University of Tartu
Faculty of Science and Technology
Institute of Mathematics and Statistics

Linnet Puskar

Estimation of MTPL claim frequency using GLM, GAM and XGBoost techniques

Actuarial and Financial Engineering

Master's thesis (30 ECTS)

Supervisor: Meelis Käärik

Tartu 2020

Liikluskindlustuse kahjusageduse hindamine üldistatud lineaarsete mudelite, üldistatud aditiivsete mudelite ja XGBoosti abil

Magistritöö

Linnet Puskar

Lühikokkuvõte. Selle magistritöö eesmärk on tutvustada XGBoosti algoritmi ning uurida selle sobivust liikluskindlustuse kahjusageduse hindamiseks. Töö esimeses kolmes peatükis tutvustatakse üldistatud lineaarset mudelit, üldistatud aditiivset mudelit ning *gradient boosting*u ja XGBoosti algoritme. Töö neljandas peatükis rakendatakse eelnevalt mainitud meetodeid Eesti Liikluskindlustuse Fondi andmetele, et luua mudel kahjusageduse hindamiseks ning võrreldakse mudeleid.

CERCS teaduseriala: P160 Statistika, operatsioonianalüüs, programmeerimine, finants- ja kindlustusmatemaatika.

Märksõnad: sõidukikindlustus, üldistatud lineaarsed mudelid, tehisõpe, Python (programmeerimiskeel), R (programmeerimiskeel).

Estimation of MTPL claim frequency using GLM, GAM and XGBoost techniques

Master's thesis

Linnet Puskar

Abstract. The purpose of this master's thesis is to provide an overview of the XGBoost algorithm and examine its suitability to model the claim frequency of motor third party liability insurance. The first three chapters introduce generalized linear models, generalized additive models and the algorithms of gradient boosting and XGBoost. In the fourth chapter, the aforementioned methods are applied on the data of Estonian Motor Insurance Bureau to predict claim frequency.

CERCS research specialisation: P160 Statistics, operations research, programming, actuarial mathematics.

Keywords: motor vehicle insurance, generalized linear models, machine learning, Python (programming language), R (programming language).

Contents

Introduction **5**

1 Generalized linear models **7**

1.1 Structure of the model . . . 7
1.2 Link functions . . . 8
1.3 The steps of modelling using GLM . . . 9
1.4 Maximum likelihood estimate . . . 10
1.5 Count data modelling . . . 12

2 Generalized additive model **14**

3 Tree models **16**

3.1 Decision trees . . . 16
3.1.1 Algorithm for growing a regression tree . . . 18
3.1.2 Stopping conditions . . . 19
3.1.3 Classification trees . . . 20
3.1.4 Advantages and disadvantages of decision trees . . . 23
3.1.5 Other tree-building methods . . . 23
3.2 Boosting . . . 24
3.2.1 Adaboost . . . 24
3.2.2 Gradient descent . . . 27
3.2.3 Stochastic gradient descent . . . 29
3.2.4 Gradient boosting . . . 30
3.2.5 XGBoost . . . 31

4 Modelling claim frequency **36**

4.1 Data preprocessing . . . 37
4.2 Modelling with XGBoost . . . 39
4.3 Modelling with GLM . . . 41
4.4 Modelling with GAM . . . 48
4.5 Results . . . 48

Conclusion 50

References 51

Appendices 54

Appendix 1. Description of the dataset 54
Appendix 2. The top feature importances of the first XGBoost model 57
Appendix 3. The feature importances of the second XGBoost model with numeric variables 58
Appendix 4. The output of the first GLM model 59
Appendix 5. The output of the second GLM model with numeric variables . . . 60
Appendix 6. The output of the first GAM model 61
Appendix 7. The output of the second GAM model with numeric variables . . . 62
Appendix 8. The code for preprocessing the data 63
Appendix 9. The code for creating an XGBoost model 80
Appendix 10. The code for creating a GLM and GAM model 87
Appendix 11. The code for calculating the deviances for GLM and GAM 95

Introduction

In order to ask a fair price for a cover, an insurance company must be able to predict how much money will the client cost the company. This question is often answered in two parts - how many claims will the client make and how big the claim will be. This thesis will focus on the first question.

A common approach to predict claim frequency is to use a generalized linear model. However, with machine learning methods gaining more popularity, it became a topic of interest to the author, whether they can be useful in insurance pricing. One popular machine learning method is XGBoost (short for extreme gradient boosting). A well-known machine learning competition website, Kaggle, published 29 of its competitions' winning solutions to its blog in 2015; 17 of those solutions used XGBoost (Chen & Guestrin, 2016).

The purpose of this master's thesis is to provide an overview of the XGBoost algorithm and examine its suitability for modelling the claim frequency of motor third party liability insurance.

The first chapter introduces the structure of generalized linear models and some distributions which are suitable for count modelling. The second chapter gives a short overview of generalized additive models. The third chapter describes decision trees and explains how they are built. Afterward, Adaboost, gradient boosting and XGBoost methods are introduced.

In the fourth chapter, generalized linear models, generalized additive models and XGBoost are used to model motor third party liability insurance claim frequency. Later, the models are compared by using Poisson deviance as a loss function. The dataset used was provided by the Estonian Motor Insurance Bureau. Python programming language was used to preprocess the data and to create an XGBoost model. Generalized linear models and generalized additive models were implemented with R programming language. The thesis is written in LaTeX typesetting system (Lamport, 1986).

The author would like to thank Estonian Motor Insurance Bureau, especially Andres Piirsalu, Kea Mei and Meelis Arumägi, for providing the data. The author is also extremely grateful for the advice and corrections provided by the supervisor Meelis Käärik.

1 Generalized linear models

A common way to assess the relationship between a variable of interest and explanatory variables, is to use an ordinary linear regression model. One of the assumptions for this model is that the residuals are normally distributed. However, for insurance data, the assumptions of a normal model do not always apply. For example, claim sizes and frequencies are not normally distributed.

A step forward from a linear regression model is a generalized linear model. It is one of the most common choices for modelling insurance data. A generalized linear model is simple to understand and easily interpretable.

1.1 Structure of the model

This subchapter is based on de Jong and Heller, 2008.

A generalized linear model (GLM) is an extension of an ordinary regression model – it assesses the relationship between a response variable Y and explanatory variables $(X_1, \ldots, X_J)$. One of the main differences between the two models is that instead of finding the relationship between the mean of the response and the explanatory variables, it evaluates the relationship between a transformation of a mean and the explanatory variables. Another difference is that GLM does not require the response variable to have a normal distribution, just that the distribution belongs to the exponential family.

A probability distribution belongs to the exponential family, if its density function (or probability mass function) can be expressed as

$$f(y) = \mathrm{c}(y, \phi) \exp \frac{y\theta - \mathrm{q}(\theta)}{\phi},$$

where θ is a parameter and $q(\theta)$ is a differentiable function that depends only on θ; $c(y, \phi)$ is a function, that does not depend on θ, but might depend on y and the dispersion parameter ϕ. It can be shown that the expected value of that distribution is equal to $q'(\theta)$ and variance is equal to $\phi q''(\theta)$. Some of the most known distributions that belong

to the family of exponential distributions are exponential, normal, Poisson, gamma and binomial distribution.

Assuming that the distribution of the response variable Y belongs to the exponential family, a generalized linear model has the following form:

$$g(\mu) = \nu, \quad \nu = \boldsymbol{x}^T\boldsymbol{\beta},$$

where $\boldsymbol{x}$ is a vector of variables $\boldsymbol{x} = (1, X_1, \dots X_J)^T$, J is the number of variables used for modelling, $\boldsymbol{\beta}$ is a vector of unknown coefficients $\boldsymbol{\beta} = (\beta_0, \beta_1, \dots, \beta_J)^T$, μ is the conditional average of the response variable Y: $\mu = E(Y \mid \boldsymbol{x})$ and $g(\mu)$ is a link function.

1.2 Link functions

This subchapter is based on Lindsey, 1997.

A link function regulates how the conditional mean μ is related to the linear predictor ν. The link function is often chosen from a known list of canonical link functions. Link function g is called a canonical link, if $g(\mu) = \theta$. Commonly used links are:

- identity link, which is the canonical link for normal distribution

$$g(\mu) = \mu,$$

- log-link, which is the canonical link for Poisson distribution

$$g(\mu) = \ln \mu,$$

- logit-link, which is the canonical link for binomial distribution

$$g(\mu) = \ln \frac{\mu}{1 - \mu},$$

- power-link, which is the canonical link for gamma distribution, if $p = -1$, and for inverse Gaussian, if $p = -2$

$$g(\mu) = \mu^p.$$

If θ has some constants in it, then those are usually omitted from the canonical link. For example, for inverse Gaussian $\theta = -\frac{1}{2\mu^2}$, but the canonical link is $g(\mu) = \frac{1}{\mu^2}$.

Usually, in case of modelling counts, for example the number of claims, the variable of interest is the occurrence rate $\frac{\mu}{n}$, where n is exposure, instead of the mean μ. For example, with log-link:

$$g\left(\frac{\mu}{n}\right) = \ln\left(\frac{\mu}{n}\right) = \nu.$$

From this, the logarithm of μ can be expressed as

$$\ln \mu = \ln n + \nu.$$

Here $\ln n$ is called an offset.

1.3 The steps of modelling using GLM

This subchapter is based on de Jong and Heller, 2008.

Assuming that the data has been gathered, the following steps should be executed to attain a GLM model.

- A distribution for the target variable Y should be chosen. This also determines the shape of $q(\theta)$.
- Another choice to be made is to select a link function. As mentioned before – a common choice is to use a canonical link.
- It should also be decided, which explanatory variables $X_1, \ldots, X_J$ to use to model $g(\mu)$.
- After the previous steps have been completed, a model is fitted to the data by estimating $\boldsymbol{\beta}$ and, if unknown, the dispersion parameter ϕ. This part is done by implementing the maximum likelihood method, which is described in the next subchapter.

- Once a model is built, the goodness of the model should be evaluated. For example, by using the model on a test dataset and seeing how much the predicted target variable values differ from the actual values. If necessary, changes to the model are made.

1.4 Maximum likelihood estimate

This subchapter is based on Hardin and Hilbe, 2007.

One of the most used methods for estimating unknown parameters in a model is the maximum likelihood method. A likelihood of a sample is defined as

$$L(\boldsymbol{\theta}, \phi, \mathbf{y}) = \prod_{i=1}^{N} f(\theta_i, \phi, y_i),$$

where the parameters $\boldsymbol{\theta}$ and ϕ are unknown and $\mathbf{y}$, the realisation of Y, is known. The goal is to find the values of $\boldsymbol{\theta}$ and ϕ such that the value of the likelihood function is maximized.

Since logarithm is a monotonically increasing function, then the logarithm of the likelihood is often maximized instead. The log likelihood is defined as

$$l = \ln L = \ln \left(\prod_{i=1}^{N} f(\theta_i, \phi, y_i) \right) = \sum_{i=1}^{N} \ln(f(\theta_i, \phi, y_i))$$

If the distribution belongs to the exponential family, then the log-likelihood function can be expressed as

$$l = \sum_{i=1}^{N} \left(\frac{y_i \theta_i - q(\theta_i)}{\phi} + \ln c(y_i, \phi) \right)$$

Since the parameters of interest are the coefficients $\boldsymbol{\beta}$, then the derivative of the log-likelihood with regards to $\boldsymbol{\beta}$ is found, using the chain rule:

$$\frac{\partial l}{\partial \beta_j} = \sum_{i=1}^{N} \frac{\partial l_i}{\partial \beta_j} = \sum_{i=1}^{N} \frac{\partial l_i}{\partial \theta_i} \frac{\partial \theta_i}{\partial \mu_i} \frac{\partial \mu_i}{\partial \nu_i} \frac{\partial \nu_i}{\partial \beta_j},$$

where $l_i = \ln(f(\theta_i, \phi, y_i))$.

In order to find those derivatives, two important equations are used. Given that all distributions, which belong to the exponential family, are regular, then

$$E\left(\frac{\partial l_i}{\partial \theta_i}\right) = 0$$

and

$$E\left(\frac{\partial^2 l_i}{\partial \theta_i^2} + \left(\frac{\partial l_i}{\partial \theta_i}\right)^2\right) = 0.$$

Therefore, based on the first property,

$$E\left(\frac{\partial l_i}{\partial \theta_i}\right) = \frac{E(Y_i) - q'(\theta_i)}{\phi} = 0$$

and

$$q'(\theta_i) = E(Y_i) = \mu_i,$$

where $Y_i = Y \mid \boldsymbol{x}_i$.

From the second property, we may write

$$\frac{-q''(\theta_i)}{\phi} + \left(\frac{E(Y_i) - q'(\theta_i)}{\phi}\right)^2 = \frac{-q''(\theta_i)}{\phi} + \frac{1}{\phi^2}E(Y_i - \mu_i)^2 = \frac{-q''(\theta_i)}{\phi} + \frac{1}{\phi^2}D(Y_i) = 0.$$

Thus,

$$q''(\theta_i) = \frac{1}{\phi}D(Y_i)$$

and

$$D(Y_i) = q''(\theta_i)\phi = \mathcal{D}(\mu_i)\phi,$$

where $\mathcal{D}(\mu_i)$ is the variance function. Using this and the previously shown equation $q'(\theta_i) = E(Y_i) = \mu_i$, we get:

$$\frac{\partial \mu_i}{\partial \theta_i} = \mathcal{D}(\mu_i).$$

Finally, since $\nu_i = \sum_{j=1}^{J} x_{i,j}\beta_j$, then

$$\frac{\partial \nu_i}{\partial \beta_j} = x_{i,j},$$

where $x_{i,j}$ is the value of the j-th variable of the i-th observation.

Plugging all those derivatives in and equalizing with 0, we get:

$$\frac{\partial l}{\partial \beta_j} = \sum_{i=1}^{N} \frac{y_i - q'(\theta_i)}{\phi} \frac{1}{\mathcal{D}(\mu_i)} \frac{\partial \mu_i}{\partial \nu_i} x_{i,j} = \sum_{i=1}^{N} \frac{y_i - \mu_i}{\phi \cdot \mathcal{D}(\mu_i)} \frac{\partial \mu_i}{\partial \nu_i} x_{i,j} = 0.$$

Since this equation is often difficult to solve analytically, Newton-Raphson method is frequently used instead.

1.5 Count data modelling

This subchapter is based on Hardin and Hilbe, 2007.

When the target variable is a count, for example the number of claims, then the model should return values greater than or equal to 0. For this reason, a common choice for a link function is log-link.

One of the most popular distributions to use, when modelling count, is Poisson. The probability mass function of Poisson distribution is

$$f(y;\mu) = \frac{e^{-\mu}\mu^y}{y!} = \exp\left(\ln\left(\frac{e^{-\mu}\mu^y}{y!}\right)\right) = \exp\left(\ln e^{-\mu} + \ln \mu^y - \ln y!\right) = \frac{1}{y!}\exp\left(y \cdot \ln\mu - \mu\right).$$

Therefore,

- $\theta = \ln \mu$,
- $q(\theta) = \mu$,
- $\phi = 1$,
- $EY = q'(\theta) = \mu$,
- $DY = \phi q''(\theta) = \mu$.

For Poisson distribution, the expected value EY is equal to the variance DY. To check the fit of Poisson distribution, the ratio of them $\frac{EY}{DY}$ should be found. If it is equal to or close to 1, then the distribution fits. If it is smaller than 1 ($DY < EY$), then there is underdispersion. This is usually not a problem. However, if the ratio is greater than 1 ($DY > EY$), then there is overdispersion. This could happen if the dataset has too many zeros or no zeros at all. If the ratio is large, then another distribution should be used instead of Poisson.

Another choice for the distribution of count data is negative binomial distribution. The probability mass function is

$$f(y,\mu,\alpha) = \frac{\Gamma(y+\frac{1}{\alpha})}{\Gamma(y+1)\Gamma(\frac{1}{\alpha})}\left(\frac{1}{1+\alpha\mu}\right)^{\frac{1}{\alpha}}\left(1-\frac{1}{1+\alpha\mu}\right)^{y}.$$

In exponential-family notation:

$$\frac{\Gamma(y+\frac{1}{\alpha})}{\Gamma(y+1)\Gamma(\frac{1}{\alpha})}\exp\left\{y\ln\left(\frac{\alpha\mu}{1+\alpha\mu}\right)+\frac{1}{\alpha}\ln\left(\frac{1}{1+\alpha\mu}\right)\right\}.$$

Therefore,

- $\theta = \ln\left(\frac{\alpha\mu}{1+\alpha\mu}\right)$,
- $q(\theta) = -\frac{1}{\alpha}\ln\left(\frac{1}{1+\alpha\mu}\right)$,
- $\phi = 1$,
- $EY = q'(\theta) = \mu$,
- $DY = \mu + \alpha\mu^2$.

2 Generalized additive model

Generalized linear models are easy to understand. However, they sometimes fail to describe practical problems, because real-life properties are often not linear. One way to add non-linearity to the model, is to use generalized additive model (GAM).

Generalized additive models differ from generalized linear models by using unspecified smooth functions f_j on the explanatory variables $X_j, j = 1, \ldots, J$. A GAM model has the following form:

$$g(\mu) = \nu, \quad \nu = \beta_0 + \sum_{j=1}^{J} f_j(X_j),$$

where g is the link function, $\mu = E(Y \mid \boldsymbol{x})$ is the conditional expected value of the target variable and β_0 is intercept. (Hastie, Tibshirani & Friedman, 2009)

A function f_j is smooth, if it has continuous derivatives up to some desired order in its domain (Weisstein, n.d.).

In order to get the estimates for β_0 and $f_1, \ldots, f_J$, the local scoring procedure is used. The local scoring procedure uses local averaging to generalize the Fisher scoring procedure. Hastie and Tibshirani (1990) present the algorithm as follows:

1. Initial values are assigned: $\beta_0^0 = g(\frac{1}{N}\sum_{i=1}^{N} y_i)$ and $f_1^0, \ldots, f_J^0 = 0$. The iteration parameter is set $m = 0$.

2. Updating the estimates.

   - A new target variable is defined

     $$z_i = \nu_i^m + (y_i - \mu_i^m)\left(\frac{\partial \nu_i}{\partial \mu_i}\right)^m,$$

     where $\nu_i^m = \beta_0^m + \sum_{j=1}^{J} f_j^m(x_{ij})$ and $\mu_i^m = g^{-1}(\nu_i^m)$.

   - Weights are constructed

     $$w_i = \left[\left(\frac{\partial \mu_i}{\partial \nu_i}\right)^m\right]^2 (D_i^m)^{-1},$$

     where $D_i^m = \hat{D}(Y_i)$, given that $\hat{E}(Y_i) = \mu_i^m$.

- A weighted additive model is fitted to z_i using the backfitting algorithm (described below) to get the estimated functions f_j^{m+1}, additive predictor ν^{m+1} (which also includes β_0^{m+1}) and fitted values μ_i^{m+1}.
- The convergence criterion is computed

$$\Delta(\nu^{m+1}, \nu^m) = \frac{\sum_{j=1}^{J} || f_j^{m+1} - f_j^m ||}{\sum_{j=1}^{J} || f_j^m ||}.$$

- The entire second step is repeated again with $m = m + 1$ until $\Delta(\nu^{m+1}, \nu^m)$ is below some small threshold δ.

