Boosting methods: Theory and application in R

Emanuel Sommer

2021-06-24

Contents

Chapter 1

Prerequisites

The reader should have some basic knowledge of the following topics:

- Statistical learning
  - Train/ test split
  - Cross-validation
  - Overfitting
  - Gradient descent
  - Bias variance trade-off
  - Hyperparameters
- Regression trees and random forest models
- Basic R knowledge (only for the applied part)

Chapter 2

Introduction

This bookdown project shows my work for the main master's seminar from my *M.Sc. Mathematics in Data Science* at the TUM. The topic of the seminar is *Statistical Methods and Models*. During the seminar I was supervised by Prof. Claudia Czado and Özge Sahin. My topic and thus covered in this project are boosting methods. Those are unarguably one of the hottest machine learning algorithms for tabular data to date.

> "Boosting is one of the most powerful learning ideas introduced in the last twenty years." (Hastie et al., 2009)

Hereby the focus will lie on the regression and not the more often discussed classification setting. The main idea of boosting is to sequentially build models from some class of base learners to finally combine them to a powerful ensemble model. As the base learners regression trees will be chosen in this project.

The next chapter will cover the theory behind boosting and especially tree-based gradient boosting. Besides a very prominent, efficient and successful implementation of tree-based gradient boosting namely XGBoost will be discussed in this chapter. It gained a lot of attention when it was integral to many winning submissions in machine learning competitions on the platform Kaggle and comes with some interesting tweaks to the general algorithm.

The application of the discussed boosting models to two real world data sets with the use of the programming language R are the content of the subsequent chapters. This will be divided into exploratory data analysis and the modeling itself. Notably the framework of `tidymodels` is used in the practical part.

Chapter 3

Theory

3.1 The powerful idea of gradient boosting

As roughly mentioned in the introduction section 2 the main idea of boosting is to sequentially build weak learners that form a powerful ensemble model. With weak learners models with high bias and low variance are meant that perform at least a little better than guessing. This already shows that the sequential approach of gradient boosting with weak learners stands in strong contrast to bagged ensembles like random forest. There many models with low bias and high variance are fitted in a parallel fashion and the variance is then reduced by averaging over the models.(Boehmke and Greenwell, 2019) It is not totally clear from which field boosting methods emerged but some claim that the work of Freund and Schapire with respect to PAC learning in the 1990s were instrumental for their growth.(Hastie et al., 2009) PAC learning can be considered one field within the broader field of learning theory that tries to find generalization bounds for algorithms that are probably approximately correct (PAC).(Wolf, 2020) This section will first cover the general setup of gradient boosting as the most prominent method to train forward stagewise additive models. Secondly tree-based gradient boosting and finally a very efficient and robust tree-based gradient boosting algorithm namely XGBoost will be discussed in detail.

3.1.1 Forward Stagewise Additive Modeling

In the setting of the data set $\mathcal{D} = \{(y_i, x_i) \mid i \in [N]\}$ with predictors $x_i \in \mathbb{R}^m$ and target $y_i \in \mathbb{R}$ boosting is fitting the following additive, still quite general, model.

$$\hat{y_i} = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F} \tag{3.1}$$

Where $\mathcal{F}$ is the space of learning algorithms that will be narrowed down later on. Additive expansions like this are at the core of many other powerful machine learning algorithms like Neural Networks or Wavelets.(Hastie et al., 2009)

The formulation (3.1) leads to so called forward stagewise additive modeling which basically means that one sequentially adds $f \in \mathcal{F}$ to the current model ϕ_k without changing anything about the previous models.(Hastie et al., 2009) The algorithm is shown below.

Algorithm 1: Forward Stagewise Additive Modeling (Hastie et al., 2009)

---

1. Initialize $\phi_0(x) = 0$
2. For $k = 1$ to K do:
   - $(\beta_k, \gamma_k) = argmin_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, \phi_{k-1}(x_i) + \beta f(x_i, \gamma))$
   - $\phi_k(x) = \phi_{k-1}(x) + \beta_k f(x, \gamma_k)$

Where γ parameterizes the learner $f \in \mathcal{F}$ and the β_k are the expansion coefficients. L should be a differentiable loss function.

