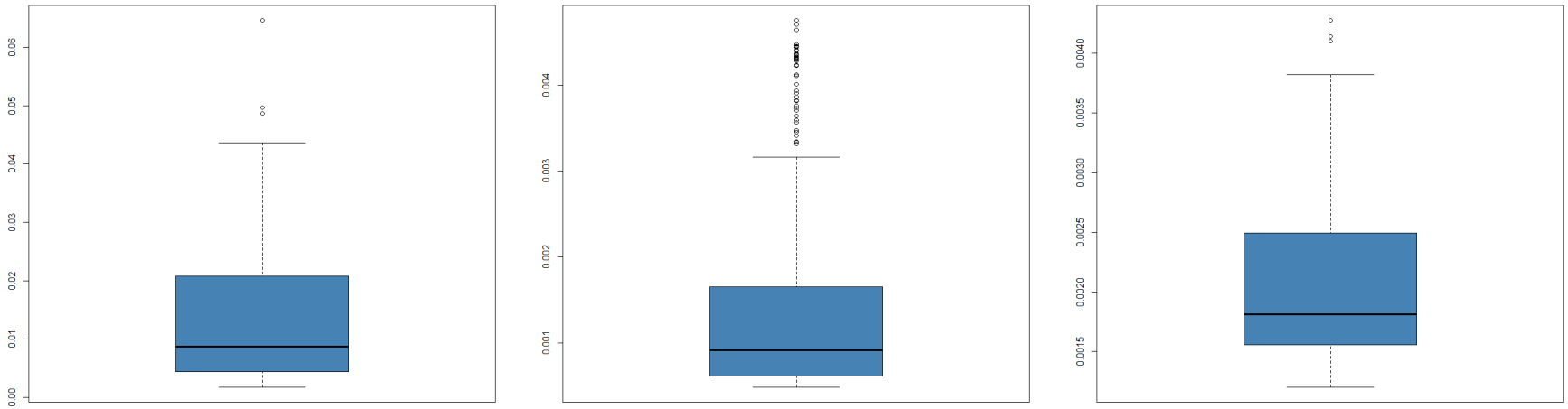
```
15  # fitte ein xgb Modells
16  xgb.model <- xgb.train(
17                         params = parameters,
18                         data = train,
19                         nrounds = 5000,
20                         objective = "reg:squarederror",
21                         early_stopping_rounds = 100,
22                         watchlist = list(train = train,
23                                          validation = validation)
24                         )
```

Abbildung: watchlist und early stopping

Programmierung in R

Code

Ergebnisse Grid Search



a) Portfolio 1 **b)** Portfolio 2 **c)** Portfolio 3

Abbildung: Verteilung des MSE auf der Validierungsmenge nach Grid Search mit XGBoost

Fehler beste Modelle

	Portfolio 1		Portfolio 2		Portfolio 3	
	MAE	MSE	MAE	MSE	MAE	MSE
train	0.0688	0.0104	0.0504	0.0072	0.1659	0.0729
validation	0.0293	0.0015	0.0146	0.0004	0.0236	0.0009
nested	0.06	0.0049	0.0214	0.001	0.0356	0.0067
base	0.0451	0.002	8.068-3	6.509e-5	5.41e-3	2.926e-5