The backfitting algorithm used to get estimated functions in the local scoring procedure is the following (Hastie and Tibshirani, 1990):

1. Initial values are assigned: $\beta_0 = \frac{1}{N}\sum_{i=1}^{N} y_i$; $f_j = f_j^0 = 0, j = 1, \ldots, J$; $m = 0$.
2. For each $j, j = 1, \ldots, J$:

$$f_j^{m+1} = S_j(y - \beta_0 - \sum_{k=1, \quad k \neq j}^{J} f_k^m \mid x_j),$$

   where $S_j(y \mid x_j)$ is a smooth function of the target variable y against the predictor x_j.
3. The second step is repeated until convergence.

James *et al.* (2013) bring out some advantages and disadvantages of using GAM. The first upside is that it allows to automatically fit a non-linear function to the variables, instead of having to try out different transformations manually. Another advantage is, it can give better predictions thanks to the non-linear fit, but still preserves the possibility to examine the effect of each variable X_j on Y, assuming all other parameters stay fixed. The main downside of GAM is that the model is restricted to finding additive relationships and might miss out on other interactions. This downside can be alleviated by manually adding additional variables in the shape of $X_j \times X_k$ to the dataset.

3 Tree models

3.1 Decision trees

Decision Tree is a non-parametric supervised learning method. The main idea behind it is to use decision rules to predict the target variable.

This chapter will focus on one of the most popular tree-type methods, called CART (Classification And Regression Tree). At first, the target variable Y is assumed to be numeric, therefore the resulting model is called a regression tree. Analogs for this model will be described later on.

The idea of a regression tree is to split the feature space by using recursive binary partitions and predict a constant c_m as a value for the numerical response variable Y in each of the M final regions.

Hastie *et al.* (2009) describe the process as follows. Let's assume there are J variables and $\boldsymbol{X}$ represents the entire feature space of the sample. Then a variable X_j would be chosen to carry out a split. If X_j is a numerical variable, a threshold $t_j \in \mathbb{R}$ is used and two new regions are obtained:

$$R_1 = \{\boldsymbol{X} \mid X_j \leq t_j\} \quad \text{and} \quad R_2 = \{\boldsymbol{X} \mid X_j > t_j\}.$$

If X_j is a categorical variable, then a subset s_j of the possible values, that the variable X_j could take, is used to conduct a split and the new regions are:

$$R_1 = \{\boldsymbol{X} \mid X_j \in s_j\} \quad \text{and} \quad R_2 = \{\boldsymbol{X} \mid X_j \notin s_j\}.$$

After splitting the data, a choice has to be made for both new regions separately. One choice is to split the region again by using either the same or another variable and the other choice is not to split the region any further and make it one of the final regions.

It is common to divide the data into two regions during a split, but there is also the possibility to split it into three or more regions. This, however, is said to be a bad

strategy, because it slices the data too quickly, not leaving enough data for lower level splits. In addition, multi-way splits can also be achieved by multiple binary splits, so the latter are preferred.

After the region $\boldsymbol{X}$ has been splitted into M subsets $R_1, R_2, \ldots, R_M$, a prediction can be made for target Y as follows:

$$\hat{y} = f(\boldsymbol{x}) = \sum_{m=1}^{M} c_m \mathbb{I}\{\boldsymbol{x} \in R_m\},$$

where c_m is some constant chosen for region m and $\mathbb{I}(A)$ is an indicator function, which equals to 1, if A is true and 0 if A is false.

The resulting model can be represented as a graph. For example, a regression tree can take the shape depicted in figure 1:

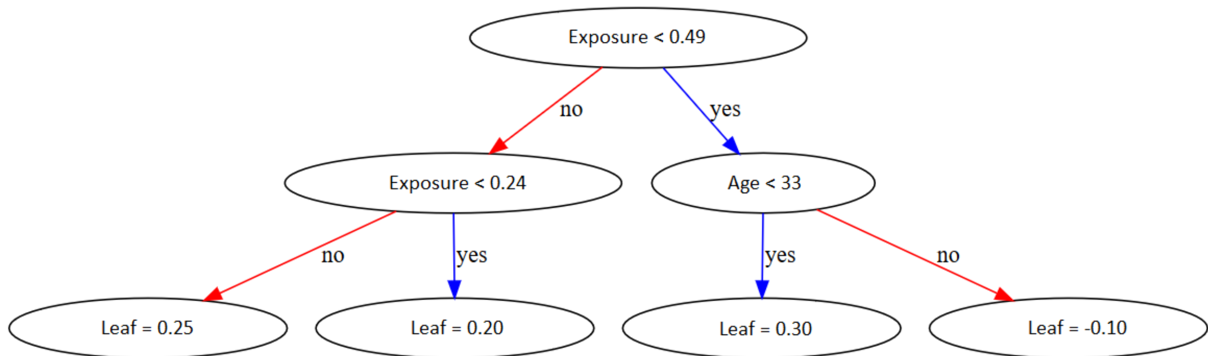


Figure 1: example of a decision tree

Certain parts of this model have specific terminology. The point where a decision rule is used to split the data, is called a node. The first (or topmost) node is called the root node and a node which is not split any further, is called a terminal node or a leaf. If node A is split into nodes B and C, then node A is the parent of B and C and nodes B and C are the children of node A.

3.1.1 Algorithm for growing a regression tree

This subsection is based on Hastie *et al.* (2009).

The previous subchapter explained, how to predict the value of the response variable when the feature space X has been split into subsets and each subset has some constant prediction. However, there are a number of ways to create those subsets depending on the chosen variable and threshold. In addition to that, there are a lot of choices for the constant predictions in each subset. This subchapter explains how to select the optimal variable, threshold and constants.

The best predictions $f(\boldsymbol{x}_i)$ are those that minimize the value of some chosen loss function, for example, the sum of squared errors:

$$L = \sum_{i=1}^{N} (y_i - f(\boldsymbol{x}_i))^2,$$

where N is the number of observations in the dataset.

A common approach to find the minima is to find the derivative and equalize it with 0:

$$L' = \left(\sum_{i=1}^{N_m} (y_i - f(\boldsymbol{x}_i))^2 \right)' = -2 \sum_{i=1}^{N_m} (y_i - f(\boldsymbol{x}_i)) = 0,$$

where N_m is the number of observations in node m and, consequently, $\sum_{m=1}^{M} N_m = N$; y_i is the value of the response variable of the i-th observation and $f(\boldsymbol{x}_i)$ is the prediction for the i-th observation.

Since regression trees usually use a constant as the prediction, then $f(\boldsymbol{x}_i)$ can be replaced with c_m. Hence, the previous expression is equal to

$$-2 \sum_{i=1}^{N_m} (y_i - c_m) = 0.$$

Therefore, the optimal prediction for c_m is the average of y_i in the node m:

$$\hat{c}_m = \frac{1}{N_m} \sum_{i=1}^{N_m} y_i.$$

The next step is to decide, which split point to use to create regions $R_1, \ldots, R_m$. Let there be two half-planes:

$$R_1(j, t_j) = \{\boldsymbol{X} \mid X_j \leq t_j\} \qquad \text{and} \qquad R_2(j, t_j) = \{\boldsymbol{X} \mid X_j > t_j\},$$

where j is the splitting variable and t_j is the split point. In order to get the minimal sum of squared error, following equation should be minimized:

$$\min_{j, t_j} \left[\min_{c_1} \sum_{\boldsymbol{x}_i \in R_1(j,t_j)} (y_i - c_1)^2 + \min_{c_2} \sum_{\boldsymbol{x}_i \in R_2(j,t_j)} (y_i - c_2)^2 \right].$$

As shown previously, the inner minimizations are solved by

$$\hat{c}_1 = \frac{1}{N_1} \sum_{\boldsymbol{x}_i \in R_1(j,t_j)} y_i \quad \text{and} \quad \hat{c}_2 = \frac{1}{N_2} \sum_{\boldsymbol{x}_i \in R_2(j,t_j)} y_i,$$

where N_1 and N_2 are the number of observations that belong to each half-plane.

The strategy to solve the outer minimization is to simply try out all the variables as j and the values those variables have in the training dataset as t_j and see which combination gives the best result. The best performing pair (j, t_j) will be chosen to execute the split.

3.1.2 Stopping conditions

This subsection is based on Hastie *et al.* (2009).

It is possible to split the data until each leaf only has one observation in it. This however leads to overfitting the model. On the other hand, if the data is not spit enough times, then the model might not capture important nuances in the dataset. Therefore, the size of a tree is an important parameter of the model.

There are some options, how to choose whether or not to split the data. One option is to carry out the split only if it prompts a greater decrease in a loss function than some threshold. However, this strategy also has a downside – it can eliminate a split which

could have a good split underneath it. A more widespread strategy is to stop splitting the data when some minimum node size (for example 5 observations) has been reached and prune the tree afterwards.

Pruning is a technique which undoes a split, meaning the observations of two nodes are merged together into their parent node. It reduces the complexity of the tree. Let T_0 be a tree with M leaves. A subtree $T, T \subset T_0$ is a tree that can be obtained by pruning the original tree T_0. In order to decide, how much a tree should be pruned, a cost complexity criterion is defined:

$$C_\alpha(T) = \sum_{m=1}^{M_T} N_m L_m(T) + \alpha M_T,$$

where M_T is the number of final regions (or leaves) in the tree T; N_m is the number of observations in region m; α is a tuning parameter, which penalizes bigger trees (large values of α result in smaller trees and vice versa) and L_m is a loss function. Although there are many choices for a loss function, a common choice is the mean squared error:

$$L_m(T) = \frac{1}{N_m} \sum_{\boldsymbol{x}_i \in R_m} (y_i - \hat{c}_m)^2.$$

The goal is to find a subtree $T_\alpha \subseteq T_0$ which minimizes $C_\alpha(T)$ for some chosen α (the best α can be found by cross-validation). It is possible to show that there is one unique smallest subtree T_α for each α. When $\alpha = 0$, then the solution is the original tree T_0. The strategy to find T_α is to prune the node that produces the smallest per-leaf increase in $\sum_m N_m L_m(T)$. This is continued until only the root of the tree is left. This gives a finite sequence of subtrees, which contains T_α.

3.1.3 Classification trees

In previous subchapters, there was an assumption that the target variable is numerical. However, the process is similar for a task, where the target variable is categorical. The difference is in the prediction estimate and in the criteria that help decide which split to

use and how to prune the tree.

In a regression task, the average of y in node m was used as a prediction and squared errors were used to decide the goodness of a split, but this does not suit a classification task. Instead a new variable is defined, representing the proportion of observations in node m that belong to class k:

$$\hat{p}_{m,k} = \frac{1}{N_m} \sum_{\boldsymbol{x}_i \in R_m} \mathbb{I}(y_i = k).$$

The observations in node m are classified to the majority class in node m:

$$k_m = \arg\max_k \hat{p}_{m,k}.$$

To decide the goodness of a split, a metric called *impurity* is used instead of the previously described mean squared error. A node is called pure if all the observations in it belong to the same class; otherwise it is impure. There are several choices for node impurity measures $L_m(T)$ (Hastie *et al.*, 2009):

- Misclassification rate:

$$\frac{1}{N_m} \sum_{x_i \in R_m} \mathbb{I}(y_i \neq k_m) = 1 - \hat{p}_{m,k_m},$$

- Gini index:

$$\sum_{k=1}^{K} \hat{p}_{m,k} \cdot (1 - \hat{p}_{m,k}),$$

- Cross-entropy:

$$-\sum_{k=1}^{K} \hat{p}_{m,k} \cdot \ln \hat{p}_{m,k},$$

where K is the number of classes. The impurity of a split can then be calculated as the following weighted average

$$\frac{1}{N}(N_L L_L(T) + N_R L_R(T)),$$

where N_L and N_R are the number of observations that go into the left and right leaf respectively and $L_L(T)$ and $L_R(T)$ are the impurity measures in each leaf.

If there are only two classes, then these formulas can be simplified. Let p note the proportion of one class. Then the formulas are following (Flach, 2012):

- Misclassification rate:
$$1 - \max(p, 1-p),$$
- Gini index:
$$2p(1-p),$$
- Cross-entropy:
$$-p \ln p - (1-p)\ln(1-p).$$

Hastie *et al.* (2009) say that for growing the tree, Gini index and cross-entropy are often preferred. There are several reasons for this: Gini index and cross-entropy are differentiable, which is an advantage when it comes to numerical optimization. In addition, they are more sensitive to changes in the leaf probabilities than misclassification rate.

For example, let there be a problem with two classes, 400 observations in each class (let's denote it by (400,400)). Let there be two choices for a split: one creates leaves (300,100) and (100, 300) and the other one (200, 400) and (200, 0). Since the second split creates a pure node, then that one is preferred. For the first split, the misclassification rate is

$$\frac{1}{800}\left(400 \cdot \left(1 - \frac{3}{4}\right) + 400 \cdot \left(1 - \frac{3}{4}\right)\right) = \frac{1}{4}$$

and for the second split it is

$$\frac{1}{800}\left(600 \cdot \left(1 - \frac{2}{3}\right) + 200 \cdot (1-1)\right) = \frac{1}{4}.$$

The Gini index for the first split is

$$\frac{1}{800}\left(400 \cdot \left(2 \cdot \frac{3}{4} \cdot \frac{1}{4}\right) + 400 \cdot \left(2 \cdot \frac{3}{4} \cdot \frac{1}{4}\right)\right) = \frac{3}{8}$$

and for the second split is

$$\frac{1}{800}\left(600 \cdot \left(2 \cdot \frac{1}{3} \cdot \frac{2}{3}\right) + 200 \cdot (2 \cdot 1 \cdot 0)\right) = \frac{1}{3}.$$

Therefore, both splits have 0.25 misclassification rate, but the second one has lower Gini index (and cross-entropy).

3.1.4 Advantages and disadvantages of decision trees

As other models, decision trees have some strong suits and also weak sides.

One of the benefits of using decision trees is that they can be visualized, which makes them easily understandable. Another advantage is that, it does not require the modeller to use dummy variables (a variable that has value 1 if the observation has a certain quality and value 0 if it does not). It can also handle both numerical and categorical data.(Decision Trees, n.d.)

However, decision trees also have some disadvantages. One of the downsides of decision trees is that they can be unstable, because using slightly different data might produce an entirely different tree. This is caused by the hierarchical structure of the tree – errors made at the top are propagated to all the nodes below it. Another disadvantage is that it is easy to overfit a model – this means that the model fits the training data very well, but lacks the ability to generalize, which gives poor results for test data. Methods like boosting, which will be described later, are created to solve a lot of the problems listed here. (Murphy, 2012)

3.1.5 Other tree-building methods

There are two main methods to build decision trees. The previous subchapters described the first method – CART (classification and regression trees). The other method is ID3 (Iterative Dichotomiser 3), which was created by Ross Quinlan in 1986. Later versions of this approach are called C4.5 and C5.0.

A downside of ID3 is that it requires the data to be categorical and does not use pruning. C4.5 and C5.0 no longer have those restrictions and are already quite similar to CART. One principal difference between the methods is that Quinlan's methods use entropy as impurity measure, and CART method uses Gini index. (Flach, 2012)

Another aspect characteristic to C5.0 is the way rule sets are developed. For this method,

the splitting rules that create the terminal nodes (leaves) are sometimes simplified after the tree is grown. This means that one or more rules can be dropped without changing group of observations that belong in that leaf. This destroys the tree structure, but creates a simpler rule set for leaves. (Hastie *et al.*, 2009)

3.2 Boosting

As described in the previous chapter, decision trees have quite a few disadvantages, such as instability or the ease of overfitting. In order to find solutions to these problems, a new approach was developed. The idea is to use a lot of weak learners (a classifier that only has a slightly better error rate than a random guess), such as decision trees, together as an ensemble.

One ensemble method is boosting. Boosting creates decision trees such, that each new tree focuses on the mistakes the previous one made. One acknowledged statistician, Leo Breiman, has called Adaboost with decision trees "the best off-the-shelf classifier in the world". (Friedman, Hastie & Tibshirani, 2000)

3.2.1 Adaboost

This subsection is based on Schapire, 2013.

One of the most popular boosting methods is Adaboost, short for adaptive boosting. It was invented in 1997 by Yoav Freund and Robert E. Schapire. Adaboost works for a two-class problem, where $Y \in \{-1, 1\}$. The idea is to give weights to all the instances, such that those, which were wrongly classified, have more influence.

As mentioned previously, boosting uses the predictions of a lot of weak learners, e.g. decision trees, together. With Adaboost, the most common choice is to use decision stumps – decision trees with one split and 2 terminal nodes. The instance weights are updated after building each tree, based on the performance of it. In the end, all the predictions $H_t(\boldsymbol{x})$ are summed together, using tree weights a_t, and the sign of that sum is given as the final prediction.

$$H(\boldsymbol{x}) = \text{sign}\left(\sum_{t=1}^{T} a_t H_t(\boldsymbol{x})\right),$$

where T is the number of trees in the model.

The algorithm to find the predictions is as follows:

1. The first step is to assign the same weight $w_{1,i} = \frac{1}{N}$ to all the instances (N is the number of observations).

2. The next step is to build a decision tree $H_t(\boldsymbol{x})$ (t is the number of the tree). For $t = 1, \dots, T$:

   (a) A decision tree is fitted to the training data using instance weights $w_{t,i}$

   (b) The fitted tree is used to get predictions $H_t(\boldsymbol{x}_i)$ for all the instances.

   (c) An error rate ϵ is calculated, using

   $$\epsilon_t = \sum_{i=1}^{N} w_{t,i} \mathbb{I}(y_i \neq H_t(\boldsymbol{x}_i)).$$

   If $\epsilon_t = 0$ or $\epsilon_t \geq \frac{1}{2}$, then the cycle is stopped and the model will consist only of $t - 1$ trees.

   (d) Tree weight a_t is calculated, using

   $$a_t = \frac{1}{2} \ln \frac{1 - \epsilon_t}{\epsilon_t}.$$

   (e) Instance weights are updated for all instances:

   $$w_{t+1,i} = w_{t,i} \cdot \frac{e^{-a_t \cdot y_i \cdot H_t(\boldsymbol{x}_i)}}{z_t},$$

   where z_t is a normalizing factor, which assures that all the weights sum up to 1.

3. In the end, the final output is calculated:

$$H(\boldsymbol{x}) = \text{sign}\left(\sum_{t=1}^{T} a_t H_t(\boldsymbol{x})\right).$$

Flach (2012) uses different form of weight updates:

$$w_{t+1,i} = \begin{cases} w_{t,i} \cdot \frac{1}{2\cdot(1-\epsilon_t)}, & \text{if} \quad y_i = H_t(\boldsymbol{x}_i) \\ w_{t,i} \cdot \frac{1}{2\cdot\epsilon_t}, & \text{if} \quad y_i \neq H_t(\boldsymbol{x}_i). \end{cases}$$

This explains the intuition behind the Adaboost algorithm. The goal is to give half of the total weight to misclassified observations and the other half to correctly classified observations. Since it is assumed, that the proportion of the misclassified observations is less than $\frac{1}{2}$, then this approach assures that the total weight of the misclassified observations is increase.