For example for the basic L_2 loss the expression to be minimized simplifies to the following:

$$L_2(y_i, \phi_{k-1}(x_i) + \beta f(x_i, \gamma)) = (y_i - \phi_{k-1}(x_i) - \beta f(x_i, \gamma))^2$$

As $y_i - \phi_{k-1}(x_i)$ is just the residual of the previous model, the next model that is added corresponds to the model that best approximates the residuals of the current model. Although the L_2 loss has many very nice properties like the above, it lacks robustness against outliers. Therefore two alternative losses for boosting in the regression setting are worth considering.

3.1.2 Robust loss functions for regression

As the L_2 loss squares the residuals, observations with large absolute residuals are overly important in the minimization step. This effect can be reduced intuitively by just using the L_1 loss i.e. minimize over the sum over just the

absolute residuals. To do this is indeed a valid approach and can reduce the influence of outliers greatly and thus make the final model more robust. Another good choice could be the **Huber** loss which tries to get the best of L_1 and L_2 loss.(Hastie et al., 2009)

$$L_{Huber}(y, f(x)) = \begin{cases} L_2(y, f(x)) & |y - f(x)| \leq \delta \\ 2\delta|y - f(x)| - \delta^2 & otherwise. \end{cases} \tag{3.2}$$

In Figure 3.1 is a comparison of the three different losses discussed so far.(Hastie et al., 2009)

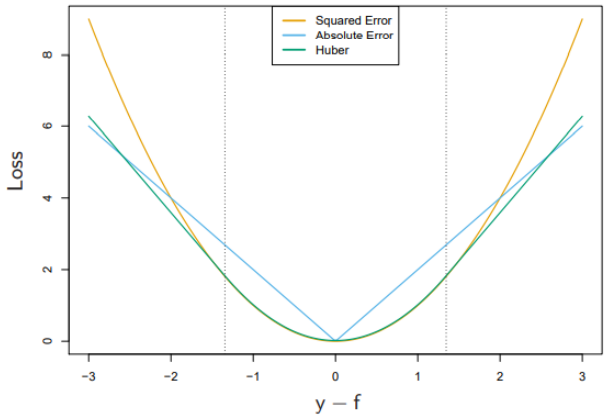


Figure 3.1: Comparison of different regression loss functions.

These alternative loss criteria are more robust but make the fitting i.e. the minimization much more complex as they do not yield such simplifications like the L_2 loss.(Hastie et al., 2009) The next step in the journey of exploring boosting is to narrow down the argument spaces of **Algorithm 1** (Forward Stagewise Additive Modeling) and to specify a subset of the general space of learning algorithms. This subset will be the space of Classification and Regression Tree (CART) models and in this case as the focus is on a regression task the space of regression trees. This choice is by no means arbitrary as in practice tree-based boosting algorithms have proven countless of times that they provide very robust and accurate models but still other learners might be chosen.(Hastie et al., 2009, Boehmke and Greenwell (2019)) The next subsection will explore how one can actually fit such a forward stagewise additive model when using regression trees as the learner class.

3.2 General gradient tree boosting

From now on there is the switch from the space of learning algorithms $\mathcal{F}$ to the space of regression trees $\mathcal{T}$. Such a regression tree can be formally expressed by:

$$t(x, \gamma, R) = \sum_{j=1}^{J} \gamma_j I(x \in R_j) \quad \text{for } t \in \mathcal{T} \tag{3.3}$$

With R_j being J distinct regions of the predictor space usually attained by recursive binary splitting. Moreover these regions correspond to the leafs of the tree and the number of leafs J or the depth of the trees are most often hyperparameters (not trained). The $\gamma_j \in \mathbb{R}$ are the predictions for a given x if x is contained in the region R_j. While it is quite easy to get the γ_j for the regions given, most often by computing $\gamma_j = \frac{1}{|\{x \in R_j\}|} \sum_{\{x \in R_j\}} x$, it is a much harder problem to get good distinct regions. The above mentioned recursive binary splitting is an approximation to do this and works in a top down greedy fashion.(Hastie et al., 2009) From now on we assume that we have an efficient way of fitting such trees to a metric outcome variable e.g. by recursive binary splitting.