Vergleich mit Random Forest

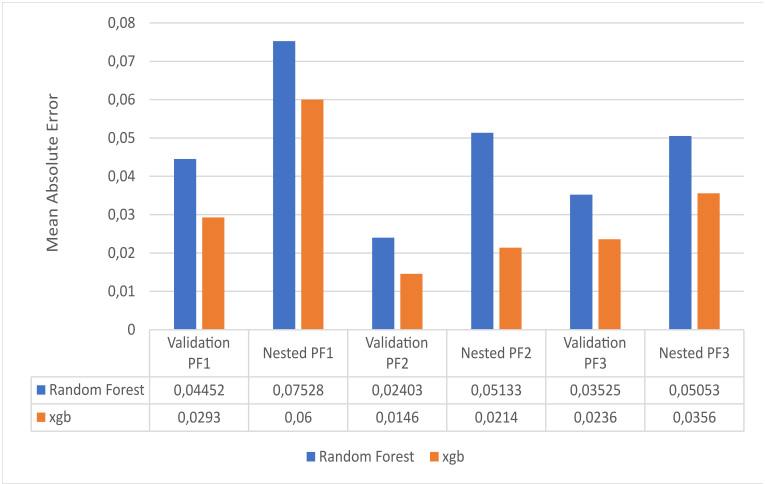


Abbildung: mean absolute error Random Forest vs. XGBoost

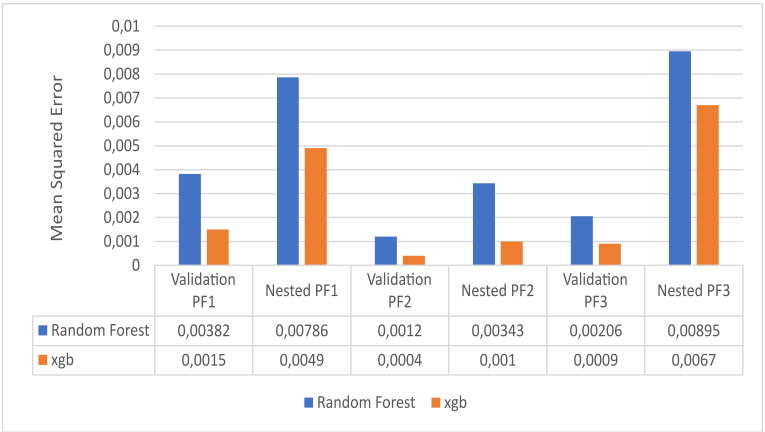


Abbildung: mean squared error Random Forest vs. XGBoost

XGBoost vs. NN Ensemble

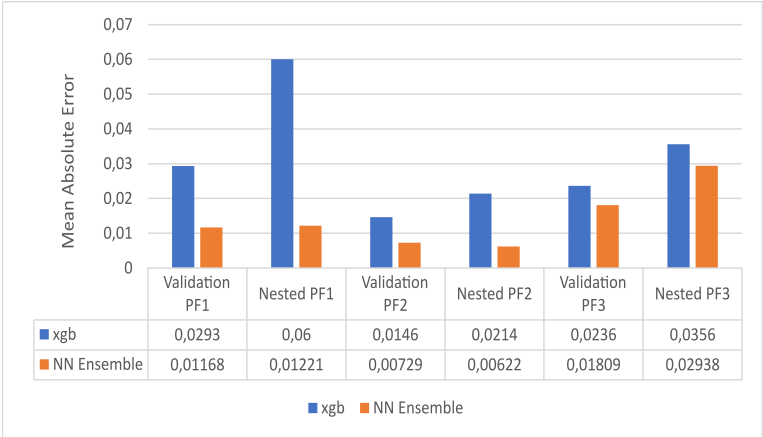


Abbildung: mean absolute error XGBoost vs. NN Ensemble

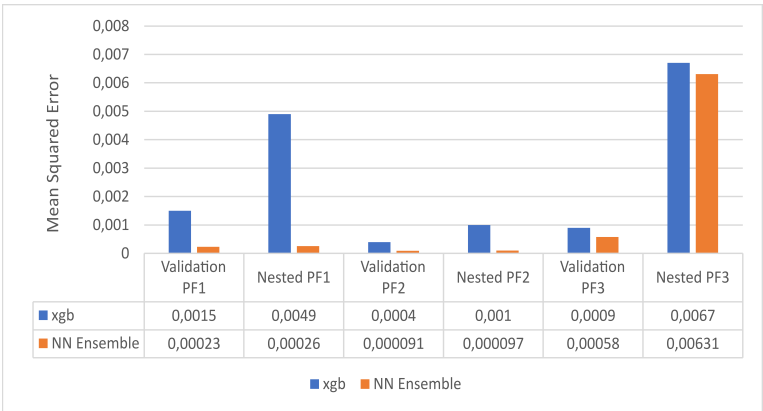


Abbildung: mean squared error XGBoost vs. NN Ensemble

Literatur

- T. Hastie, R. Tibshirani und J. Friedmann , The Elements of Statistical Learning, Springer Verlag (2008)
- G. James, D. Witten, T. Hastie, R. Tibishirani , An Introduction to Statistical Learning, Springer Verlag (2017)
- A. Krah, Z. Nikolic, R. Korn , A Least-Squares Monte Carlo Framework in Proxy Modeling of Life Insurance Companies, Risks (2018)
- DAV, Arbeitsgruppe Statistische Methoden des Ausschusses Actuarial Data Science, Use (this Solvency II) case! Neuronale Netze treffen auf Least Squares Monte Carlo, https://aktuar.de/unsere-themen/bigdata/anwendungsfaelle/Seiten/anwendungsfall2.aspx (2020)