It is shown that the two weight updates are equivalent as a contribution by the author of this thesis. Since y_i and $H_t(\boldsymbol{x}_i)$ both only take values from $\{-1, 1\}$, then $y_i \cdot H_t(\boldsymbol{x}_i) = 1$, if the the instance is correctly classified and -1, if the instance is incorrectly classified. Therefore, the new weights are

$$w_{t+1,i} = \begin{cases} w_{t,i} \cdot \frac{e^{-a_t}}{z_t}, & \text{if} \quad y_i = H_t(\boldsymbol{x}_i) \\ w_{t,i} \cdot \frac{e^{a_t}}{z_t}, & \text{if} \quad y_i \neq H_t(\boldsymbol{x}_i). \end{cases}$$

Since $a_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$, it holds that $e^{a_t} = \sqrt{\frac{1-\epsilon_t}{\epsilon_t}}$. Thus, the weights can be written as

$$w_{t+1,i} = \begin{cases} w_{t,i} \cdot \frac{\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}}{z_t}, & \text{if} \quad y_i = H_t(\boldsymbol{x}_i) \\ w_{t,i} \cdot \frac{\sqrt{\frac{1-\epsilon_t}{\epsilon_t}}}{z_t}, & \text{if} \quad y_i \neq H_t(\boldsymbol{x}_i). \end{cases}$$

This can be expanded:

$$w_{t+1,i} = \begin{cases} w_{t,i} \cdot \frac{\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}}{z_t} = w_{t,i} \cdot \frac{\sqrt{\frac{\epsilon_t}{(1-\epsilon_t)^2\cdot\epsilon_t}}}{z_t/\sqrt{\epsilon_t\cdot(1-\epsilon_t)}} = w_{t,i} \cdot \frac{\frac{1}{(1-\epsilon_t)}}{z_t/\sqrt{\epsilon_t\cdot(1-\epsilon_t)}}, & \text{if} \quad y_i = H_t(\boldsymbol{x}_i) \\ w_{t,i} \cdot \frac{\sqrt{\frac{1-\epsilon_t}{\epsilon_t}}}{z_t} = w_{t,i} \cdot \frac{\sqrt{\frac{1-\epsilon_t}{\epsilon_t^2\cdot(1-\epsilon_t)}}}{z_t/\sqrt{\epsilon_t\cdot(1-\epsilon_t)}} = w_{t,i} \cdot \frac{\frac{1}{\epsilon_t}}{z_t/\sqrt{\epsilon_t\cdot(1-\epsilon_t)}}, & \text{if} \quad y_i \neq H_t(\boldsymbol{x}_i). \end{cases}$$

Since z_t is a normalizing factor, then $\sqrt{\epsilon_t \cdot (1-\epsilon_t)}$ can be omitted. This gives:

$$w_{t+1,i} = \begin{cases} w_{t,i} \cdot \frac{\frac{1}{(1-\epsilon_t)}}{z_t} = w_{t,i} \cdot \frac{1}{z_t \cdot (1-\epsilon_t)}, & \text{if} \quad y_i = H_t(\boldsymbol{x}_i) \\ w_{t,i} \cdot \frac{\frac{1}{\epsilon_t}}{z_t} = w_{t,i} \cdot \frac{1}{z_t \cdot \epsilon_t}, & \text{if} \quad y_i \neq H_t(\boldsymbol{x}_i). \end{cases}$$

The value of z_t can be calculated by using the knowledge that the sum of weights is 1:

$$\sum_{i=1}^{N} w_{t+1,i} = \sum_{\substack{i=1 \\ y_i = H_t(\boldsymbol{x}_i)}}^{N} w_{t,i} \cdot \frac{1}{z_t \cdot (1-\epsilon_t)} + \sum_{\substack{i=1 \\ y_i \neq H_t(\boldsymbol{x}_i)}}^{N} w_{t,i} \cdot \frac{1}{z_t \cdot \epsilon_t} =$$

$$= \frac{1}{z_t \cdot (1-\epsilon_t)} \cdot \sum_{\substack{i=1 \\ y_i = H_t(\boldsymbol{x}_i)}}^{N} w_{t,i} + \frac{1}{z_t \cdot \epsilon_t} \cdot \sum_{\substack{i=1 \\ y_i \neq H_t(\boldsymbol{x}_i)}}^{N} w_{t,i} =$$

$$= \frac{1}{z_t \cdot (1-\epsilon_t)} \cdot (1-\epsilon_t) + \frac{1}{z_t \cdot \epsilon_t} \cdot \epsilon_t = \frac{2}{z_t} = 1.$$

Therefore, $z_t = 2$, and an alternate formula for updating the weights is achieved:

$$w_{t+1,i} = \begin{cases} w_{t,i} \cdot \frac{1}{2 \cdot (1-\epsilon_t)}, & \text{if} \quad y_i = H_t(\boldsymbol{x}_i) \\ w_{t,i} \cdot \frac{1}{2 \cdot \epsilon_t}, & \text{if} \quad y_i \neq H_t(\boldsymbol{x}_i). \end{cases}$$

3.2.2 Gradient descent

In machine learning, it is often necessary to minimize a function, for example some loss, cost or error function.

A common way to find a global minimum is to equalize partial derivatives with respect to the parameters to zero and calculate the solution. Unfortunately, these equations are only solvable for some simpler problems – more complex problems require too much computing power. (Kelleher, Namee & D'Arcy, 2015)

One algorithm that can handle more complicated problems is gradient descent (also known as steepest descent). Gradient descent is an optimization algorithm that is often used to minimize a function. The main idea is to take a random initial guess at parameters and

then change the estimates based on the value of the negative gradient of that function. The estimates are changed until the global minimum is reached. (Kelleher *et al.*, 2015)

More specifically, Friedman (2001) explains the following.

Let there be a loss function L to calculate the loss attained from using $F(\boldsymbol{x}_i)$ as the prediction $\hat{y}_i$ ($i = 1, \ldots, N$). Let $\mathscr{L}$ be the sum of the losses of all instances in the dataset:

$$\mathscr{L}(F) = \sum_{i=1}^{N} L(y_i, F(\boldsymbol{x}_i)).$$

The goal is to find a function F^* that minimizes the sum of losses $\mathscr{L}$.

The idea of any numerical optimization procedures is to improve the estimation for $F(\boldsymbol{x})$ with iteration, so the final solution will be a sum of previous updates:

$$F^*(\boldsymbol{x}) = \sum_{t=0}^{T} f_t(\boldsymbol{x}),$$

where $f_0(\boldsymbol{x})$ is an initial guess and $f_t(\boldsymbol{x})$ are incremental functions, also called steps ($t = 1, \ldots, T; T$ is the number of steps). Let $F_t(\boldsymbol{x}) = \sum_{\tau=0}^{t} f_\tau(\boldsymbol{x})$. Then $F^*(\boldsymbol{x}) = F_T(\boldsymbol{x})$.

In case of gradient descent, the first guess $F_0(\boldsymbol{x}) = f_0(\boldsymbol{x})$ is chosen randomly and then the next steps are calculated by using the previous values. The step $f_t(\boldsymbol{x})$ can be expressed as a product of learning rate η_t and the value of the negative gradient of the loss function at $F(\boldsymbol{x}) = F_{t-1}(\boldsymbol{x})$:

$$f_t(\boldsymbol{x}) = -\eta_t \cdot g_t(\boldsymbol{x}).$$

The gradient $g_t(\boldsymbol{x})$ is:

$$g_t(\boldsymbol{x}) = \left[\frac{\partial L(y, F(\boldsymbol{x}))}{\partial F(\boldsymbol{x})}\right]_{F(\boldsymbol{x})=F_{t-1}(\boldsymbol{x})}.$$

According to Murphy (2012), a great challenge with gradient descent is how to choose the learning rate. One option is to use a constant, but it is important to keep in mind that

if the chosen rate is too small, then the convergence will be slow. On the other hand, if it is too large, then the method might "jump over" the minimum and fail to converge.

After choosing the learning rate, the new estimate for the solution can be expressed as

$$F_t(\boldsymbol{x}) = F_{t-1}(\boldsymbol{x}) - \eta_t \cdot g_t(\boldsymbol{x}).$$

Then these calculations are done again until the step size $f_t(\boldsymbol{x})$ is smaller than some threshold or a maximum number of steps T is reached.

3.2.3 Stochastic gradient descent

This subchapter is based on (Ruder, 2016).

Regular gradient descent, which was described before, upgrades the estimates after calculating the derivatives for every row in the dataset. For large datasets, this can be very time-consuming and create memory problems. To solve these issues, new approaches have been invented, which make a trade-off between the accuracy of the result and the calculation time.

One adaption of this method is stochastic gradient descent, which at first shuffles all the rows in the dataset and then updates the estimates after going through each row in the dataset, instead of going through all the rows before updating the parameters. In addition to usually being faster than regular gradient descent, it has another advantage – stochastic gradient descent gives the opportunity to update parameters when new data becomes available.

Stochastic gradient descent also has a downside – due to the fact that it is updating the estimates after each row, the estimates have a high variance. This can sometimes cause convergence issues.

Another adaption of gradient descent is mini-batch gradient descent. It performs an update after every "mini-batch" of n rows. This way it has more stable convergence than

stochastic gradient descent, but is still faster than regular gradient descent.

3.2.4 Gradient boosting

This subchapter is based on Friedman (1999).

Gradient boosting is another ensemble method, which iteratively fits some weak learners (e.g. decision trees) to pseudo-residuals. The pseudo-residuals are the gradient of some differentiable loss function with respect to the model values at training points, evaluated at the current step. This means that each consecutive learner learns from the mistakes that the previous one made.

The goal is to find a function $F^*(\boldsymbol{x})$ that maps the observation $\boldsymbol{x}$ to the target variable y so, that the sum of loss functions $\mathscr{L}$ is minimized.

As previously described, boosting uses a weighed sum of some weak learners $b(\boldsymbol{x}, \mathbf{p})$:

$$F(\boldsymbol{x}) = \sum_{t=1}^{T} \beta_t b(\boldsymbol{x}, \mathbf{p}_t),$$

where β_t are the weights, $\mathbf{p}_t$ are parameters for the weak learner, and T is the number of weak learners.

In a problem, where decision tree is used as a weak learner, $b(\boldsymbol{x}, \mathbf{p})$ is a M-terminal node regression tree. At each iteration, the decision tree divides the input space into M regions $R_{m,t}$ and predicts the conditional mean $\bar{y}_{m,t}$ of the region $R_{m,t}$ as output:

$$b(\boldsymbol{x}; \{R_{m,t}\}_{m=1,\dots,M}) = \sum_{m=1}^{M} \bar{y}_{m,t} \mathbb{I}(\boldsymbol{x} \in R_{m,t}).$$

More precisely, the algorithm for gradient boosting using decision trees is:

1. First, an initial guess is made:

$$F_0(\boldsymbol{x}) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$$

2. The next step is to add a decision tree to the model. For $t = 1, \dots, T$:

(a) The gradient is calculated for each observation in the dataset:

$$\tilde{y}_{i,t} = -\left[\frac{\partial L(y_i, F(\boldsymbol{x}_i))}{\partial F(\boldsymbol{x}_i)}\right]_{F(x)=F_{t-1}(x)}, \quad i = 1, \dots, N.$$

(b) A decision tree D_t with M terminal nodes is fit to the input data $(\boldsymbol{x}_i)_{i=1,\dots,N}$ to predict the gradient vector $(\tilde{y}_{i,t})_{i=1,\dots,N}$. Terminal regions $R_{m,t}$ are created $(m = 1, \dots, M)$.

(c) A constant $\gamma_{m,t}$, which is going to be the prediction for gradients that belong to $R_{m,t}$ is calculated for each terminal region $m = 1, \dots, M$,

$$\gamma_{m,t} = \arg\min_{\gamma} \sum_{\boldsymbol{x}_i \in R_{m,t}} L(y_i, F_{t-1}(\boldsymbol{x}_i) + \gamma).$$

(d) A new prediction for the observations is found, using learning rate $0 < \eta \leq 1$:

$$F_t(\boldsymbol{x}) = F_{t-1}(\boldsymbol{x}) + \eta \cdot \gamma_{m,t} \mathbb{I}(\boldsymbol{x} \in R_{m,t}).$$

Empirically it was found that the generalization error is smaller, if the learning rate is small ($\eta \leq 0.1$). However, this also increases computing time.

3. The final prediction is made by $F_T(\boldsymbol{x})$.

In this algorithm, all the instances were used at each iteration. However, it is shown that if randomisation is used in the process, then approximation accuracy and computation speed can be improved. This gave the idea of stochastic gradient boosting. The process is analogous to stochastic gradient descent: instead of using the entire dataset, a subsample is drawn without replacement and used to fit the decision tree and compute a model update.

3.2.5 XGBoost

This subchapter is based on Chen & Guestrin (2016).

Boosting algorithms are often used on large datasets, thus the model building process can become very time-consuming. Another expansion of gradient boosting – XGBoost (extreme gradient boosting) was created to make the tree-building process quicker. XGBoost

takes the base of gradient boosting and improves it, for example, with cache access patterns and data compression, resulting in a system that runs ten times faster than gradient boosting.

The improvements can be divided into two groups: mathematical and computer hardware related. The mathematical improvements that XGBoost makes, are the following:

- **Regularization.** Instead of minimizing just the loss function, XGBoost adds a regularization term to the loss function. Therefore, it minimizes the following function:

$$\sum_{i=1}^{N} L(y_i, \hat{y}_i) + \sum_{t=1}^{T} \Omega(D_t), \qquad \Omega(D) = \alpha M + \frac{1}{2}\lambda \sum_{m=1}^{M} \gamma_m^2,$$

  where Ω is the regularization function, T is the number of trees, D_t is a tree structure with M leaves and leaf predictions γ_m $(m = 1, \ldots, M)$, α is a penalizing parameter (which controls, how much the tree size is penalized) and λ is another penalizing parameter (which controls, how much the size of leaf predictions is penalized).

- **Taylor series approximation.** In order to minimize the new loss function

$$\mathcal{L}_t = \sum_{i=1}^{N} L(y_i, \hat{y}_i^{(t-1)} + D_t(\boldsymbol{x}_i)) + \Omega(D_t)$$

  a second order Taylor approximation is used:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{N} [L(y_i, \hat{y}_i^{(t-1)}) + g_i D_t(\boldsymbol{x}_i) + \frac{1}{2} h_i D_t^2(\boldsymbol{x}_i)] + \Omega(D_t),$$

  where $g_i = \partial_{\hat{y}^{(t-1)}} L(y_i, \hat{y}_i^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} L(y_i, \hat{y}_i^{(t-1)})$.

  Since the constant term plays no part in optimizing, it can be omitted:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{N} [g_i D_t(\boldsymbol{x}_i) + \frac{1}{2} h_i D_t^2(\boldsymbol{x}_i)] + \Omega(D_t).$$

  Let $I_m = \{i \mid \boldsymbol{x}_i \in R_m\}$ denote a set of instances in leaf m. Then the previous equation can be expressed as:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{N} [g_i D_t(\boldsymbol{x}_i) + \frac{1}{2} h_i D_t^2(\boldsymbol{x}_i)] + \alpha M + \frac{1}{2}\lambda \sum_{m=1}^{M} \gamma_m^2 =$$

$$= \sum_{m=1}^{M} [(\sum_{i \in I_m} g_i)\gamma_m + \frac{1}{2}(\sum_{i \in I_m} h_i + \lambda)\gamma_m^2] + \alpha M$$

Therefore, the leaf predictions γ_m^* can be computed by

$$\gamma_m^* = -\frac{\sum_{i \in I_m} g_i}{\sum_{i \in I_m} h_i + \lambda}$$

and the corresponding optimal value of the loss function is

$$\tilde{\mathcal{L}}^{(t)}(D) = -\frac{1}{2} \sum_{m=1}^{M} \frac{(\sum_{i \in I_m} g_i)^2}{\sum_{i \in I_m} h_i + \lambda} + \alpha M.$$

In order to evaluate the goodness of a split, the following function is used, assuming that I_L and I_R are the sets of instances in the left and right nodes accordingly $(I = I_L \cup I_R)$:

$$\mathcal{L}_{split} = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \alpha.$$

- **Column subsampling.** XGBoost boost also uses column subsampling. This means that only a random subset of features is used to build a tree.

- **Approximate algorithm.** Gradient boosting finds the best split by trying out all the possible splits. However, this method, called the exact greedy algorithm, is computationally demanding. To do splitting effectively, the algorithm first sorts numerical data by the feature values. Unfortunately, that is impossible, when the data does not entirely fit into memory. To solve this problem, an approximate algorithm is used instead, where the splitting point is chosen from a list of proposed candidates. There are two versions of this method – the global variant chooses the splitting candidates in the initial phase and the local variant re-chooses the candidates after every split.

- **Weighted quantile sketch.** An important part of the approximate algorithm is to choose the candidates of the split point. Usually, percentiles of the feature are used. Let $\mathcal{D}_j = \{(x_{1,j}, h_1), \ldots, (x_{N,j}, h_N)\}$ be a set of pairs of the j-th feature values

and the second order gradient statistics. Then the rank function $r_j(z)$ is defined as the proportion of instances whose feature j value is smaller than z, $r_j : \mathbb{R} \rightarrow [0, \infty)$:

$$r_j(z) = \frac{1}{\sum_{(x,h)\in D_j} h} \sum_{\substack{(x,h)\in D_j \\ x<z}} h.$$

The aim is to find candidate split points $\{s_{j,1}, \ldots, s_{j,l}\}$ such that

$$| \, r_j(s_{j,k}) - r_j(s_{j,k+1}) \, | < \epsilon, \quad s_{j,1} = \min_i x_{i,j}, \quad s_{j,l} = \max_i x_{i,j},$$

where ϵ is the approximation factor, $\frac{1}{\epsilon} \approx l$.

Since the loss function (introduced in the beginning) can be re-written as

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{N} [g_i D_t(\boldsymbol{x}_i) + \frac{1}{2} h_i D_t^2(\boldsymbol{x}_i)] + \Omega(D_t) = \sum_{i=1}^{N} \frac{1}{2} h_i (D_t(\boldsymbol{x}_i) - \frac{g_i}{h_i})^2 + \Omega(D_t) + constant,$$

which is a weighted squared loss with labels $\frac{g_i}{h_i}$ and weights h_i, then h_i can be thought of as instance weights.

Chen and Guestrin introduced a new weighted quantile sketch algorithm to solve this quantile finding problem on weighted data. The details of this weighted quantile sketch algorithm can be found in their (2016) paper.

- **Sparsity aware split finding.** Real-life problems often produce datasets that are sparse. There could be several reasons for that, for example, missing values in the data, zero entries in the data, one-hot encoding is used etc. To deal with this problem, XGBoost adds a default direction in each node. When a value is missing, then the instance is classified in the default direction. In order to find the default direction, the algorithm finds the best split point without using the missing values. After that, it tries to classify all the instances with missing values to the left leaf and then to the right leaf and sees which gives maximum gain. The leaf with maximum gain becomes the default direction.

In addition to new mathematical alternations, XGBoost also exploits the properties of computer hardware.

- **Column block for parallel learning.** The most time-consuming part of building a tree is sorting the data. In order to save time on it, XGBoost stores the data in in-memory units called a *block*. The data is sorted once in the beginning, saved in a compressed column (CSC) format and then this layout is reused in the later iterations. Using the CSC format allows the collection of statistics for each column to be parallelized.

  In the exact greedy algorithm, the entire dataset is stored in the same block. When splitting the data, all the leaves are found collectively, so the dataset is scanned only once. In case of the approximate algorithm, the data can be stored in multiple blocks, each block corresponding to a subset of rows. Those blocks can be stored across different machines or on a disk.