A nice graphical example of an additive model based on trees is displayed in the figure 3.2 below.(Chen and Guestrin, 2016)

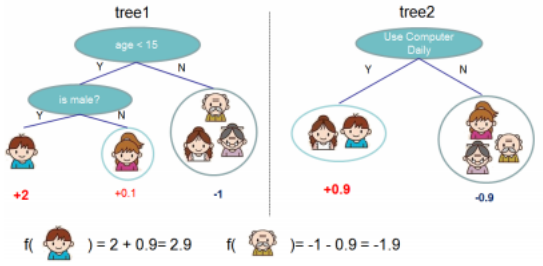


Figure 3.2: Example of an additive tree ensamble.

Having now the new space $\mathcal{T}$ for the general boosting model (3.1) one can write down the optimization problem that has to be solved in each step of the forward stagewise process of fitting the model.

$$(\gamma^{(k)}, R^{(k)}) = argmin_{\gamma, R} \sum_{i=1}^{N} L(y_i, \phi_{k-1}(x_i) + t(x_i, \gamma, R)) \tag{3.4}$$

This can be estimated fast and quite straight forward if there is a simplification like the one seen for the L_2 loss. But in the more general case of an arbitrary

differentiable convex loss function like the Huber loss techniques from numerical optimization are needed to derive fast algorithms.(Hastie et al., 2009)

3.2.1 Numerical optimization

According to **Algorithm 1** the goal in order to fit the boosting model is to minimize over the full loss of the training data $\mathcal{D}$ which is the sum over all observation losses.

$$L(\phi) = \sum_{i=1}^{N} L(y_i, \phi(x_i))$$

And thus the $\hat{\phi}$ additive boosting model we try to get is the following.

$$\hat{\phi} = argmin_{\phi} L(\phi)$$

Now we basically follow the spirit of the general gradient descent algorithm for differentiable functions. In the general case we minimize a function $f(x)$ by stepping iteratively along the direction of the steepest descent i.e. the negative gradient. The step length can then either be a constant small scalar or be determined by a line search.

In the setting of the additive boosting model ϕ can be viewed as a vector of dimension N that contains the prediction according to ϕ of the corresponding observation i.e. $\phi = (\phi(x_1), ..., \phi(x_N))$. So the loss function $L(\phi)$ corresponds to the $f(x)$ in the general gradient descent algorithm. Numerical optimization, here gradient descent, then solves for $\hat{\phi}$ by a sum of vectors of the same dimension as the ϕ.(Hastie et al., 2009) The result of the sum ϕ_K ((3.5)) can be viewed as the current proposal for the optimal $\hat{\phi}$ after K optimization steps and each $h^{(k)}$ is the proposed improvement step.

$$\phi_K = \sum_{k=0}^{K} h^{(k)} \quad h^{(k)} \in \mathbb{R}^N \tag{3.5}$$

While $h^{(0)}$ is just an initial guess the subsequent $h^{(k)}$ are again the prediction vectors of the corresponding model out of $\mathcal{T}$ i.e. $h_i^{(k)} = t_k(x_i)$ with $t \in \mathcal{T}$. This means that each $\phi_k = \phi_{k-1} + h^{(k)}$. The $h^{(k)}$ are calculated via the gradient which finally comes into play. As mentioned above one minimizes the loss the most by going towards the direction of the steepest descent. For (3.5) follows from its additive formulation and defining $g^{(k)}$ as the gradient of $L(\phi_k)$ evaluated for ϕ_{k-1} the update $h^{(k)} = -\lambda_k g^{(k)}$. As we assumed the loss to be differentiable we see that $g^{(k)}$ is well defined. Here λ_k is the usual step length for gradient descent methods. It is the solution of the line search $\lambda_k = argmin_{\lambda} L(\phi_{k-1} - \lambda g^{(k)})$.

This λ_k almost exactly corresponds to the β_k in **Algorithm 1** although here the optimization is performed for every region of the tree separately.(Hastie et al., 2009)

With these insights it is clear that the tree predictions correspond to the negative gradient $-g^{(k)}$. Of course the predictions are not independent as the prediction is constant for each leaf of the tree. So the new optimization proposed by numerical optimization via gradient boosting is given in (3.6) below.

$$(\tilde{\gamma}^{(k)}, \tilde{R}^{(k)}) = argmin_{\gamma,R} \sum_{i=1}^{N} [-g_i^{(k)} - t(x_i, \gamma, R)]^2 \tag{3.6}$$

In words this just means fitting a regression tree by least squares to the negative gradients that were evaluated with the current predictions. The solution regions will not exactly match the ones from (3.4) but should be very similar.(Hastie et al., 2009) After having estimated the regions one estimates the parameters γ by solving the line search (3.7).

$$\tilde{\gamma}_j^{(k)} = argmin_{\gamma_j^{(k)}} \sum_{x \in R_j^{(k)}} L(y_i, \phi_{k-1}(x_i) + \gamma_j^{(k)}) \tag{3.7}$$

Putting all of this back together with **Algorithm 1** results in **Algorithm 2** that covers a general algorithm for tree-based gradient boosting.