- **Cache-aware access.** The previously described block structure helps to optimize the split finding algorithm, but requires non-continuous memory access to retrieve the gradient statistics by row index. That slows down split finding when the gradient statistics do not fit into CPU cache. In case of exact greedy algorithm, this problem is solved by allocating internal buffers in each thread, into which the gradient statistics are fetched. For approximate algorithm, the problem is solved by using the right block size.

- **Blocks for out-of-core computation.** When the dataset is too large to fit into the main memory, it is saved onto the hard disk. However, reading the data from the disk takes a lot of time. To reduce that time, XGBoost compresses blocks before saving them on the disk. Another strategy to improve the out-of-core computation, is to save the data onto multiple disks, if more than one are available. This technique is called block sharding.

4 Modelling claim frequency

In this section, a generalized linear model, a generalized additive model and an XGBoost model will be fit to real-life motor third party liability data to predict claims frequency.

Motor third party liability (MTPL) is a type of insurance, which is compulsory to all vehicles registered in the Estonian Motor Vehicle Register (with some exceptions). It covers the cost of damage caused to a third party's health and/or vehicle in case of an accident. ("Kohustuslik liikluskindlustus", n.d.)

Two datasets were acquired from the Estonian Motor Insurance Bureau (LKF). The first one contained all the regular policies for passenger cars (category M1), all-terrain vehicles (M1G and N1G) and goods vehicles (N1), where the start date is between January 1st 2014 and December 31st 2018. This dataset had 9 472 796 rows and 20 variables: policy ID, cover start date, cover end date, type of the owner of the vehicle (a person or a legal entity), the first 7 digits of the owner's national identification number, type of the policy holder, the first 7 digits of the policy holder's national identification number, type of the responsible user of the vehicle, the first 7 digits of the responsible user's national identification number, make of the vehicle, model of the vehicle, category of the vehicle (M1, M1G, N1 or N1G), the year of the first registration of the vehicle, engine power in kilowatts, gross weight, number of seats, speed limit, frame type, color and fuel type of the vehicle.

The second dataset contained information about the claims, where the accident was caused by a person related to a policy in the first dataset. It had 137 317 rows and 4 variables: policy ID, accident date and time, accident location country and the first 3 digits of the national identification number of the person, who caused the accident.

4.1 Data preprocessing

Before any analysis could be done, the data had to be preprocessed. Python software (version 3.6.5) (Van Rossum & Drake, 2009) was used for this and the code can be found in appendix 8.

Since the goal of this analysis is to predict claim frequency in order to give a fair price for the protection, then the variables in the claims dataset cannot be used, because the value of those variables would not be known at the time of pricing. Therefore, the only useful information the second dataset holds, is the number of claims related to a policy. This information was appended to the first dataset.

Since claim frequency is the quotient of number of claims and exposure, then in order to calculate the latter, the number of days between the cover end date and start date was found. There were some rows, where exposure was negative, meaning the start date would come after the end date. All instances that had exposure less than one day, were deleted from the dataset. Cover start month was also extracted from the dates.

Two new variables were created from each of the personal identification number related columns: gender and age. The first number of the identification number determines the gender – "3" stands for a male born between 1900-1999, "4" stands for a female born between 1900-1999, "5" and "6" for a male and female born after 2000, respectively. If the owner/policy holder/responsible user was a legal entity, then the corresponding personal identification number would be the company's register code. In that case, the personal identification number was set to N/A (Not Available). There were also some instances, where the first number was not 3,4,5 nor 6 – those were set to N/A as well.

The second and third number of an identification number determine the person's year of birth, the fourth and fifth determine the month and sixth and seventh determine the day. Owner's, policy holder's and responsible user's ages were calculated by subtracting the date of birth from cover start date. There were also some negative ages, which implied that the person was born after the cover start date. Assuming it was incorrect data, the

gender and age both were set to N/A for those instances. The dataset also included some extreme ages. If the age was less than 18 or more than 95, then again the gender and age were set to N/A.

Most of the categorical variables had a lot of different values, some of which were only represented once. Therefore, almost all the categorical features were regrouped by keeping some values, that had the highest count and grouping the rest of the values (that were not N/A) as "other". The number of categories kept was decided separately for each variable by looking at the point, where the count was under some threshold (e.g. 100000) or dropped significantly compared to the previous count. As a result, 20 most popular values were kept for the make of the vehicle; 5 values were kept for the model of the vehicle; 8 values were kept for the number of seats; 7 values were kept for the frame type of the vehicle; 5 values were kept for fuel type.

The color of the vehicle was also altered by grouping similar colors together. For example light red, red and dark red and pink were grouped together as "red". As a result, 28 different categories were replaced by 10: white, grey, brown, red, yellow, green, blue, silver, black and N/A.

Using the first registration date, the age of the vehicle was also calculated as the difference between cover start date and the vehicle's registration date. Since there were some cars with extreme ages, then all vehicle's that had age over 40 were classified as "older".

There were 4 rows, where the engine power was 0 kW. Those rows were deleted. The variable "speed limit" had 9 243 310 N/A values and was also dropped from the dataset.

A new dataset was acquired as a result of the preprocessing. This dataset had 9 245 584 rows and 22 variables: cover start month, the type of the owner, owner's gender, owner's age, the type of the policy holder, policy holder's gender, policy holder's age, the type of the responsible user, responsible user's gender, responsible user's age, vehicle make, vehicle model, vehicle age, vehicle category, engine power, gross weight, number of seats,

frame type, color, fuel type of the vehicle, number of claims, exposure in years. Since the dataset did not contain information about the planned duration of the policy, exposure was also used in modelling as a planned duration. More information about the dataset can be found in appendix 1.

This dataset was divided into 3 parts – the training set (60 % of the data), which was used for training the model, validation set (20 % of the data), which was used to tune the model parameters, and a test set (20 % of the data), which was used to test the results.

4.2 Modelling with XGBoost

The first model was created using XGBoost in Python (version 3.6.5). The core version of the XGBoost was used in this thesis (Chen & Guestrin, 2016); however, there is another interface for it in the `"Scikit-learn"` package. The code can be found in appendix 9.

Even though it is possible to build decision trees by using categorical variables as predictors, XGBoost does not support it. Therefore, all the categorical variables were altered by using one-hot encoding. One-hot encoding creates new variables for all the levels of all categorical variables with values 1 or 0, based on whether or not the original variable value is equal to that level or not. As a result, a new dataset with 109 variables was obtained.

In order to be able to compare the results of the XGBoost model to the results of the generalized linear model and the generalized additive model, Poisson deviance is chosen as a loss function. It is specified by using the `"objective"` parameter.

The XGBoost core package has a data structure called a DMatrix, which is optimized for both memory efficiency and training speed. The parameter `"base_margin"` sets a global bias to all the predictions and is therefore an analogue to the offset functionality in GLM. Natural logarithm of the exposure was used as the base margin.

After preparing the data, the next step was to find suitable values for other parameters. Five most important parameters were studied – the learning rate (which can be set by the parameter `"eta"`), the maximum depth of the tree (parameter `"max_depth"`, penalizing factors α and λ (parameters `"alpha"` and `"lambda"`) and the maximum number of trees built.

At first, a 50000 row subset was taken from the training dataset and 768 different combinations of the parameters were looked through: six values for the learning rate: 0.005, 0.01, 0.05, 0.1, 0.3 and 0.5; four values for the depth of the tree: 3, 5, 6, 8; four values for lambdas: 0, 1, 3, 5; four values for alphas: 0, 1, 3, 5; two for the maximum numbers of trees built: 50 and 100. However, a parameter called `"early_stopping_rounds"` was used. It has a default value of 10, which means, if the loss function value has not reduced by building the last 10 trees, then no more trees are added to the model. Therefore not all models had 50 or 100 trees in them.

The best result was given by 0.3 as learning rate, 3 as the maximum depth, 1 as lambda, 3 as alpha and 100 trees as the maximum number of trees (parameter group 1). It turned out that alpha and lambda values did not affect the results much, therefore they were not looked into any more and all the next models had lambda set to 1 and alpha set to 3. It was also seen that 100 trees was not enough for the smaller learning rates to reach their best predictions, therefore new models with learning rates 0.005, 0.01 and 0.05 were tested with maximum depth set to 3 and the maximum number of trees increased to 1500 (and early stopping round set to 10). They reached their best predictions between 800 and 1000 trees, however, these predictions were not better than the predictions given by the parameter group 1.

In order to find the best learning rate 0.2 and 0.4 were also considered, but 0.3 still gave the best result. After comparing 0.26, 0.28, 0.3, 0.32, 0.34 as learning rates and 1, 2, 3, 4, 5 as the maximum depth of the tree, the best combination was attained by learning rate 0.34 and the maximum depth 2.

A new model was built by using the best parameters on the entire dataset. The best model was attained with 410 trees. However, it is worth mentioning that the last trees only gave a very small gain in the loss (the difference in the mean Poisson deviance of the validation set between using 40 trees and 410 trees was less than 0.001).

The most used variables when predicting with XGBoost were: exposure (173 splits), the age of the vehicle (158 splits), the age of the owner (103 splits), the engine power of the vehicle (91 splits), the age of the policy holder (85 splits), gross weight of the vehicle (73 splits), the age of the responsible user (62 splits). More details can be found in appendix 2. Therefore, XGBoost used mostly the numeric variables to make a prediction.

A simpler model, using only the numeric variables, was also fit to the model. The parameters were studied again, using only a subset of 50000 training data rows. At first, alpha and lambda were looked at, but they did not influence the results much, so they were again set to 3 and 1 correspondingly. Next, the combinations of 2,3,4 and 5 as tree depth and 0.2, 0.3, 0.4 as learning rate were studied, using 100 trees and early stopping round set to 10. The best result were given by tree depth 3 and learning rate 0.3. Finally, the combinations of learning rates 0.26, 0.28, 0.3, 0.32, 0.34 and tree depths 2 and 3 were studied. The best model was achieved with learning rate 0.28 and a tree depth of 2.

A new model was built by using all the rows in the training dataset with learning rate 0.28, maximum tree depth 2, alpha 3, lambda 1 and maximum number of trees 1500 with early stopping round parameter set to 10. The new model had 313 trees. The most used variables were vehicle age (192 splits) and exposure (188 splits). More details can be found in appendix 3.

4.3 Modelling with GLM

Next, generalized linear models were used to create two models - one started with all the variables and another used only the numeric variables.

One of the downsides of GLM is that it cannot handle missing values. For this reason, the dataset was altered again. First, the age of the owner was changed to 0, and the gender was changed to "legal", if the type of the owner was a legal entity. The same changes were made for the policy holder and the responsible user. If the type of the responsible user was N/A, then the gender was changed to "No responsible user" and the age was set to 0. After these changes, the gender variables uniquely determine the type of the owner, policy holder and responsible user. For this reason, the variables describing the types were deleted from the dataset.

For the rest of the categorical variables, either the category "other" or, if there was no such category, the most popular category was appointed. The missing values for the genders of the owner and policy holder were changed to "male". The most popular category of the variable "responsible user" was "no responsible user". However, since this value would not make sense (because the remaining rows do have a responsible user), then the gender was changed to the second most popular value – "male" – instead. The missing values of the color of the car were substituted with "grey". The number of seats, frame type and fuel type were appointed the value "other".

There were five numeric variables, which had missing values: the age of the owner (17142 missing values), the age of the policy holder (10782), the age of the responsible user (143), the gross weight of the vehicle (67), the age of the vehicle (10). The first idea was to use a k-nearest neighbor method for the imputation. However, since this process is computationally heavy and since there are not many missing values, just a rounded value of the average of the training set was used instead.

The analysis was done with R software (version 3.4.4) (R Core Team, 2014) by using RStudio (version 1.1.442) (RStudio Team, 2015). Function `"bigglm"` from package `"biglm"` (Lumley, 2013) was used for modelling. The code can be found in appendix 10.

The number of claims are assumed to have a Poisson distribution and therefore, the logarithm of exposure in years could be used as the offset.

At first, the correlations between the numeric variables were examined. As expected, the age of the owner and the age of the policy holder were highly correlated. Therefore, the variable "age of the owner" was removed from the dataset.

The dataset also contained a lot of categorical variables, with many levels. Keeping all of those levels would make the model harder to understand and also less accurate. Therefore, the levels of the categorical variables were regrouped after creating the first model. The values which had similar coefficients were grouped together.

- The make of the car was divided into 4 groups. Group 1 contained BMW, Mercedes-Benz,Mazda and Renault. Group 2 contained Chrysler, Citro, Honda, Opel, Peugeot, Subaru, Toyota and Audi. Group 3 had Ford, Kia, Nissan, Škoda, Volkswagen, Volvo and the makes that were classified as other in the dataset. Group 4 had Mitsubishi and Hyundai.

- The color of the car was divided into 3 groups: the first one contained blue, black, grey and silver; the second had red, brown and green; the third group contained white and yellow.

- The start months of the cover was also divided into 3 groups: the first contained January, February, August, September, October and December. The second group only contained November and the third group contained March, April, May, June and July.

- The number of seats was divided into 4 groups: the first group contained 2 and 3 seats, the second 4 and 5 seats and all "other" seat numbers. The third group contained 6 and 7. The fourth group contained 8 and 9.

- The models of the car were divided into 2 groups: The first one had "Passat" and "Passat Variant". The second group had all the other models.

- The frame type was divided into 3 groups: The first one had sedan and hatchback. The second group contained coupé, wagon and minivans. The third group contained pickup trucks, vans and the frame types that were categorized as "other".

- The fuel types were also categorized into 3 groups. The first one contained "gasoline-hybrid", "gasoline-catalyst", "diesel". The second one had only "gasoline". The third one contained "electricity" and the fuel types that were classified as "other".
- The categories of the vehicles were divided into 2 groups. The first one contained M and M1. The second one contained N and N1.
- All the levels of the gender of the responsible user, besides "no responsible user", were merged into one group. Therefore the variable now represents whether or not there is a responsible user.

After the data alterations, all the variables were statistically significant. New models were created by removing each of the variables one by one and the new models were compared to the current one by using the likelihood ratio test. None of the new models with one less variables could be proven to be better. The model was checked for overdispersion by using Pearson residuals. The final model was (appendix 4):

$\ln(\text{number of claims}) = \ln(\text{exposure}) - 2.6428$

$+ 0.0415 \cdot \mathbb{I}(\text{cover start month} = \text{November})$

$- 0.0468 \cdot \mathbb{I}(\text{cover start month} \in \{ \text{ March, April, May, June, July } \})$

$- 0.0298 \cdot \mathbb{I}(\text{the owner is a legal entity}) - 0.0527 \cdot \mathbb{I}(\text{the gender of the owner is male})$

$- 0.2049 \cdot \mathbb{I}(\text{the policy holder is a legal entity})$

$- 0.1068 \cdot \mathbb{I}(\text{the gender of the policy holder is male}) - 0.0054 \cdot \text{the age of the policy holder}$

$+ 0.2897 \cdot \mathbb{I}(\text{there exists a responsible user}) - 0.0042 \cdot \text{the age of the responsible user}$

$- 0.1205 \cdot \mathbb{I}(\text{the vehicle make is Chrysler, Citroën, Honda, Opel, Peugeot, Subaru,}$
$\text{Toyota or Audi })$

$- 0.1861 \cdot \mathbb{I}(\text{the vehicle make is Ford, Kia, Nissan, Škoda, Volkswagen, Volvo or "other"})$

$- 0.2500 \cdot \mathbb{I}(\text{the vehicle make is Mitshubishi or Hyundai})$

$+ 0.0529 \cdot \mathbb{I}(\text{the vehicle model is Passat or Passat Variant})$

$- 0.0127 \cdot \text{the age of the vehicle}$

$+ 0.2644 \cdot \mathbb{I}(\text{the category of the vehicle is N1 or N1G})$

$+ 0.0003 \cdot \text{the engine power} + 0.0002 \cdot \text{the gross weight}$

$- 0.1743 \cdot \mathbb{I}(\text{the seat count is 4 or 5 or "other"}) - 0.2190 \cdot \mathbb{I}(\text{the seat count is 6 or 7})$

$- 0.3913 \cdot \mathbb{I}(\text{the seat count is 8 or 9})$

$- 0.4725 \cdot \mathbb{I}(\text{the frame type is pickup, van or "other"})$

$+ 0.0985 \cdot \mathbb{I}(\text{the frame type is sedan or a hatchback})$

$- 0.0368 \cdot \mathbb{I}(\text{the color is red or brown or green}) + 0.0631 \cdot \mathbb{I}(\text{the color is white or yellow})$

$- 0.4352 \cdot \mathbb{I}(\text{the fuel type is gasoline}) + 0.5432 \cdot \mathbb{I}(\text{the fuel type is electric or "other"})$

$- 0.7644 \cdot \text{exposure}$

The baselines for categorical variables were:

- cover start month: January, February, August, September, October, December;
- gender of the owner: female;
- gender of the policy holder: female;
- type of responsible user: no responsible user;
- vehicle make: BMW, Mercedes-Benz, Mazda, Renault;
- vehicle model: Avensis, Golf, Octavia, other;
- vehicle category: M1 or M1G;
- seat count: 2 or 3;
- frame type: coupé, wagon, minivan;
- color: blue, black, grey, silver;
- fuel type: gasoline-hybrid, gasoline-catalyst, diesel.

For the following analysis, it is assumed, that when discussing the change of one variable, all other variable values stay the same.

The influence of the age of the policy holder can be interpreted as such: if there are two covers and the policy holder of the first cover is one year older than the second, then, per average, the first cover will result in 0.54% less claims ($e^{-0.0054} = 0.9946, \quad 0.9946 - 1 = -0.0054$). The effect of other numeric variables can be explained analogically.

The influence of the policy holder being a legal entity can be interpreted as such: if there are two covers and the policy holder of the first one is a legal entity and the policy holder of the second cover is a person, then, per average, the first cover will result in 18.53% less claims ($e^{-0.2049} = 0.8147, \quad 0.8147 - 1 = -0.1853$). The effect of other indicator variables can be explained analogically.

If a coefficient of a numeric variable is positive, then a bigger variable value implies, per average, more claims. If a coefficient of an indicator variable is positive, then a cover which has the quality implied in the indicator function, results in more claims (per average), than a cover that has the quality of a baseline group.

If a cover started in March, April, May, June or July, then, per average it had the least claims; policies started in January, February, August, September, October, December had more claims and the policies that started in November caused the most claims per average.

If the owner of the vehicle was male, then the number of claims estimated was the lowest; legal entities had a higher claim estimation and women had the highest estimation. For policy holders, the order from lowest to highest claim count estimation was: legal entities, men, women. Policies that had a responsible user had, per average, more claims than the policies that did not have a responsible user.

The least claims happened to Mitshubishi and Hyundai vehicles. More claims happen to Ford, Kia, Nissan, Škoda, Volkswagen, Volvo and "other" vehicles. Third were Chrysler, Citroën, Honda, Opel, Peugeot, Subaru, Toyota, Audi. The most claims happened to BMW, Mercedes-Benz, Mazda and Renault vehicles. If the model of the vehicle was Passat or Passat Variant, then per average, the policy had more claims than policies with

other model types.

N1 and N1G category vehicles had a higher claim estimate than M1 and M1G. If the vehicle's frame type was pickup, van or "other", then it had the least claims per average. Coupés, wagons and minivans had more claims; sedans and hatchbacks had the highest count, per average.

The least claims happened to vehicles with 8 or 9 seats; more claims happened to vehicles with 6 or 7 seats; next were 4, 5 and "other" seats; the most claims happened to vehicles with 2 or 3 seats.

Red, brown and green vehicles had the lowest claim estimate; blue, black, grey and silver cars had a higher estimate; white and yellow vehicles had the highest estimate. One explanation to this could be that taxis in Estonia are ofter colored white or yellow.

The vehicles that had gasoline as fuel type, caused the least accidents. Next were gasoline-hybrid, gasoline-catalyst and diesel cars. Electric and "other" vehicles had the highest estimate.

Numeric variables engine power and gross weight had a positive coefficient, therefore the bigger the value, the more claims are estimated. Other numeric variables had a negative coefficient and therefore lowered the estimated number of claims.

Another model was created by using only numeric variables. The model did not have an overdispersion problem. The best model included all of the numeric variables (appendix 5):

$$\begin{aligned}
&\ln(\text{number of claims}) = \ln(\text{exposure}) - 2.8118 - 0.0052 \cdot \text{the age of the policy holder} \\
&+ 0.0007 \cdot \text{the age of the responsible user} - 0.0186 \cdot \text{the age of vehicle} \\
&+ 0.0011 \cdot \text{the engine power} + 0.0001 \cdot \text{the gross weight} - 0.8133 \cdot \text{exposure}
\end{aligned}$$

Therefore, bigger engine power, bigger gross weight and an older responsible user increase

the estimated number of claims. Other numeric variables decreased the estimates.

4.4 Modelling with GAM

The third type of model applied was a generalized additive model. The same dataset that was used for GLM (with recoded categorical variables) was also used for GAM. The modelling was done with function `"bam"` from package `"mgcv"` (Wood, 2017), which is meant for creating a generalized additive model using a big dataset. The code can be found in appendix 10.

The first model (appendix 6) was created by using all the variables and by applying cubic splines as smooths on the numeric variables. The best number of knots was found by trying out different values. Knots are cutpoints - the function fits a cubic polynomial with continuous derivatives piecewise at each knot.

As a result, a model that contained all the variables, except the age of the owner, was attained. The baselines for categorical variables were the same as for GLM and the influences of the indicator variables were ordered exactly the same way as for the GLM model. The number of knots used for policy holder's age was 15, for responsible user's age, vehicle's age, engine power and gross weight, 8 knots were used; for exposure, 20.

Another model (appendix 7) was created by using just the numeric variables and applying cubic splines as smooths again. For policy holder's age 10 knots were used, for responsible user's age, vehicle's age, engine power and gross weight, 5 knots were used; for exposure, 10. Neither of the models had a problem with overdispersion.

4.5 Results

Six models were obtained as a result of the modelling process. Three of them were created by using all the variables and three were created by using only the numeric variables.

Poisson deviance was used to compare the performances of the models, based on the test data (appendix 11). Poisson deviance can be calculated as:

$$D = \sum_{i=1}^{N} 2 \cdot (y_i \ln \frac{y_i}{p_i} - (y_i - p_i)),$$

where N is the number of observations, y_i is the true value of the response variable and p_i is the predicted value (Rodriguez, 2020).

The Poisson deviances for the models that used all of the variables as a starting point were:

Model	XGBoost	GLM	GAM
Poisson deviance	221 206	223 092	221 946

The Poisson deviances for the models that used only the numeric variables as a starting point were:

Model	XGBoost	GLM	GAM
Poisson deviance	221 887	223 719	222 597

Therefore, both XGBoost models had better Poisson deviances than GLM or GAM. GAM had the second best outcome and GLM had the worst result out of these models.

However, it is also worth mentioning that even though GLM had the worst results, it has a great benefit – it is the easiest to interpret.

On the other hand, from modeller's perspective, XGBoost was the easiest to work with due to the fact that it did not require missing values to be substituted nor the categorical variables to be regrouped. It also did not expect the modeller to choose which variables to use in the modelling, but found the best variables by itself instead.

Conclusion

The purpose of this master's thesis was to provide an overview of the XGBoost algorithm and examine its suitability to model the claim frequency of motor third party liability insurance.

In the first chapter, the structure of generalized linear models was introduced. The steps of modelling were explained, including choosing a distribution for modelling count data and a proper link function. The second chapter gave a short overview of the generalized additive models and explained the local scoring procedure, used to fit GAM models.

The third chapter covered the basics of decision trees – how to grow a regression or a classification tree and when to stop. Later, some of the most common boosting methods were explained – Adaboost, Gradient boosting and XGBoost.

In the last chapter, generalized linear models, generalized additive models and XGBoost were applied to a dataset provided by the Estonian Motor Insurance Bureau to model motor third party liability insurance claim frequency. Two models were created with each technique - one by using the entire dataset and the other by using only numeric variables.

The data was preprocessed and divided into a training, validation and test set by using Python programming language. XGBoost models were also created by using Python, but R programming language was preferred when modelling GLM and GAM models. Later, the performances of the models on the test set were compared by using Poisson deviance as a loss function.

The best result was achieved with the XGBoost model that was trained by using all of the variables. The second best model was XGBoost that used only numeric variables. Next were GAM models and the worst ones were GLM models. When comparing a model with only numeric variables and a model with all variables, the latter model was better for all model types.

References

Chen, T., Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). New York: Association for Computing Machinery.

Decision Trees (n. d). Retrieved on 14.01.2020 from https://scikit-learn.org/stable/modules/tree.html

Flach, P. (2012). *Machine learning: the art and science of algorithms that make sense of data*. Cambridge: Cambridge University Press

Friedman, J. H. (1999). Stochastic gradient boosting. *Computational Statistics & Data Analysis, 38(4), 367-378.* https://doi.org/10.1016/S0167-9473(01)00065-2

Friedman, J., Hastie, T., Tibshirani, R. (2000). Additive Logistic Regression: A Statistical View of Boosting *The Annals of Statistics, 28(2), 337-407.* doi:10.1214/aos/1016120463

Friedman, J. H. (2001). Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics, 29(5), 1189-1232.* doi:10.1214/aos/1013203451

Hardin, J. W., Hilbe, J. M. (2007). *Generalized linear models and extensions*(2nd edition). Texas: Stata Press.

Hastie, T. J., Tibshirani, R. J. (1990). *Generalized additive models.* London: Chapman and Hall

Hastie, T., Tibshirani, R., Friedman, J. (2009). *The elements of statistical learning: data mining, inference, and prediction.* New York: Springer.

James, G., Witten, D., Hastie, T., Tibshirani, R. (2013). *An Introduction to Statistical Learning with Applications in R.* New York: Springer.

de Jong, P., Heller, G. Z. (2008). *Generalized linear models for insurance data.* New York: Cambridge University Press.

Kelleher, J. D., Namee, B. M., D'Arcy, A. (2015). *Fundamentals of machine learning for predictive data analysis: algorithms, working examples, and case studies.* Cambridge: The MIT Press.

Kohustuslik liikluskindlustus. (n.d.). Retrieved on 25.04.2020 from: https://www.lkf.ee/et/kindlustamise-tavad/kohustuslik-liikluskindlustus

Lamport, L. (1986). LATEX: A Document Preparation System, Adison. Wesley Publishing Co., Reading. Ma.

Lindsey, J. K. (1997). *Applying generalized linear models.* New York: Springer.

Lumley, T. (2013). biglm: bounded memory linear and generalized linear models. R package version 0.9-1. https://CRAN.R-project.org/package=biglm

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective.* Cambridge: The MIT Press.

R Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org

Rodríguez, G. (2020). *The Poisson Distribution* Retrieved on 11.04.2020 from: https://data.princeton.edu/wws509/notes/a2s5

Van Rossum, G., Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.

RStudio Team (2015). RStudio: Integrated Development for R. RStudio, Inc., Boston, MA. http://www.rstudio.com

Ruder, S. (2016). *An overview of gradient descent optimization algorithms.* Retrieved on 11.03.2020 from https://ruder.io/optimizing-gradient-descent/index.html#gradientdescentvariants

Schapire, R. E. (2013). *Explaining AdaBoost.* In: Schölkopf B., Luo Z., Vovk V. (eds) *Empirical inference.* Berlin: Springer.

Weisstein, E. W. (n.d.). *Smooth function.* Retrieved on 12.04.2020 from https://mathworld.wolfram.com/SmoothFunction.html

Wood, S.N. (2017) Generalized Additive Models: An Introduction with R (2nd edition). Chapman and Hall/CRC.

Appendices

Appendix 1. Description of the dataset

Below are the values of some variables that were used for modelling:

1. **the type of the owner**

Type	Count	Percentage
A person	6 327 087	68.43 %
A legal entity	2 918 497	31.57 %

2. **owner's gender**

Gender	Count	Percentage
Male	4 325 108	46.78 %
N/A	2 939 931	31.80 %
Female	1 980 545	21.42 %

3. **owner's age**

Min	18
Max	95
Mean	44.6
Std	15.2
N/A	2 939 958 (31.80 %)

4. **the type of the policy holder**

Type	Count	Percentage
A person	7 525 043	81.39 %
A legal entity	1 720 541	18.61 %

5. **policy holder's gender**

Gender	Count	Percentage
Male	5 196 411	56.20 %
N/A	2 315 294	25.04 %
Female	1 733 879	18.75 %

6. **policy holder's age**

Min	18
Max	95
Mean	44.0
Std	14.4
N/A	1 734 018 (18.76 %)

7. **the type of the responsible user**

Type	Count	Percentage
N/A	6 735 054	72.85 %
A person	1 846 545	19.97 %
A legal entity	663 985	7.18 %

8. **responsible user's gender**

Gender	Count	Percentage
N/A	7 399 205	80.03 %
Male	1 245 036	13.47 %
Female	601 343	6.50 %

9. **responsible user's age**

Min	18
Max	95
Mean	43.1
Std	11.7
N/A	7 399 205 (80.03 %)

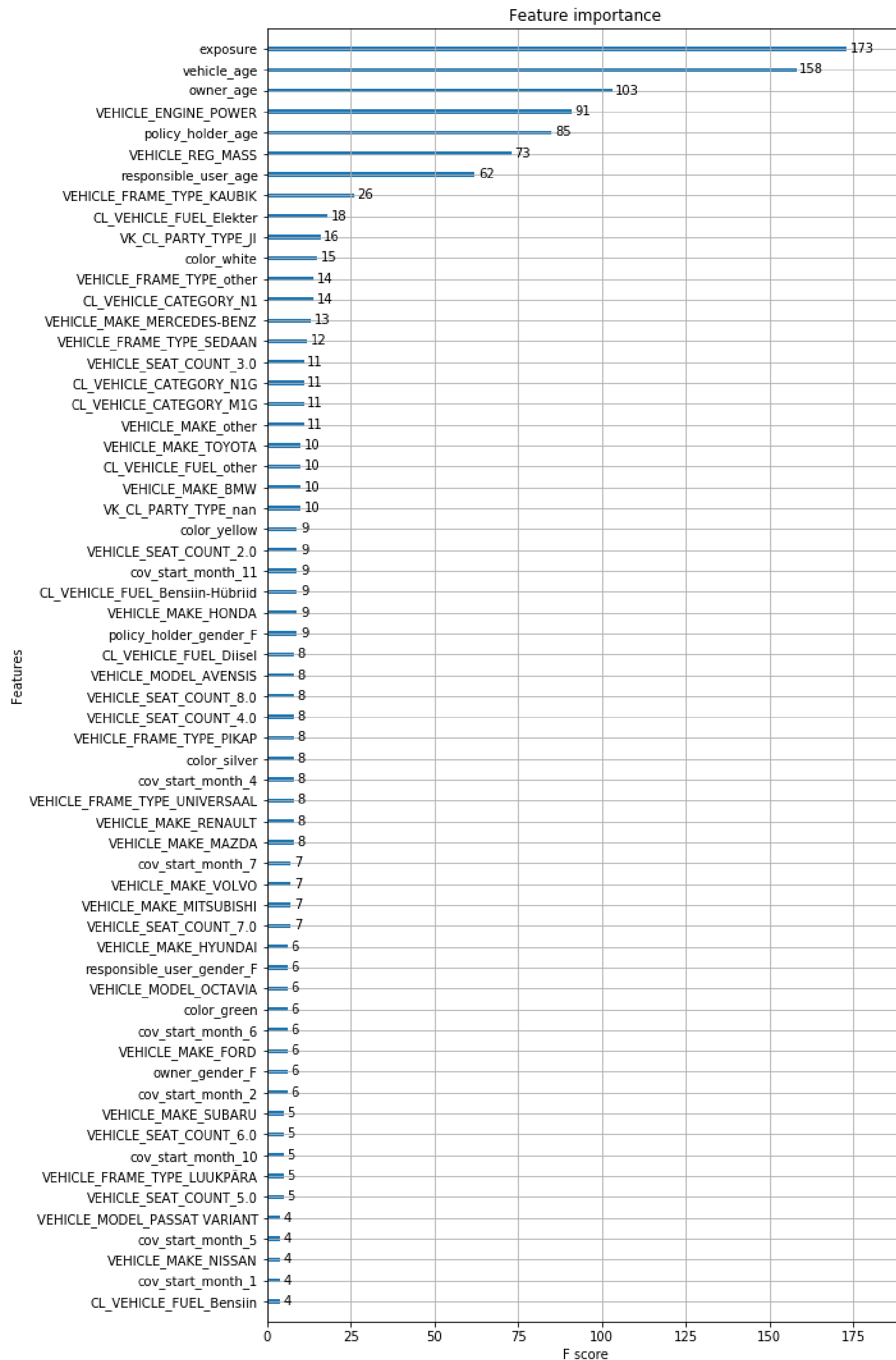
10. **exposure**

Min	0.003
Max	1.08
Mean	0.49
Std	0.37
N/A	0 (0 %)

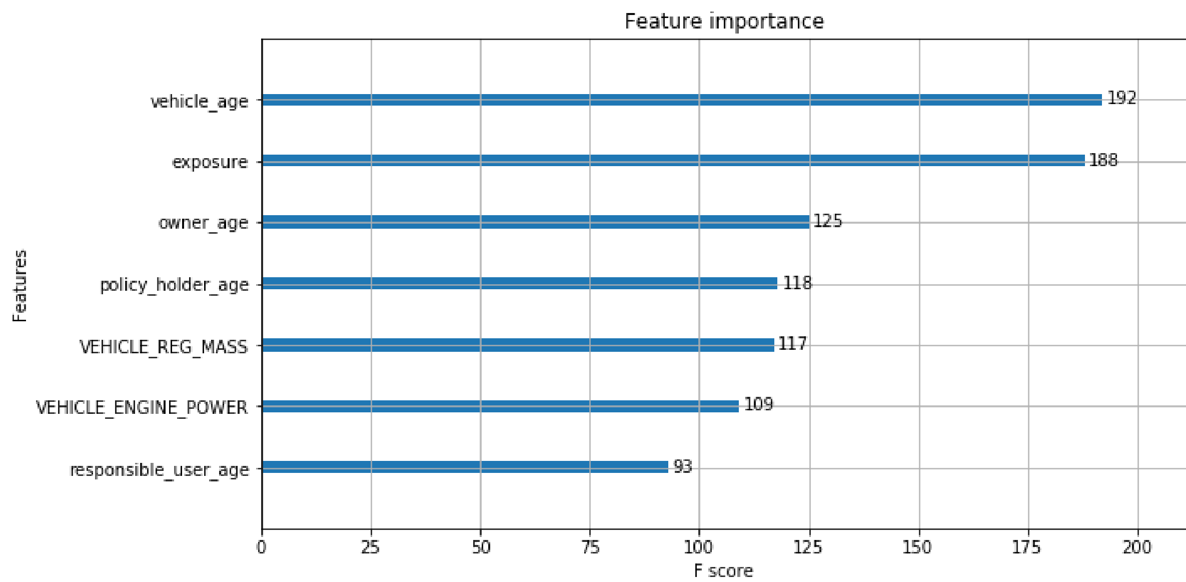
11. **number of claims**

Min	0
Max	7
Mean	0.01
Std	0.12
N/A	0 (0 %)

Appendix 2. The top feature importances of the first XGBoost model



Appendix 3. The feature importances of the second XGBoost model with numeric variables



Appendix 4. The output of the first GLM model

```
Large data regression model: bigglm(terms(count ~ cov_start_month + owner_gender + policy_holder_gender +
    policy_holder_age + responsible_user_gender + responsible_user_age +
    VEHICLE_MAKE + VEHICLE_MODEL + vehicle_age + CL_VEHICLE_CATEGORY +
    VEHICLE_ENGINE_POWER + VEHICLE_REG_MASS + VEHICLE_SEAT_COUNT +
    VEHICLE_FRAME_TYPE + color + CL_VEHICLE_FUEL + exposure +
    offset(log(exposure)), data = X_train_GLM), data = X_train_GLM,
    family = poisson(), maxit = 30)
Sample size =  5547350
                                         Coef    (95%     CI)     SE      p
(Intercept)                           -2.6428 -2.7164 -2.5691 0.0368 0.0000
cov_start_month11                      0.0415  0.0160  0.0669 0.0127 0.0011
cov_start_month3_4_5_6_7              -0.0468 -0.0615 -0.0322 0.0073 0.0000
owner_genderLegal                     -0.0298 -0.0668  0.0073 0.0185 0.1078
owner_genderM                         -0.0527 -0.0839 -0.0215 0.0156 0.0007
policy_holder_genderLegal             -0.2049 -0.2474 -0.1623 0.0213 0.0000
policy_holder_genderM                 -0.1068 -0.1356 -0.0780 0.0144 0.0000
policy_holder_age                     -0.0054 -0.0060 -0.0048 0.0003 0.0000
responsible_user_genderYes_r_user      0.2897  0.2623  0.3171 0.0137 0.0000
responsible_user_age                  -0.0042 -0.0048 -0.0035 0.0003 0.0000
VEHICLE_MAKEGroup_2                   -0.1205 -0.1409 -0.1001 0.0102 0.0000
VEHICLE_MAKEGroup_3                   -0.1861 -0.2064 -0.1658 0.0101 0.0000
VEHICLE_MAKEGroup_4                   -0.2500 -0.2930 -0.2070 0.0215 0.0000
VEHICLE_MODELGroup_2                   0.0529  0.0166  0.0892 0.0182 0.0036
vehicle_age                           -0.0127 -0.0142 -0.0112 0.0007 0.0000
CL_VEHICLE_CATEGORYN1_N1G              0.2644  0.1646  0.3642 0.0499 0.0000
VEHICLE_ENGINE_POWER                   0.0003  0.0001  0.0005 0.0001 0.0046
VEHICLE_REG_MASS                       0.0002  0.0002  0.0002 0.0000 0.0000
VEHICLE_SEAT_COUNT4_5_other           -0.1743 -0.2191 -0.1296 0.0224 0.0000
VEHICLE_SEAT_COUNT6_7                 -0.2190 -0.2671 -0.1708 0.0241 0.0000
VEHICLE_SEAT_COUNT8_9                 -0.3913 -0.4627 -0.3199 0.0357 0.0000
VEHICLE_FRAME_TYPEPIKAP_KAUBIK_other  -0.4725 -0.5713 -0.3736 0.0494 0.0000
VEHICLE_FRAME_TYPESEDAAN_LUUKP         0.0985  0.0821  0.1148 0.0082 0.0000
colorred_brown_green                  -0.0368 -0.0542 -0.0194 0.0087 0.0000
colorwhite_yellow                      0.0631  0.0389  0.0872 0.0121 0.0000
CL_VEHICLE_FUELbensiin                -0.4352 -0.5129 -0.3576 0.0388 0.0000
CL_VEHICLE_FUELelekter_other           0.5432  0.4476  0.6388 0.0478 0.0000
exposure                              -0.7644 -0.7870 -0.7419 0.0113 0.0000
```