Algorithm 2: Tree-based gradient boosting (Hastie et al., 2009)

---

1. Initialize $\phi_0(x)$ as a singular node tree.
2. For $k = 1$ to K do:
   - For $i = 1$ to N compute:
     $g_i^{(k)} = \left[\frac{\partial L(y_i, \phi(x_i))}{\partial \phi(x_i)} \right]_{\phi = \phi_{k-1}}$
   - Fit a regression tree by least squares to the outcome vector $-g^{(k)}$ in order to get the $J^{(k)}$ distinct regions $\tilde{R}_j^{(k)}$.
   - For each of these $J^{(k)}$ regions perform a line search in order to compute the leaf predictions $\tilde{\gamma}_j^{(k)}$ exactly like in (3.7).
   - Set $\phi_k(x) = \phi_{k-1}(x) + t(x, \tilde{\gamma}_j^{(k)}, \tilde{R}_j^{(k)})$ with $t \in \mathcal{T}$

The only unknowns in this algorithm are now the differentiable loss function and the hyperparameters like the $J^{(k)}$ and the K. While choices for the hyperparameters are discussed further below, the following table 3.1 displays the gradients for the losses discussed so far.

Table 3.1: Negative gradients of the discussed losses (Hastie et al., 2009)

Loss	Negative gradient
L_2: $(y_i - \phi(x_i))^2$	$2(y_i - \phi(x_i))$
L_1: $\lvert y_i - \phi(x_i)\rvert$	$sign(y_i - \phi(x_i))$
Huber loss (3.2)	$y_i - \phi(x_i)$ for $\lvert y_i - \phi(x_i)\rvert \leq \delta$ $\delta sign(y_i - \phi(x_i))$ otherwise, with δ quantile of $\lvert y_i - \phi(x_i)\rvert$

3.2.2 Single tree depth

The question which $J^{(k)}$ should be used at each iteration is now shortly discussed. Basically using the tree depth ($log_2(J^{(k)}) = Depth^{(k)}$) of one means allowing only the main effects and no interactions. A value of 2 already allows all two way interaction effects and so on. As one stacks these models additive and does not restrict oneself to a single one, the building of a large tree and then pruning it back at each iteration as the regular CART algorithms do would be a computational overkill. Instead it has proven to be sufficient and efficient in practice to set the values $Depth^{(k)}$ to a constant $Depth \approx 6$.(Hastie et al., 2009) Also decision stumps ($Depth = 1$) could be used but may require a lot more iterations.

3.2.3 Combat overfitting

No learning algorithm could be fully covered without treating the good old problem of overfitting. In the setting of boosting the experienced eye could have spotted the problem of overfitting already in the definition of the additive model (3.1). There the number K of models that form the ensemble was introduced but was not discussed further till now. Of course one can arbitrarily fit or better say remember some given training data in order to minimize the loss with such an additive model by letting K be arbitrarily large. This comes from the fact that the loss reduces usually after each iteration over K.(Hastie et al., 2009) The easiest way to prevent overfitting is to have a validation set at hand which is disjoint from the training data. Having such a validation set at hand can be used to monitor the loss on the unseen data. In the case the loss would

rise again on the validation data one can consider to stop the algorithm and to use the current iteration number for the final parameter K. This approach is often called early stopping. Besides that there are methods that regularize the individual trees that are fitted in each iteration. Two of those will be discussed now.

3.2.3.1 Shrinkage

Shrinkage basically refers to just introducing a learning rate η. This learning rate η scales the contribution of the new model. In general the learning rate should be in $(0, 1]$ but in practice it has been shown that rather small values like $\eta < 0.1$ work very good.(Hastie et al., 2009) As almost everything in life this does not come for free. A smaller learning rate usually comes with a computational cost as with lower η a larger K is required. Those two hyperparameters represent a trade-off in a way. In practice it is advisable to use a small learning rate η and adjust the K accordingly which in this case would mean to make the K large enough until one reaches an early stopping point. Still it is good to keep in mind that a η that is too small can lead to an immense and unnecessary computational effort. All in all using shrinkage the update step in **Algorithm 2** changes to the following.

$$\phi_k(x) = \phi_{k-1}(x) + \eta * t(x, \tilde{\gamma}_j^{(k)}, \tilde{R}_j^{(k)}) \tag{3.8}$$

3.2.3.2 Subsampling

The other method to regularize the trees will be subsampling. There are two different kinds of subsampling. The first one is row-subsampling which basically means just using a random fraction of the training data in each iteration when fitting the new tree. Common values are $\frac{1}{2}$ but for a larger training set the value can be chosen to be smaller. This does not only help to prevent overfitting and thus achieving a better predictive performance but reduces also the computational effort in each iteration.(Hastie et al., 2009) The approach is very similar to the dropout technique in the setting of deep neural networks.

Moreover one can apply column-subsampling or feature-subsampling. Here only a fraction of the features or an explicit number of the features are used in each iteration. This is the exact same method and intention as the one applied in random forest models. Again the technique not only boosts performance on unseen data but also reduces the computational time.(Chen and Guestrin, 2016) Still it must be noted that by using one or both subsampling methods one introduces one or two additional hyperparameters to the model that have to be tuned.

Having the **Algorithm 2** for tree-based gradient boosting alongside some regularization techniques it is time to look at a very popular, efficient and powerful

open source implementation.

3.3 XGBoost a highly efficient implementation

The XGBoost algorithm is a highly scalable, efficient and successful implementation and optimization of **Algorithm 2** introduced by Chen and Guestrin in 2016.(Chen and Guestrin, 2016, Boehmke and Greenwell (2019)) It has gained a lot of spotlight when it was integral for many winning submissions at kaggle machine learning challenges. In the following the most important tweaks that are proposed to **Algorithm 2** are covered.

3.3.1 Regularized loss

Instead of just minimizing a convex and differentiable loss L XGBoost minimizes the following regularized loss.(Chen and Guestrin, 2016)

$$\begin{aligned} \mathcal{L}(\phi) &= L(\phi) + \sum_{k=1}^{K} \Omega(t_k) \\ \text{where } \Omega(t) &= \nu J + \frac{1}{2}\lambda ||\gamma||^2 \quad t \in \mathcal{T} \end{aligned} \tag{3.9}$$

Here J and γ again parameterize the regression trees $t \in \mathcal{T}$ as defined in (3.3). As evident from the formula both hyperparameters ν and λ favor when increased less complex models at each iteration. This is again a measure against overfitting.

This regularized objective leads to a slightly modified version of the minimization problem (3.4) that has to be solved in each iteration.

$$(\gamma^{(k)}, R^{(k)}) = argmin_{\gamma,R} \sum_{i=1}^{N} L(y_i, \phi_{k-1}(x_i) + t(x_i, \gamma, R)) + \Omega(t(x_i, \gamma, R)) \tag{3.10}$$

Instead of the first order approach, that has been introduced in 3.2.1, XGBoost uses a second order approximation to further simplify the objective. This means that besides the first order gradient $g^{(k)}$ of the loss function evaluated at ϕ_{k-1} also the second order gradient $z_i^{(k)} = \left[\frac{\partial^2 L(y_i, \phi(x_i))}{\partial^2 \phi(x_i)}\right]_{\phi=\phi_{k-1}}$ is used. By neglecting constants one reaches the approximate new objective below.(Chen and Guestrin, 2016)

$$\begin{aligned}\tilde{\mathcal{L}}^{(k)} &= \sum_{i=1}^{N}[g_i^{(k)}t^{(k)}(x_i) + \frac{1}{2}t^{(k)}(x_i)^2] + \Omega(t^{(k)}) \\ &= \sum_{j=1}^{J}[(\sum_{\{i|x_i \in R_j^{(k)}\}} g_i^{(k)})\gamma_j^{(k)} + \frac{1}{2}(\sum_{\{i|x_i \in R_j^{(k)}\}} z_i^{(k)} + \lambda)(\gamma_j^{(k)})^2] + \nu J\end{aligned} \tag{3.11}$$

This representation is used as it can be decomposed by the J leafs of the tree. One can then define a scoring function for split finding that only depends on $g^{(k)}$ and $z^{(k)}$. This approach speeds up the split finding and allows further parallel computations with respect to the leafs.