Appendix 5. The output of the second GLM model with numeric variables

```
Large data regression model: bigglm(terms(count ~ policy_holder_age + responsible_user_age +
    vehicle_age + VEHICLE_ENGINE_POWER + VEHICLE_REG_MASS + exposure +
    offset(log(exposure)), data = X_train_GLM), data = X_train_GLM,
    family = poisson(), maxit = 30)
Sample size =  5547350
                        Coef    (95%     CI)     SE      p
(Intercept)          -2.8118 -2.8549 -2.7687 0.0216 0.0000
policy_holder_age    -0.0052 -0.0055 -0.0049 0.0002 0.0000
responsible_user_age  0.0007  0.0003  0.0011 0.0002 0.0007
vehicle_age          -0.0186 -0.0199 -0.0173 0.0006 0.0000
VEHICLE_ENGINE_POWER  0.0011  0.0009  0.0012 0.0001 0.0000
VEHICLE_REG_MASS      0.0001  0.0001  0.0001 0.0000 0.0000
exposure             -0.8133 -0.8350 -0.7917 0.0108 0.0000
```

Appendix 6. The output of the first GAM model

```
Parametric coefficients:
                                    Estimate Std. Error z value             Pr(>|z|)    
(Intercept)                        -2.205887   0.106873 -20.640 < 0.0000000000000002 ***
cov_start_month11                   0.040723   0.012754   3.193             0.001409 ** 
cov_start_month3_4_5_6_7           -0.042466   0.007395  -5.742   0.0000000093387347 ***
owner_genderLegal                  -0.007456   0.019242  -0.387             0.698390    
owner_genderM                      -0.068800   0.015652  -4.396  0.00001104270501624 ***
policy_holder_genderLegal          -4.410750   0.560743  -7.866  0.00000000000000366 ***
policy_holder_genderM              -0.115132   0.014423  -7.982  0.00000000000000144 ***
responsible_user_genderYes_r_user   0.327415   0.014643  22.361 < 0.0000000000000002 ***
VEHICLE_MAKEGroup_2                -0.102027   0.010282  -9.922 < 0.0000000000000002 ***
VEHICLE_MAKEGroup_3                -0.164581   0.010205 -16.128 < 0.0000000000000002 ***
VEHICLE_MAKEGroup_4                -0.230025   0.021609 -10.645 < 0.0000000000000002 ***
VEHICLE_MODELGroup_2                0.050151   0.018456   2.717             0.006581 ** 
CL_VEHICLE_CATEGORYN1_N1G           0.186775   0.050843   3.674             0.000239 ***
VEHICLE_SEAT_COUNT4_5_other        -0.166325   0.022858  -7.276  0.00000000000034261 ***
VEHICLE_SEAT_COUNT6_7              -0.194371   0.024808  -7.835  0.00000000000000469 ***
VEHICLE_SEAT_COUNT8_9              -0.375226   0.036724 -10.218 < 0.0000000000000002 ***
VEHICLE_FRAME_TYPEPIKAP_KAUBIK_other -0.416007 0.049293  -8.439 < 0.0000000000000002 ***
VEHICLE_FRAME_TYPESEDAAN_LUUKP      0.082153   0.008840   9.293 < 0.0000000000000002 ***
colorred_brown_green               -0.019901   0.008813  -2.258             0.023938 *  
colorwhite_yellow                   0.085369   0.012275   6.955  0.00000000000352651 ***
CL_VEHICLE_FUELbensiin             -0.157278   0.043810  -3.590             0.000331 ***
CL_VEHICLE_FUELelekter_other        0.660071   0.048442  13.626 < 0.0000000000000002 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:
                          edf Ref.df  Chi.sq              p-value    
s(policy_holder_age)   11.232     14 2132.44 < 0.0000000000000002 ***
s(responsible_user_age) 3.735      7  224.64 < 0.0000000000000002 ***
s(vehicle_age)          5.742      7  685.91 < 0.0000000000000002 ***
s(VEHICLE_ENGINE_POWER) 6.426      7   69.99    0.000000000000104 ***
s(VEHICLE_REG_MASS)     4.941      7  291.07 < 0.0000000000000002 ***
s(exposure)            16.817     19 5782.92 < 0.0000000000000002 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Appendix 7. The output of the second GAM model with numeric variables

```
Family: poisson
Link function: log

Formula:
count ~ s(policy_holder_age, k = 10, bs = "cs") + s(responsible_user_age,
    k = 5, bs = "cs") + s(vehicle_age, k = 5, bs = "cs") + s(VEHICLE_ENGINE_POWER,
    k = 5, bs = "cs") + s(VEHICLE_REG_MASS, k = 5, bs = "cs") +
    s(exposure, k = 10, bs = "cs") + offset(log(exposure))