3.3.2 Shrinkage and subsampling

As discussed above in 3.2.3 shrinkage and subsampling are great ways to regularize the model and prevent overfitting. XGBoost implements both shrinkage and subsampling. It enables the user to use column as well as row subsampling. The authors claim that in practice column subsampling has prevented overfitting more than row subsampling.(Chen and Guestrin, 2016)

3.3.3 Even more tweaks

Besides the changes above the authors also implemented an effective approximate algorithm for split finding for the tree building step. A very nice attribute of this approximate procedure is that the algorithm is sparsity aware i.e. the processing of missing values or 0 values is very efficient. This is done via a learned default direction in each split. Moreover they used very efficient data structures to allow a lot of parallel processing.(Chen and Guestrin, 2016) It is written in C++ but has APIs in various languages like in R via the `xgboost` package.(Chen et al., 2021)

3.3.4 Hyperparameters overview

As the **tidymodels** framework will be used in the applied part, the according names from the parsnip interface for tree-based boosting i.e. `boost_tree` are shown below.(Chen and Guestrin, 2016, Chen et al. (2021), Kuhn and Wickham (2020))

Table 3.2: XGBoost hyperparameters overview.

Hyperparameter	Notation above	Meaning
`trees`	K	Number of trees in the ensemble.
`tree_depth`	$max(log_2 J^{(k)})$	Maximum depth of the trees.
`learn_rate`	η	Learning rate (shrinkage).
`min_n`		Minimum number of observations in terminal region $R_j^{(k)}$.
`lambda`	λ	L_2 regularization parameter on γ as in (3.11) (default = 0).
`alpha`		L_1 regularization parameter on γ (default = 0).
`loss_reduction`		Minimal loss reduction for terminal partitions. (default = 0)
`mtry`		Proportion or number of columns for column subsampling.
`sample_size`		Proportion of observations for row subsampling.
`stop_iter`		Allowed number of rounds without improvement before early stopping.

Note that the L_1 penalization term would be added to the Ω term in the loss $\mathcal{L}$ of XGBoost.

This closes the chapter on the theory of boosting methods with a focus on tree-based gradient boosting. The discussed tools will be put to the test in the subsequent sections where real world data will be analyzed with them.

Chapter 4

Explore the data

Before one starts with the actual modeling it is crucial to get to know the data and to bring it to the correct format. This process of getting familiar with the data is well known as Exploratory Data Analysis (EDA). To do this many packages are used.(Wickham et al., 2019; Kuhn and Wickham, 2020; Garnier, 2018a; Wilke, 2020; Garnier, 2018b, ; Pedersen, 2020; Tierney, 2017; Grolemund and Wickham, 2011; Meschiari, 2021; Schloerke et al., 2021; Sievert, 2020) The most important ones will be loaded below.

```
library(tidyverse) # general data handling tools
library(tidymodels) # data modeling and preprocessing
# color palettes
library(viridis)
library(viridisLite)
library(patchwork) # composing of ggplots
```

Before the fun can begin a quick outline of the steps performed on each data set.

- A general overview of the classes of the features and a visualization to detect any missing values.
- The distribution and the main effect on the outcome variable of each feature.
- The pairwise relationships of the predictors.
- Optionally some feature engineering.
- Using the gained knowledge to fix all pre-processing and feature engineering steps by using a `recipes::recipe`.

If one is mainly interested in the models themselves one can just have a look at the recipes and skip the rest of this section.

Some might ask why no interactions of predictors are covered in this EDA. If one would use a standard OLS, lasso or ridge regression it would be very important to have a look at them but as the focus here is on tree-based gradient boosting one already includes interactions if one is not restrictive to regression stumps.

It should also be mentioned that especially during the EDA most of the plotting code is not shown in order to support a better reading flow. If one wants to look up the code for any of these visualizations or also the full bookdown project one can have a deep dive at this Github reposatory. For example the respective code for the EDA can be found in the `03-eda.Rmd` file and the one for the modeling in the `04-modeling.Rmd` file. So having this said let's start with the first data set!

4.1 Burnout data

The data is from the machine learning challenge *HackerEarth Machine Learning Challenge: Are your employees burning out?*. And can be downloaded here: https://www.kaggle.com/blurredmachine/are-your-employees-burning-out?select=train.csv

```
# load the data
burnout_data <- read_csv("_data/burn_out_train.csv")
# convert colnames to snake_case
colnames(burnout_data) <- tolower(
  stringr::str_replace_all(
    colnames(burnout_data),
    " ",
    "_"
  ))
# omit missing values in the outcome variable
burnout_data <- burnout_data[!is.na(burnout_data$burn_rate),]
```

4.1.1 Train-test split

To not allow information leakage the train-test split is performed at the very start of the whole analysis.

```
set.seed(2)
burnout_split <- rsample::initial_split(burnout_data, prop = 0.80)
burnout_train <- rsample::training(burnout_split)
burnout_test  <- rsample::testing(burnout_split)
```

The training data set contains 17300 rows and 9 variables.

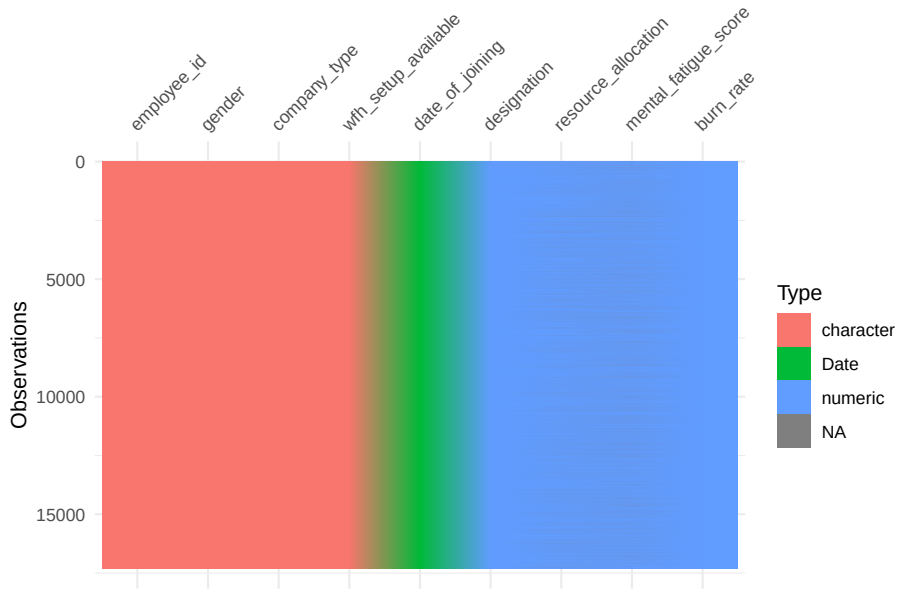
The test data set contains 4326 observations and naturally also 9 variables.

4.1.2 Quick general overview

First look at the classes of the variables.

column	class
employee_id	character
date_of_joining	Date
gender	character
company_type	character
wfh_setup_available	character
designation	numeric
resource_allocation	numeric
mental_fatigue_score	numeric
burn_rate	numeric

A general visualization of the whole data set to detect missing values below.




```
# percentage of missing values in the training data set
mean(rowSums(is.na(burnout_train)) > 0)
```

```
## [1] 0.1419075
```

As we know that XGBoost can handle missing values we do not have to be concerned. Although one could of course think about imputation or even removal.