Parametric coefficients:
             Estimate Std. Error z value             Pr(>|z|)
(Intercept) -3.29849    0.00459  -718.7 <0.0000000000000002 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
                          edf Ref.df Chi.sq              p-value
s(policy_holder_age)    8.414      9 2524.6 <0.0000000000000002 ***
s(responsible_user_age) 3.593      4  103.0 <0.0000000000000002 ***
s(vehicle_age)          3.823      4 1086.0 <0.0000000000000002 ***
s(VEHICLE_ENGINE_POWER) 3.649      4  203.3 <0.0000000000000002 ***
s(VEHICLE_REG_MASS)     3.908      4  260.0 <0.0000000000000002 ***
s(exposure)             8.903      9 5867.5 <0.0000000000000002 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Appendix 8. The code for preprocessing the data

```

#Importing packages
import time
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

#Changing the number of displayed columns and rows.
pd.set_option("display.max_columns", 50)
pd.set_option("display.max_rows", 1000)


#Importing covers' dataset and measuring the time it takes.
start_time = time.time()

#Importing the dataset in chunks to get better performance. Since Python
    cannot handle missing values for integer type variables, then all
   the numeric variables are imported as floats. Since variables cannot
   be imported as categorical variables with read_csv, then they are
   imported as "object" and later converted to categorical.
chunks_covers = pd.read_csv("covers.csv",
                        sep=";", encoding = "ANSI", chunksize=100000,
                        dtype = {"COVER_NO_HASH" : "object",
                        "COVER_DCP_START" : "object",
                        "COVER_DCP_END" : "object",
                        "OM_CL_PARTY_TYPE" : "object",
                        "OM_SSID_FIRST_7_NUMBERS" : "object",
                        "KV_CL_PARTY_TYPE" : "object",
                        "KV_SSID_FIRST_7_NUMBERS" : "object",
                        "VK_CL_PARTY_TYPE" : "object",
                        "VK_SSID_FIRST_7_NUMBERS" : "object",
                        "VEHICLE_MAKE" : "object",
                        "VEHICLE_MODEL" : "object",
                        "CL_VEHICLE_CATEGORY" : "object",
                        "VEHICLE_PRIME_REGISTRATION" : "float64",
                        "VEHICLE_ENGINE_POWER" : "float64",
```

```
33                              "VEHICLE_REG_MASS" : "float64",
34                              "VEHICLE_SEAT_COUNT" : "float64",
35                              "VEHICLE_SPEED_LIMIT" : "float64",
36                              "VEHICLE_FRAME_TYPE" : "object",
37                              "CL_VEHICLE_COLOR_CODE" : "float64",
38                              "CL_VEHICLE_FUEL" : "object"})
39
40 #Merging the chunks back together.
41 covers = pd.concat(chunks_covers)
42
43 #Converting cover start and end dates to datetime variables.
44 covers["COVER_DCP_START"] =  pd.to_datetime(covers["COVER_DCP_START"],
       format="%m.%d.%Y %H:%M:%S")
45 covers["COVER_DCP_END"] =  pd.to_datetime(covers["COVER_DCP_END"],
       format="%m.%d.%Y %H:%M:%S")
46
47 end_time = time.time()
48 print("%s seconds" % (end_time - start_time))
49
50 #It took 160.28 seconds to import the dataset.
51
52
53
54
55 #Importing claims' dataset and measuring the time it takes.
56 start_time = time.time()
57
58 #Importing the dataset in chunks to get better performance.
59 chunks_claims = pd.read_csv("claims.csv",
60                              sep=";", chunksize = 100000,
61                              dtype = {"COVER_NO_HASH" : "object",
62                              "ACC_DATE_START" : "object",
63                              "CL_ACC_COUNTRY" : "object",
64                              "CL_EHAK" : "object",
65                              "JH_SSID_FIRST_3_NUMBERS" : "object"})
66
67 #Merging the chunks back together.
```

```
68 claims = pd.concat(chunks_claims)
69
70 #Converting accident date to datetime variable.
71 claims["ACC_DATE_START"] =  pd.to_datetime(claims["ACC_DATE_START"],
    format="%m.%d.%Y %H:%M:%S")
72
73 end_time = time.time()
74 print("%s seconds" % (end_time - start_time))
75
76 #It took 1.55 seconds to import the dataset.
77
78
79
80 #Finding the nr of claims related to each policy ID by grouping by
    policy ID and then finding the number of rows in each group.
81
82 claim_counts = claims.groupby("COVER_NO_HASH").size().reset_index(name='
    counts')
83
84 #Creating a new dataset by left joining covers' dataset to the number of
     claims on policy ID.
85 data = pd.merge(covers, claim_counts, how='left', on=["COVER_NO_HASH"])
86
87 #Setting the number of claims to 0 for all policies that have N/A value
    for the number of claims.
88 data["counts"] = data["counts"].fillna(0)
89
90
91 #Checking if sum of claim counts is the same as the number of rows in
    claims' table.
92 sum_of_counts = data["counts"].sum(axis = 0, skipna = True)
93 print(sum_of_counts)
94
95 nr_of_rows_claims = claims.shape[0]
96 print(nr_of_rows_claims)
97
98
```

```
99  #Extracting the years of cover start and end dates to understand how the
        policies are distributed between years
100 data["cov_start_year"] = pd.DatetimeIndex(data["COVER_DCP_START"]).year
101 data["cov_end_year"] = pd.DatetimeIndex(data["COVER_DCP_END"]).year
102
103
104 #Extracting the month of cover start.
105 data["cov_start_month"] = pd.DatetimeIndex(data["COVER_DCP_START"]).
       month
106
107 #Calculating exposure in days.
108 data["exposure"]= round((data["COVER_DCP_END"] - data["COVER_DCP_START"
       ])/np.timedelta64(1,'D'),0)
109
110 #Deleting the rows, where exposure is less than one day.
111 data = data[data.exposure >= 1]
112
113
114 #Extracting the first number of the policy holder's identification
       number.
115 data["owner_gender"] = data["OM_SSID_FIRST_7_NUMBERS"].str[0]
116
117
118 #Turning those identification numbers into NaN (not a number), where the
        personal identification number does not start
119 #with 3,4,5 or 6 or where the the type of the policy holder is a legal
       entity.
120 condition_owner_gen = np.logical_or(np.logical_not((data.owner_gender ==
        "3") | (data.owner_gender == "4") |
121                                    (data.owner_gender == "5") | (data.
       owner_gender == "6")),
122                                    (data.OM_CL_PARTY_TYPE == "JI"))
123 data["OM_SSID_FIRST_7_NUMBERS"] = np.where(condition_owner_gen, np.nan ,
        data.OM_SSID_FIRST_7_NUMBERS)
124
125
```

```
126 #Extracting the first number again in order to only keep numbers 3,4,5 and 6.
127 data["owner_gender"] = data["OM_SSID_FIRST_7_NUMBERS"].str[0]
128
129
130 #Calculating the birthday of the owner.
131 data["owner_birth_day"] = data["OM_SSID_FIRST_7_NUMBERS"].str[1:7]
132 data["owner_birth_day"] = pd.to_datetime(data["owner_birth_day"], format="%y%m%d", errors= "coerce")
133
134
135 #Since the personal identification number only has two last digits of the year and python automatically turns
136 #values 69-99 to 1969-1999, and values 0  68  to 2000  2068  , then the year needs to be fixed for those instances,
137 #where birth year is after 2000, but the firts number of the personal identification number is 3 or 4.
138 condition_owner_bd = (pd.DatetimeIndex(data.owner_birth_day).year >= 2000) & ((data.owner_gender == "3") | (data.owner_gender == "4"))
139 data["owner_birth_day"] = np.where(condition_owner_bd, data.owner_birth_day - pd.DateOffset(years = 100), data.owner_birth_day)
140
141
142 #There are  some identification numbers that indicate the person was born on a future date (for example 5400101)
143 #or over a 100 years ago. It is assumed that those identificiation numbers are incorrect and the values of
144 #identification numbers, birth day and gender are changed to N/A.
145 condition_owner_bd_2 = ((pd.DatetimeIndex(data.owner_birth_day).year > 2018) | (pd.DatetimeIndex(data.owner_birth_day).year <= 1918))
146 data["OM_SSID_FIRST_7_NUMBERS"] = np.where(condition_owner_bd_2, np.nan , data.OM_SSID_FIRST_7_NUMBERS)
147 data["owner_birth_day"] = np.where(condition_owner_bd_2, np.datetime64("NaT") , data.owner_birth_day)
148 data["owner_gender"] = np.where(condition_owner_bd_2, np.nan , data.owner_gender)
149
```

```
150
151 #Calculating the age of the owner in years.
152 data["owner_age"] = round((data["COVER_DCP_START"] - data["
    owner_birth_day"] )/np.timedelta64(1,'Y'),0)
153
154 #In order to get rid of extreme values, those rows, where the age is
    less than 18 or more than 95, are turned to N/A.
155 condition_owner_bd_3 = ((data.owner_age < 18) | (data.owner_age > 95))
156 data["OM_SSID_FIRST_7_NUMBERS"] = np.where(condition_owner_bd_3, np.nan
    , data.OM_SSID_FIRST_7_NUMBERS)
157 data["owner_birth_day"] = np.where(condition_owner_bd_3, np.datetime64("
    NaT") , data.owner_birth_day)
158 data["owner_gender"] = np.where(condition_owner_bd_3, np.nan , data.
    owner_gender)
159 data["owner_age"] = np.where(condition_owner_bd_3, np.nan , data.
    owner_age)
160
161
162 #Extracting the first number of the policy holder's identification
    number.
163 data["policy_holder_gender"] = data["KV_SSID_FIRST_7_NUMBERS"].str[0]
164
165
166 #Turning those identification numbers into NaN (not a number), where the
     personal identification number does not start
167 #with 3,4,5 or 6 or where the the type of the policy holder is a legal
    entity.
168 condition_p_h_gen = np.logical_or(np.logical_not((data.
    policy_holder_gender == "3") | (data.policy_holder_gender == "4") |
169                                   (data.policy_holder_gender == "5") |
    (data.policy_holder_gender == "6")),
170                                   (data.KV_CL_PARTY_TYPE == "JI"))
171 data["KV_SSID_FIRST_7_NUMBERS"] = np.where(condition_p_h_gen, np.nan ,
    data.KV_SSID_FIRST_7_NUMBERS)
172
173
```

```
174 #Extracting the first number again in order to only keep numbers 3,4,5 and 6.
175 data["policy_holder_gender"] = data["KV_SSID_FIRST_7_NUMBERS"].str[0]
176
177
178 #Calculating the birthday of the policy holder.
179 data["policy_holder_birth_day"] = data["KV_SSID_FIRST_7_NUMBERS"].str[1:7]
180 data["policy_holder_birth_day"] = pd.to_datetime(data["policy_holder_birth_day"], format="%y%m%d", errors= "coerce")
181
182
183 #Since the personal identification number only has two last digits of the year and python automatically turns
184 #values 69-99 to 1969-1999, and values 0 68  to 2000 2068 , then the year needs to be fixed for those instances,
185 #where birth year is after 2000, but the firts number of the personal identification number is 3 or 4.
186 condition_p_h_bd = (pd.DatetimeIndex(data.policy_holder_birth_day).year >= 2000) & ((data.policy_holder_gender == "3") | (data.policy_holder_gender == "4"))
187 data["policy_holder_birth_day"] = np.where(condition_p_h_bd, data.policy_holder_birth_day - pd.DateOffset(years = 100), data.policy_holder_birth_day)
188
189
190 #There are  some identification numbers that indicate the person was born on a future date (for example 5400101)
191 #or over a 100 years ago. It is assumed that those identificiation numbers are incorrect and the values of
192 #identification numbers, birth day and gender are changed to N/A.
193 condition_p_h_bd_2 = (pd.DatetimeIndex(data.policy_holder_birth_day).year > 2018) | (pd.DatetimeIndex(data.policy_holder_birth_day).year <= 1918)
194 data["KV_SSID_FIRST_7_NUMBERS"] = np.where(condition_p_h_bd_2, np.nan , data.KV_SSID_FIRST_7_NUMBERS)
```

```
195 data["policy_holder_birth_day"] = np.where(condition_p_h_bd_2, np.
        datetime64("NaT") , data.policy_holder_birth_day)
196 data["policy_holder_gender"] = np.where(condition_p_h_bd_2, np.nan ,
        data.policy_holder_gender)
197
198
199 #Calculating policy holder's age
200 data["policy_holder_age"] = round((data["COVER_DCP_START"] - data["
        policy_holder_birth_day"]  )/np.timedelta64(1,'Y'),0)
201
202
203 #In order to get rid of extreme values, those rows, where the age is
        less than 18 or more than 95, are turned to N/A.
204 condition_p_h_bd_3 = ( data.policy_holder_age < 18 ) | ( data.
        policy_holder_age > 95 )
205 data["KV_SSID_FIRST_7_NUMBERS"] = np.where(condition_p_h_bd_3, np.nan ,
        data.KV_SSID_FIRST_7_NUMBERS)
206 data["policy_holder_birth_day"] = np.where(condition_p_h_bd_3, np.
        datetime64("NaT") , data.policy_holder_birth_day)
207 data["policy_holder_gender"] = np.where(condition_p_h_bd_3, np.nan ,
        data.policy_holder_gender)
208 data["policy_holder_age"] = np.where(condition_p_h_bd_3, np.nan , data.
        policy_holder_age)
209
210
211 #Extracting the first number of the responsible user's identification
        number.
212 data["responsible_user_gender"] = data["VK_SSID_FIRST_7_NUMBERS"].str[0]
213
214 #Turning those identification numbers into NaN (not a number), where the
         personal identification number does not start
215 #with 3,4,5 or 6 or where the the type of the responsible user is a
        legal entity.
216 condition_r_u_gen = np.logical_or(np.logical_not((data.
        responsible_user_gender == "3") | (data.responsible_user_gender == "4
        ") |
```

```python
217                                     (data.responsible_user_gender == "5") | (data.responsible_user_gender == "6")),
218                                     (data.VK_CL_PARTY_TYPE == "JI"))
219 data["VK_SSID_FIRST_7_NUMBERS"] = np.where(condition_r_u_gen, np.nan , data.VK_SSID_FIRST_7_NUMBERS)
220
221
222 #Extracting the first number again in order to only keep numbers 3,4,5 and 6.
223 data["responsible_user_gender"] = data["VK_SSID_FIRST_7_NUMBERS"].str[0]
224
225
226 #Calculating the birthday of the responsible user.
227 data["responsible_user_birth_day"] = data["VK_SSID_FIRST_7_NUMBERS"].str[1:7]
228 data["responsible_user_birth_day"] = pd.to_datetime(data["responsible_user_birth_day"], format="%y%m%d", errors= "coerce")
229
230
231 #Since the personal identification number only has two last digits of the year and python automatically turns
232 #values 69-99 to 1969-1999, and values 0 68  to 2000 2068  , then the year needs to be fixed for those instances,
233 #where birth year is after 2000, but the firts number of the personal identification number is 3 or 4.
234 condition_r_u_bd = (pd.DatetimeIndex(data.responsible_user_birth_day).year >= 2000) & ((data.responsible_user_gender == "3") | (data.responsible_user_gender == "4"))
235 data["responsible_user_birth_day"] = np.where(condition_r_u_bd, data.responsible_user_birth_day - pd.DateOffset(years = 100), data.responsible_user_birth_day)
236
237
238 #There are  some identification numbers that indicate the person was born on a future date (for example 5400101)
239 #or over a 100 years ago. It is assumed that those identificiation numbers are incorrect and the values of
```

```
240 #identification numbers, birth day and gender are changed to N/A.
241 condition_responsible_user_bd_2 = (pd.DatetimeIndex(data.
       responsible_user_birth_day).year > 2018) | (pd.DatetimeIndex(data.
       responsible_user_birth_day).year <= 1918)
242 data["VK_SSID_FIRST_7_NUMBERS"] = np.where(
       condition_responsible_user_bd_2, np.nan , data.
       VK_SSID_FIRST_7_NUMBERS)
243 data["responsible_user_birth_day"] = np.where(
       condition_responsible_user_bd_2, np.datetime64("NaT") , data.
       responsible_user_birth_day)
244 data["responsible_user_gender"] = np.where(
       condition_responsible_user_bd_2, np.nan , data.
       responsible_user_gender)
245
246
247 #Calculating responsible user's age
248 data["responsible_user_age"] = round(( data["COVER_DCP_START"] - data["
       responsible_user_birth_day"]  )/np.timedelta64(1,'Y'),0)
249
250
251 #In order to get rid of extreme values, those rows, where the age is
       less than 18 or more than 95, are turned to N/A.
252 condition_responsible_user_bd_3 = (data.responsible_user_age <18) | (
       data.responsible_user_age >95)
253 data["VK_SSID_FIRST_7_NUMBERS"] = np.where(
       condition_responsible_user_bd_3, np.nan , data.
       VK_SSID_FIRST_7_NUMBERS)
254 data["responsible_user_birth_day"] = np.where(
       condition_responsible_user_bd_3, np.datetime64("NaT") , data.
       responsible_user_birth_day)
255 data["responsible_user_gender"] = np.where(
       condition_responsible_user_bd_3, np.nan , data.
       responsible_user_gender)
256 data["responsible_user_age"] = np.where(condition_responsible_user_bd_3,
        np.nan , data.responsible_user_age)
257
258
```

```
259
260 #Checking the minimum and maximum values of all birthdays.
261
262 print(data["owner_birth_day"].min())
263 print(data["owner_birth_day"].max())
264
265
266 print(data["policy_holder_birth_day"].min())
267 print(data["policy_holder_birth_day"].max())
268
269 print(data["responsible_user_birth_day"].min())
270 print(data["responsible_user_birth_day"].max())
271
272
273 #Converting the date of vehicle registration to datetime.
274 data["VEHICLE_PRIME_REGISTRATION"] = pd.to_datetime(data["
       VEHICLE_PRIME_REGISTRATION"], format="%Y", errors= "coerce")
275
276 #Calculating the age of the vehicle in years.
277 data["vehicle_age"]= round((data["COVER_DCP_START"] - data["
       VEHICLE_PRIME_REGISTRATION"])/np.timedelta64(1,'Y'),0)
278
279
280 #All categorical variables were assigned the datatype "category".
281 data.OM_CL_PARTY_TYPE = data.OM_CL_PARTY_TYPE.astype("category")
282 data.KV_CL_PARTY_TYPE = data.KV_CL_PARTY_TYPE.astype("category")
283 data.VK_CL_PARTY_TYPE = data.VK_CL_PARTY_TYPE.astype("category")
284 data.VEHICLE_MAKE = data.VEHICLE_MAKE.astype("category")
285 data.VEHICLE_MODEL = data.VEHICLE_MODEL.astype("category")
286 data.CL_VEHICLE_CATEGORY = data.CL_VEHICLE_CATEGORY.astype("category")
287 data.VEHICLE_SEAT_COUNT = data.VEHICLE_SEAT_COUNT.astype("category")
288 data.VEHICLE_FRAME_TYPE = data.VEHICLE_FRAME_TYPE.astype("category")
289 data.CL_VEHICLE_COLOR_CODE = data.CL_VEHICLE_COLOR_CODE.astype("category
       ")
290 data.CL_VEHICLE_FUEL = data.CL_VEHICLE_FUEL.astype("category")
291 data.cov_start_month = data.cov_start_month.astype("category")
292 data.owner_gender = data.owner_gender.astype("category")
```

```
293 data.policy_holder_gender = data.policy_holder_gender.astype("category")
294 data.responsible_user_gender = data.responsible_user_gender.astype("
    category")
295 
296 #Regrouping the make of the vehicle - keeping 20 most popular values and
     N/A values and categorizing the rest as "other".
297 nr_of_cat = 20
298 keep_categories = data["VEHICLE_MAKE"].value_counts().head(nr_of_cat).
    index.tolist()
299 condition_cat_vehicle_make = np.bitwise_not((data["VEHICLE_MAKE"].isin(
    keep_categories)) | (data["VEHICLE_MAKE"].isnull()))
300 data["VEHICLE_MAKE"] = np.where(condition_cat_vehicle_make, "other" ,
    data.VEHICLE_MAKE)
301 
302 
303 #Regrouping the model of the vehicle - keeping 5 most popular values and
     N/A values and categorizing the rest as "other".
304 nr_of_cat = 5
305 keep_categories = data["VEHICLE_MODEL"].value_counts().head(nr_of_cat).
    index.tolist()
306 condition_cat_vehicle_model = np.bitwise_not((data["VEHICLE_MODEL"].isin
    (keep_categories)) | (data["VEHICLE_MODEL"].isnull()))
307 data["VEHICLE_MODEL"] = np.where(condition_cat_vehicle_model, "other" ,
    data.VEHICLE_MODEL)
308 
309 
310 #Deleting those 4 rows, where the engine power is 0 kW.
311 data = data[data["VEHICLE_ENGINE_POWER"] > 0]
312 
313 
314 #Regrouping the seat count of the vehicle - keeping 8 most popular
    values and N/A values and categorizing the
315 #rest as "other".
316 nr_of_cat = 8
317 keep_categories = data["VEHICLE_SEAT_COUNT"].value_counts().head(
    nr_of_cat).index.tolist()
```

```
318 condition_cat_vehicle_seat_count = np.bitwise_not((data["
        VEHICLE_SEAT_COUNT"].isin(keep_categories)) | (data["
        VEHICLE_SEAT_COUNT"].isnull()))
319 data["VEHICLE_SEAT_COUNT"] = np.where(condition_cat_vehicle_seat_count,
        "other" , data.VEHICLE_SEAT_COUNT)
320
321
322 #Regrouping the frame type of the vehicle - keeping 7 most popular
        values and N/A values and categorizing the
323 #rest as "other".
324 nr_of_cat = 7
325 keep_categories = data["VEHICLE_FRAME_TYPE"].value_counts().head(
        nr_of_cat).index.tolist()
326 condition_cat_vehicle_frame_type = np.bitwise_not((data["
        VEHICLE_FRAME_TYPE"].isin(keep_categories)) | (data["
        VEHICLE_FRAME_TYPE"].isnull()))
327 data["VEHICLE_FRAME_TYPE"] = np.where(condition_cat_vehicle_frame_type,
        "other" , data.VEHICLE_FRAME_TYPE)
328
329
330 #Regrouping the colors by uniting different variations of the colors (e.
        g. light blue, blue and dark blue were
331 #categorized as blue).
332
333 #white
334 condition_color_white = (data.CL_VEHICLE_COLOR_CODE == 0)
335
336 #light grey, grey, dark grey
337 condition_color_grey = ((data.CL_VEHICLE_COLOR_CODE == 10) | (data.
        CL_VEHICLE_COLOR_CODE == 11) |
338                         (data.CL_VEHICLE_COLOR_CODE == 19))
339
340 #light brown, light beige, brown, beige, dark brown, golden
341 condition_color_brown = ((data.CL_VEHICLE_COLOR_CODE == 20) | (data.
        CL_VEHICLE_COLOR_CODE == 21) |
342                         (data.CL_VEHICLE_COLOR_CODE == 22) | (data.
        CL_VEHICLE_COLOR_CODE == 28) |
```

```
343                             (data.CL_VEHICLE_COLOR_CODE == 29) | (data.
       CL_VEHICLE_COLOR_CODE == 95))
344
345 #light red, red, dark red, pink
346 condition_color_red = ((data.CL_VEHICLE_COLOR_CODE == 30) | (data.
       CL_VEHICLE_COLOR_CODE == 33) |
347                         (data.CL_VEHICLE_COLOR_CODE == 39) | (data.
       CL_VEHICLE_COLOR_CODE == 93))
348
349 #orange, light yellow, yellow, dark yellow
350 condition_color_yellow = ((data.CL_VEHICLE_COLOR_CODE == 44) | (data.
       CL_VEHICLE_COLOR_CODE == 50) |
351                         (data.CL_VEHICLE_COLOR_CODE == 55) | (data.
       CL_VEHICLE_COLOR_CODE == 59))
352
353 #light green, green, dark green
354 condition_color_green = ((data.CL_VEHICLE_COLOR_CODE == 60) | (data.
       CL_VEHICLE_COLOR_CODE == 66) |
355                         (data.CL_VEHICLE_COLOR_CODE == 69) )
356
357 #light blue, blue, dark blue, purple
358 condition_color_blue = ((data.CL_VEHICLE_COLOR_CODE == 70) | (data.
       CL_VEHICLE_COLOR_CODE == 77) |
359                         (data.CL_VEHICLE_COLOR_CODE == 79) | (data.
       CL_VEHICLE_COLOR_CODE == 83))
360
361 #silver
362 condition_color_silver = (data.CL_VEHICLE_COLOR_CODE == 91)
363
364 #black
365 condition_color_black = (data.CL_VEHICLE_COLOR_CODE == 99)
366
367 #not available values - some are coded as 90 and some are N/A.
368 condition_color_na = (data.CL_VEHICLE_COLOR_CODE == 90)
369 condition_color_na2 = (data.CL_VEHICLE_COLOR_CODE.isnull())
370
371
```

```
372 #creating a new variable for the new color grouping
373 data["color"] = np.nan
374 data["color"] = np.where(condition_color_white, "white" , data.color)
375 data["color"] = np.where(condition_color_grey, "grey" ,  data.color)
376 data["color"] = np.where(condition_color_brown, "brown" ,  data.color)
377 data["color"] = np.where(condition_color_red, "red" ,  data.color)
378 data["color"] = np.where(condition_color_yellow, "yellow" ,  data.color)
379 data["color"] = np.where(condition_color_green, "green" ,  data.color)
380 data["color"] = np.where(condition_color_blue, "blue" ,  data.color)
381 data["color"] = np.where(condition_color_silver, "silver" ,  data.color)
382 data["color"] = np.where(condition_color_black, "black" ,  data.color)
383 data["color"] = np.where(condition_color_na, np.nan ,  data.color)
384 data["color"] = np.where(condition_color_na2, np.nan ,  data.color)
385
386 #Changing the datatype to categorical.
387 data.color = data.color.astype("category")
388
389
390 #Regrouping the fuel type of the vehicle - keeping 5 most popular values
        and N/A values and categorizing the
391 #rest as "other".
392 nr_of_cat = 5
393 keep_categories = data["CL_VEHICLE_FUEL"].value_counts().head(nr_of_cat)
       .index.tolist()
394 condition_cat_vehicle_fuel = np.bitwise_not((data["CL_VEHICLE_FUEL"].
       isin(keep_categories)) | (data["CL_VEHICLE_FUEL"].isnull()))
395 data["CL_VEHICLE_FUEL"] = np.where(condition_cat_vehicle_fuel, "other" ,
        data.CL_VEHICLE_FUEL)
396
397
398 #Changing the value of the owner's gender from 3 and 5 to "M" (male) and
        from "4" and "6" to "F" (female).
399 data["owner_gender"] = np.where((data.owner_gender == "3") | (data.
       owner_gender == "5") , "M" , data.owner_gender)
400 data["owner_gender"] = np.where((data.owner_gender == "4") | (data.
       owner_gender == "6") , "F" , data.owner_gender)
401
```

```
402
403 #Changing the value of the policy holder's gender from 3 and 5 to "M" (
       male) and from "4" and "6" to "F" (female).
404 data["policy_holder_gender"] = np.where((data.policy_holder_gender == "3
       ") | (data.policy_holder_gender == "5") , "M" , data.
       policy_holder_gender)
405 data["policy_holder_gender"] = np.where((data.policy_holder_gender == "4
       ") | (data.policy_holder_gender == "6") , "F" , data.
       policy_holder_gender)
406
407
408 #Changing the value of the responsible user's gender from 3 and 5 to "M"
        (male) and from "4" and "6" to "F" (female).
409 data["responsible_user_gender"] = np.where((data.responsible_user_gender
        == "3") | (data.responsible_user_gender == "5") , "M" , data.
       responsible_user_gender)
410 data["responsible_user_gender"] = np.where((data.responsible_user_gender
        == "4") | (data.responsible_user_gender == "6") , "F" , data.
       responsible_user_gender)
411
412
413 #Classifying all vehicle ages over 40 as "older than 40". This is
       denoted by 41 in order to keep the variable type numeric.
414 condition_vehicle_age = (data["vehicle_age"] > 40) & (np.bitwise_not(
       data["vehicle_age"].isnull()))
415 data["vehicle_age"] = np.where(condition_vehicle_age, 41 , data.
       vehicle_age)
416
417
418 #Keeping only those variables that are going to be used for modelling.
419 data_final = data[["cov_start_month",
420                    "OM_CL_PARTY_TYPE", "owner_gender", "owner_age",
421                    "KV_CL_PARTY_TYPE", "policy_holder_gender", "
       policy_holder_age",
422                    "VK_CL_PARTY_TYPE", "responsible_user_gender", "
       responsible_user_age",
```

```
423                     "VEHICLE_MAKE", "VEHICLE_MODEL", "vehicle_age", "
    CL_VEHICLE_CATEGORY",
424                      "VEHICLE_ENGINE_POWER", "VEHICLE_REG_MASS", "
    VEHICLE_SEAT_COUNT",
425                      "VEHICLE_FRAME_TYPE", "color", "CL_VEHICLE_FUEL",
426                      "counts", "exposure"]]
427
428
429 #Saving the dataset to a csv file.
430 data_final.to_csv("data_final.csv",
431                           index = False, sep = ";")
432
433
434 #Getting information about the numeric variables.
435 data_final.describe()
436
437
438 #Getting the value counts of all variables
439 for variable in data_final.columns:
440     print(variable)
441     print(data_final[variable].value_counts(dropna=False))
442     print("")
```

Appendix 9. The code for creating an XGBoost model

```
#Import packages
import time
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
pd.set_option("display.max_columns", 50)
pd.set_option("display.max_rows", 1000)
import xgboost as xgb
import sklearn
from datetime import datetime


#Import covers
start_time = time.time()

data_chunks = pd.read_csv("data_final.csv",
                          sep=";", chunksize=100000,
                          dtype = {"cov_start_month" : "object",
                          "OM_CL_PARTY_TYPE": "object",
                          "owner_gender" : "object",
                          "owner_age" : "float64",
                          "KV_CL_PARTY_TYPE" : "object" ,
                          "policy_holder_gender" : "object",
                          "policy_holder_age" : "float64",
                          "VK_CL_PARTY_TYPE" : "object",
                          "responsible_user_gender" : "object",
                          "responsible_user_age" : "float64",
                          "VEHICLE_MAKE" : "object",
                          "VEHICLE_MODEL" : "object",
                          "vehicle_age" : "float64",
                          "CL_VEHICLE_CATEGORY" : "object",
                          "VEHICLE_ENGINE_POWER" : "float64" ,
                          "VEHICLE_REG_MASS" : "float64",
                          "VEHICLE_SEAT_COUNT" : "object",
                          "VEHICLE_FRAME_TYPE" : "object",
                          "color" : "object",
```

```
37                               "CL_VEHICLE_FUEL" : "object",
38                               "counts" : "float64",
39                               "exposure" : "float64"})
40 
41 data = pd.concat(data_chunks)
42 
43 end_time = time.time()
44 print("%s seconds" % (end_time - start_time))
45 
46 
47 #Converting the datatypes to categorical
48 
49 start_time = time.time()
50 
51 data.cov_start_month = data.cov_start_month.astype("category")
52 data.OM_CL_PARTY_TYPE = data.OM_CL_PARTY_TYPE.astype("category")
53 data.owner_gender = data.owner_gender.astype("category")
54 data.KV_CL_PARTY_TYPE = data.KV_CL_PARTY_TYPE.astype("category")
55 data.policy_holder_gender = data.policy_holder_gender.astype("category")
56 data.VK_CL_PARTY_TYPE = data.VK_CL_PARTY_TYPE.astype("category")
57 data.responsible_user_gender = data.responsible_user_gender.astype("
       category")
58 data.VEHICLE_MAKE = data.VEHICLE_MAKE.astype("category")
59 data.VEHICLE_MODEL = data.VEHICLE_MODEL.astype("category")
60 data.CL_VEHICLE_CATEGORY = data.CL_VEHICLE_CATEGORY.astype("category")
61 data.VEHICLE_SEAT_COUNT = data.VEHICLE_SEAT_COUNT.astype("category")
62 data.VEHICLE_FRAME_TYPE = data.VEHICLE_FRAME_TYPE.astype("category")
63 data.color = data.color.astype("category")
64 data.CL_VEHICLE_FUEL = data.CL_VEHICLE_FUEL.astype("category")
65 
66 data.exposure = data.exposure/365.0
67 
68 end_time = time.time()
69 print("%s seconds" % (end_time - start_time))
70 
71 
72 #Creating a dummy-variables, using one-hot-encoding
```

```
start_time = time.time()

data_dummy = pd.get_dummies(data, dummy_na = True)

end_time = time.time()
print("%s seconds" % (end_time - start_time))


data_dummy.shape


del data


#Creating a training, validation and test set

start_time = time.time()

X_train, X_test, y_train, y_test = train_test_split(data_dummy.drop(
    columns= ["counts"]), data_dummy.counts, test_size=0.2, random_state
    =1)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
    test_size=0.25, random_state=1)

end_time = time.time()
print("%s seconds" % (end_time - start_time))

del data_dummy


#XGBoost works faster, when the data has given a DMatrix structure.
#base_margin gives the opportunity to set an offset

start_time = time.time()

train_dmatrix = xgb.DMatrix(data=X_train, label=y_train, base_margin=np.
    log(X_train.exposure))
```

```
106 val_dmatrix = xgb.DMatrix(data=X_val,label=y_val, base_margin=np.log(
        X_val.exposure))
107 test_dmatrix = xgb.DMatrix(data=X_test,label=y_test, base_margin=np.log(
        X_test.exposure))
108
109 end_time = time.time()
110 print("%s seconds" % (end_time - start_time))
111
112
113 #Finding the best parameters
114
115
116 #Parameters to check
117 etas = [0.005, 0.01, 0.05, 0.1, 0.3, 0.5]
118 #etas = [0.26, 0.28, 0.3, 0.32, 0.34]
119 max_depths= [3, 5, 6, 8]
120 #max_depths= [1, 2, 8, 9]
121 lambdas = [0, 1, 3, 5]
122 alphas = [0, 1, 3, 5]
123 numrounds = [50, 100]
124 deviances = []
125 #How many trials:
126 print(len(etas) * len(max_depths) * len(lambdas) * len(alphas) * len(
        numrounds))
127
128
129 trial_count=1
130 best_validation_deviance = 100
131 best_validation_trial = 0
132
133 for eta_i in etas:
134     for max_depth_i in max_depths:
135         for lambda_i in lambdas:
136             for alpha_i in alphas:
137                 for numround_i in numrounds:
138                     #Printing information about the trial
139                     print("Trial number: ", trial_count)
```

```
140                     start_time = time.time()
141                     print("Started at: ", datetime.fromtimestamp(
    start_time))
142
143                     param = {'eta': eta_i, 'max_depth': max_depth_i, '
    lambda': lambda_i, 'alpha': alpha_i,
144                              'objective': 'count:poisson', 'tree_method'
    : 'approx'}
145                     print(param)
146
147                     #Training XGBoost
148                     bst = xgb.train(param, train_dmatrix,
    num_boost_round=numround_i,
149
    early_stopping_rounds = 10, evals=[(train_dmatrix, "Train"),(
    val_dmatrix, "Val")])
150                     #Calculating the Poisson deviance on the validation
    set
151                     preds_val = bst.predict(val_dmatrix)
152                     poisson_deviance_val = sklearn.metrics.
    mean_poisson_deviance(y_val,preds_val)
153                     print("Poisson_deviance val: %f" % (
    poisson_deviance_val))
154                     deviances.append(poisson_deviance_val)
155
156                     #Saving the best deviance and trial number
157                     if(poisson_deviance_val < best_validation_deviance):
158                         best_validation_deviance = poisson_deviance_val
159                         best_validation_trial = trial_count
160
161                     trial_count +=1
162                     end_time = time.time()
163                     print("%s seconds" % round((end_time - start_time)
    ,2))
164                     print()
165                     print()
166
```

```
#Creating the best model
start_time = time.time()

#Printing the time the program started
print("Started at: ", datetime.fromtimestamp(start_time))

#Specifying parameters
param = {'eta': 0.34, 'max_depth': 2, 'lambda': 1, 'alpha': 3,
         'objective': 'count:poisson', 'tree_method': 'approx'}

#Train xgboost
bst = xgb.train(param, train_dmatrix, num_boost_round = 1500,
    early_stopping_rounds = 10,
                evals=[(train_dmatrix, "Train"),(val_dmatrix, "Val")])

preds_val = bst.predict(val_dmatrix)
poisson_deviance_val = sklearn.metrics.mean_poisson_deviance(y_val,
    preds_val)
print("Poisson_deviance val: %f" % (poisson_deviance_val))

end_time = time.time()
print("%s seconds" % round((end_time - start_time),2))

#Predicting on the test-set
preds_test = ennustus = bst.predict(test_dmatrix)
poisson_deviance_test = sklearn.metrics.mean_poisson_deviance(y_test,
    preds_test)
print("Poisson_deviance test: %f" % (poisson_deviance_test))

#Saving the model
#bst.save_model("best_model")


#Plotting
from xgboost import plot_importance
from matplotlib import pyplot
```

```
pyplot.rcParams["figure.figsize"]=30,30

#Plotting the variable importances
plot_importance(bst)

#Plotting the first tree
xgb.to_graphviz(bst, num_trees=0)
```

Appendix 10. The code for creating a GLM and GAM model

```
#########
## GLM ##
#########

#Setting the seed for reproduction of analysis.
set.seed(1)

#Reading in the dataset
library(readr)
X_train_GLM <- read_delim("X_train_GLM_final.csv",
                          ";", escape_double = FALSE, col_types = cols(
    VEHICLE_SEAT_COUNT = col_character(),

    cov_start_month = col_character()),
                          trim_ws = TRUE)


y_train_GLM <- read_csv("y_train_GLM_final.csv",
                        col_names = FALSE)


#Changing the type of categorical variables
X_train_GLM$cov_start_month=as.factor(X_train_GLM$cov_start_month)
X_train_GLM$OM_CL_PARTY_TYPE=as.factor(X_train_GLM$OM_CL_PARTY_TYPE)
X_train_GLM$KV_CL_PARTY_TYPE=as.factor(X_train_GLM$KV_CL_PARTY_TYPE)
X_train_GLM$VK_CL_PARTY_TYPE=as.factor(X_train_GLM$VK_CL_PARTY_TYPE)
X_train_GLM$VEHICLE_MODEL=as.factor(X_train_GLM$VEHICLE_MODEL)
X_train_GLM$VEHICLE_MAKE=as.factor(X_train_GLM$VEHICLE_MAKE)
X_train_GLM$owner_gender=as.factor(X_train_GLM$owner_gender)
X_train_GLM$policy_holder_gender=as.factor(X_train_GLM$policy_holder_
    gender)
X_train_GLM$responsible_user_gender=as.factor(X_train_GLM$responsible_
    user_gender)
X_train_GLM$CL_VEHICLE_CATEGORY=as.factor(X_train_GLM$CL_VEHICLE_
    CATEGORY)
X_train_GLM$VEHICLE_SEAT_COUNT=as.factor(X_train_GLM$VEHICLE_SEAT_COUNT)
```

```
32 X_train_GLM$VEHICLE_FRAME_TYPE=as.factor(X_train_GLM$VEHICLE_FRAME_TYPE)
33 X_train_GLM$color=as.factor(X_train_GLM$color)
34 X_train_GLM$CL_VEHICLE_FUEL=as.factor(X_train_GLM$CL_VEHICLE_FUEL)
35 
36 #Changing the exposure in days to exposure in years
37 X_train_GLM$exposure=X_train_GLM$exposure/365.0
38 
39 #Adding the counts to the dataset
40 X_train_GLM$count = as.numeric(unlist(y_train_GLM))
41 
42 #Removing the variable y_train_GLM to save memory
43 rm(y_train_GLM)
44 
45 #Deleting the party type variables, since they are described by the
     gender variables now
46 X_train_GLM$OM_CL_PARTY_TYPE<-NULL
47 X_train_GLM$KV_CL_PARTY_TYPE<-NULL
48 X_train_GLM$VK_CL_PARTY_TYPE<-NULL
49 
50 #Looking at the correlations between numeric variables.
51 cor(X_train_GLM[,c(3,5,7,10,12,13,18)])
52 
53 
54 #Running the first model
55 library(biglm)
56 #Not using the exponential notation for presenting the results
57 options(scipen=999)
58 
59 m1 = bigglm(terms(count ~ cov_start_month  + owner_gender + policy_
     holder_gender + policy_holder_age + responsible_user_gender +
60                      responsible_user_age + VEHICLE_MAKE + VEHICLE_MODEL
     + vehicle_age +  CL_VEHICLE_CATEGORY + VEHICLE_ENGINE_POWER +
61                      VEHICLE_REG_MASS + VEHICLE_SEAT_COUNT + VEHICLE_
     FRAME_TYPE +  color + CL_VEHICLE_FUEL +
62                      exposure +  offset(log(exposure)),data=X_train_GLM),
63            data=X_train_GLM, family=poisson(), maxit=30)
64 
```

```
65 #Checking, if the model converged
66 m1$converged
67 #Looking at the summary of the model
68 summary(m1)
69
70
71 #Regrouping the variables
72
73 library(car)
74
75 X_train_GLM$VEHICLE_MAKE <- recode(X_train_GLM$VEHICLE_MAKE,
76                                     "c('BMW', 'MERCEDES-BENZ','MAZDA',
       'RENAULT')='Group_1';
77                                     c('CHRYSLER', 'CITROEN', 'HONDA',
      'OPEL', 'PEUGEOT', 'SUBARU', 'TOYOTA', 'AUDI')='Group_2';
78                                     c('FORD', 'KIA', 'NISSAN', 'other
      ', 'SKODA', 'VOLKSWAGEN', 'VOLVO')='Group_3';
79                                     c('MITSUBISHI', 'HYUNDAI')='Group_
      4' ")
80 #Checking the if the grouping was correxct
81 levels(X_train_GLM$VEHICLE_MAKE)
82
83
84 X_train_GLM$color <- recode(X_train_GLM$color,
85                              "c('blue', 'black', 'grey', 'silver')='
      blue_black_grey_silver';
86                              c('red', 'brown', 'green')='red_brown_
      green';
87                              c('white', 'yellow')='white_yellow' ")
88 levels(X_train_GLM$color)
89
90
91 X_train_GLM$cov_start_month <- recode(X_train_GLM$cov_start_month,
92                            "c('1', '2', '8', '9','10','12')='1_2_8_9_10
      _12';
93                              c('11')='11';
94                              c('3', '4', '5', '6', '7')='3_4_5_6_7' ")
```

```r
levels(X_train_GLM$cov_start_month)


X_train_GLM$VEHICLE_SEAT_COUNT <- recode(X_train_GLM$VEHICLE_SEAT_COUNT,
                                          "c('2.0', '3.0')='2_3';
                                           c('4.0', '5.0', 'other')='4_5_other';
                                           c('6.0', '7.0')='6_7';
                                           c('8.0', '9.0')='8_9' ")
levels(X_train_GLM$VEHICLE_SEAT_COUNT)


X_train_GLM$VEHICLE_MODEL <- recode(X_train_GLM$VEHICLE_MODEL,
                                     "c('AVENSIS', 'GOLF', 'OCTAVIA', 'other')='GROUP_1';
                                      c('PASSAT', 'PASSAT VARIANT')='Group_2' ")
levels(X_train_GLM$VEHICLE_MODEL)


X_train_GLM$VEHICLE_FRAME_TYPE <- recode(X_train_GLM$VEHICLE_FRAME_TYPE,
                                          "c('SEDAAN', 'LUUKP RA')='SEDAAN_LUUKP';
                                          c('KUPEE', 'MAHTUNIVERSAAL', 'UNIVERSAAL')='KUPEE_M_UNIVERSAAL';
                                          c('PIKAP', 'other', 'KAUBIK')='PIKAP_KAUBIK_other' ")
levels(X_train_GLM$VEHICLE_FRAME_TYPE)


X_train_GLM$CL_VEHICLE_FUEL <- recode(X_train_GLM$CL_VEHICLE_FUEL,
                                       "c('Bensiin-H briid', 'Bensiin-Kat', 'Diisel')='bens_hyb_kat__diisel';
                                          c('Elekter', 'other')='elekter_other';
                                          c('Bensiin')='bensiin' ")

```

```
124 levels(X_train_GLM$CL_VEHICLE_FUEL)
125
126
127 X_train_GLM$CL_VEHICLE_CATEGORY <- recode(X_train_GLM$CL_VEHICLE_
    CATEGORY,
128                                           "c('M1', 'M1G')='M1_M1G';
129                                       c('N1', 'N1G')='N1_N1G'")
130 levels(X_train_GLM$CL_VEHICLE_CATEGORY)
131
132
133 X_train_GLM$responsible_user_gender <- recode(X_train_GLM$responsible_
    user_gender,
134                                          "c('F', 'M', 'Legal')='Yes_r_
    user';
135                                       c('No_r_user')='No_r_user'")
136 levels(X_train_GLM$responsible_user_gender)
137
138
139 ###############
140 ## Modelling ##
141 ###############
142
143 #Running a new model after leaving out the engine power
144 m2 = bigglm(terms(count ~ cov_start_month  + owner_gender + policy_
    holder_gender + policy_holder_age + responsible_user_gender +
145                   responsible_user_age + VEHICLE_MAKE + VEHICLE_MODEL
    + vehicle_age +  CL_VEHICLE_CATEGORY + VEHICLE_ENGINE_POWER +
146                   VEHICLE_REG_MASS + VEHICLE_SEAT_COUNT + VEHICLE_
    FRAME_TYPE +  color + CL_VEHICLE_FUEL + exposure +  offset(log(
    exposure)),
147                 data=X_train_GLM),data=X_train_GLM,family=poisson(),
    maxit=30)
148
149 m2$converged
150 summary(m2)
151 deviance(m2)
152
```

```
153
154 #Since bigglm does not support lrtest function from package lmtest, then
        a new likelihood ratio test function
155 #was written by the author.
156 library(broom)
157 l_r_test <- function(model1, model2){
158   dev1=glance(model1)$deviance
159   dev2=glance(model2)$deviance
160
161   teststat<- -2*(dev1-dev2)
162   print(c("test-statistic: ", teststat))
163
164   df <- length(coef(model1)) - length(coef(model2))
165   print(c("df: ", df))
166
167   p_value= pchisq(teststat,df=df,lower.tail=FALSE)
168   print(c("p-value: ", p_value))
169 }
170
171
172 #Taking out the variables one by one and using the l_r_test function to
      compare them
173 m3 = bigglm(terms(count ~ owner_gender + policy_holder_gender + policy_
      holder_age + responsible_user_gender +
174                       responsible_user_age + VEHICLE_MAKE + VEHICLE_MODEL
      + vehicle_age +  CL_VEHICLE_CATEGORY + VEHICLE_ENGINE_POWER +
175                       VEHICLE_REG_MASS + VEHICLE_SEAT_COUNT + VEHICLE_
      FRAME_TYPE +  color + CL_VEHICLE_FUEL + exposure +  offset(log(
      exposure)),
176                    data=X_train_GLM),data=X_train_GLM,family=poisson(),
      maxit=30)
177 m3$converged
178 summary(m3)
179 deviance(m3)
180
181 l_r_test(m2,m3)
182
```

```
183 #Looking at the AIC values for both models
184 AIC(m2)
185 AIC(m3)
186
187
188
189
190
191
192
193 #######################
194 ## Modelling with GAM ##
195 #######################
196
197 set.seed(1)
198
199 library(mgcv)
200
201 #Measuring the time it takes to build the model
202 beginning_time <-Sys.time()
203
204 #Using the function bam, which runs a gam model with big data
205
206 b1 <- bam(count ~ s(policy_holder_age, k=10, bs="cs") + s(responsible_
       user_age, k=5, bs="cs") +
207             s(vehicle_age, k=5, bs="cs") +
208             s(VEHICLE_ENGINE_POWER, k=5, bs="cs") +
209             s(VEHICLE_REG_MASS, k=5, bs="cs") + s(exposure, k=10, bs="cs
       ") +
210             cov_start_month + owner_gender + policy_holder_gender +
       responsible_user_gender + VEHICLE_MAKE +
211             VEHICLE_MODEL + CL_VEHICLE_CATEGORY + VEHICLE_SEAT_COUNT +
       VEHICLE_FRAME_TYPE +
212             color + CL_VEHICLE_FUEL + offset(log(exposure)),
213           data=X_train_GLM, family=poisson())
214
215 #Checking if the model converged
```

```
216 b1$converged
217 #Looking at the model summary
218 summary.gam(b1)
219 #Checking if the k-values are suitable
220 k.check(b1)
221 gam.check(b1)
222 
223 end_time <-Sys.time()
224 end_time-beginning_time
225 
226 
227 #Increasing the values for k and building a new model:
228 beginning_time <-Sys.time()
229 
230 b2 <- bam(count ~ s(policy_holder_age, k=15, bs="cs") + s(responsible_
    user_age, k=8, bs="cs") +
231             s(vehicle_age, k=8, bs="cs") +
232             s(VEHICLE_ENGINE_POWER, k=8, bs="cs") +
233             s(VEHICLE_REG_MASS, k=8, bs="cs") + s(exposure, k=20, bs="cs
    ") +
234             cov_start_month + owner_gender + policy_holder_gender +
    responsible_user_gender + VEHICLE_MAKE +
235             VEHICLE_MODEL + CL_VEHICLE_CATEGORY + VEHICLE_SEAT_COUNT +
    VEHICLE_FRAME_TYPE +
236             color + CL_VEHICLE_FUEL + offset(log(exposure)),
237           data=X_train_GLM, family=poisson())
238 
239 #Checking, if the model converged
240 b2$converged
241 #Looking at the summary of the model
242 summary.gam(b2)
243 #Checking if the k-values are suitable
244 gam.check(b2)
245 k.check(b2)
246 
247 end_time <-Sys.time()
248 end_time-beginning_time
```

Appendix 11. The code for calculating the deviances for GLM and GAM

```
#Import packages
import pandas as pd
import numpy as np
import sklearn

#Import y_test
data_chunks = pd.read_csv("y_test_glm_final.csv",
                          chunksize=100000, header=None)
y_test= pd.concat(data_chunks)


#Import GLM predictions
data_chunks = pd.read_csv("y_predictions_glm.csv",
                          chunksize=100000)
y_pred_GLM = pd.concat(data_chunks)


#Calculating the deviance for GLM model
poisson_deviance_test = sklearn.metrics.mean_poisson_deviance(y_test,
    y_pred_GLM)
print("Poisson_deviance: %f" % (poisson_deviance_test))


#Import GAM predictions
data_chunks = pd.read_csv("y_predictions_gam.csv",
                          chunksize=100000)
y_pred_GAM = pd.concat(data_chunks)


#Calculating the deviance for GAM model
poisson_deviance_test_gam = sklearn.metrics.mean_poisson_deviance(y_test
    ,y_pred_GAM)
print("Poisson_deviance: %f" % (poisson_deviance_test_gam))

```

```
33
34 #Import GLM predictions, only numeric variables
35 data_chunks = pd.read_csv("y_predictions_glm_numeric.csv",
36                           chunksize=100000)
37
38 y_pred_GLM_numeric = pd.concat(data_chunks)
39
40
41 #Calculating the deviance for GLM numeric model
42 poisson_deviance_test = sklearn.metrics.mean_poisson_deviance(y_test,
       y_pred_GLM_numeric)
43 print("Poisson_deviance: %f" % (poisson_deviance_test))
44
45
46 #Import GAM predictions, only numeric variables
47 data_chunks = pd.read_csv("y_predictions_gam_numeric.csv",
48                           chunksize=100000)
49 y_pred_GAM_numeric = pd.concat(data_chunks)
50
51
52 #Calculating the deviance for GAM numericmodel
53 poisson_deviance_test = sklearn.metrics.mean_poisson_deviance(y_test,
       y_pred_GAM_numeric)
54 print("Poisson_deviance: %f" % (poisson_deviance_test))
```

Non-exclusive licence to reproduce thesis and make thesis public

I, Linnet Puskar,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,
   Estimation of MTPL claim frequency using GLM, GAM and XGBoost techniques,
   supervised by Meelis Käärik.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Linnet Puskar
26/05/2020