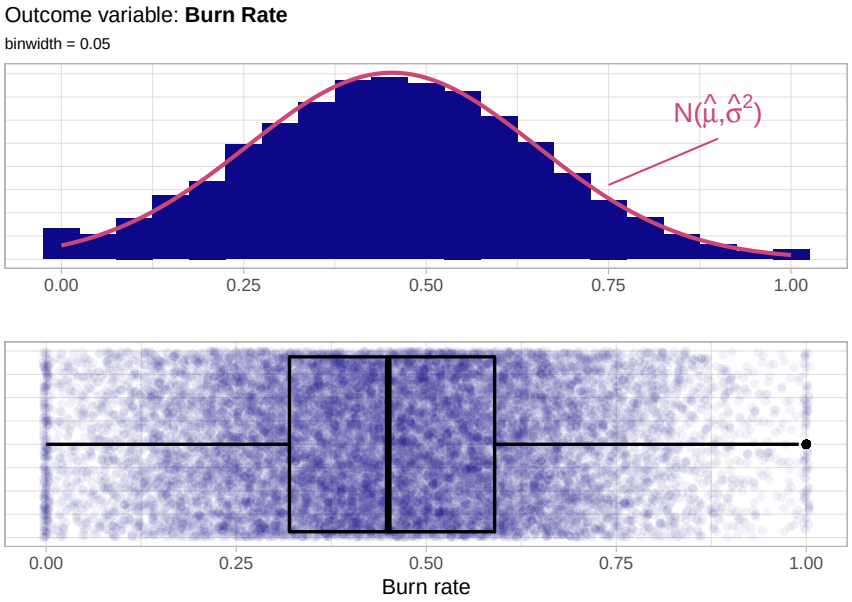
4.1.3 What about the outcome variable?

`burn_rate`: For each employee telling the rate of burnout should be in $[0, 1]$. The greater the score the worse the burnout (0 means no burnout at all). As the variable is continuous we have a regression task. Yet it has bounds which has to be treated with when predicting.

The five point summary below shows that the full range is covered and no invalid values are in the data.

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.3200  0.4500  0.4531  0.5900  1.0000
```

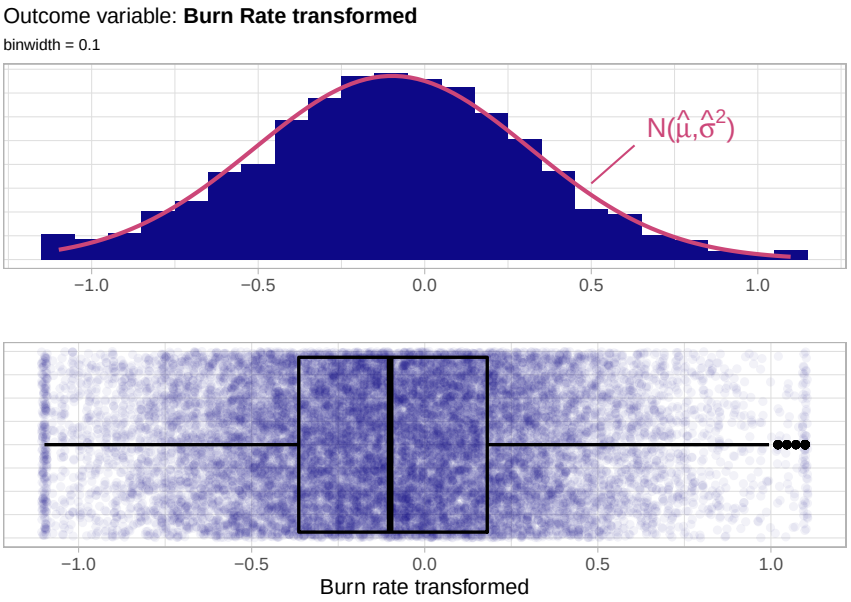
Now the distribution of the outcome.



The distribution of the outcome is very much symmetrical and bell shaped around 0.5 and the whole defined region $[0, 1]$ is covered quite well. Actually by overlaying a normal distribution with the sample mean $\hat{\mu}$ and the sample standard deviation $\hat{\sigma}^2$ as the parameters one can clearly see that the outcome almost perfectly follows a normal distribution. One could further fit a Q-Q-plot to visualize the normality. **BUT** of course here there is a bounded domain while the normal distribution has the whole $\mathbb{R}$ as domain. This bounded domain does not interfere with the boosted model as tree-based models do not superimpose a distribution assumptions upon the target variable. Nevertheless one can transform the outcome with the empirical logit $log(\frac{y_i+0.5}{1-y_i+0.5})$.By doing this one removes the bounds on the target. One can then re-transform the predictions

in the end by applying $\frac{2}{exp(-y)+1} - 0.5$. Here to see whether this transformation changes the behavior or improves the boosting model one will have a look not only at the untransformed target `burn_rate` but also at the transformed one `burn_rate_trans`. The focus will be on the untransformed modeling as the low pre-processing strength of such boosting models should be emphasized. Below is the distribution of the transformed one.

```
# Add transformed outcome
burnout_train$burn_rate_trans <- log((burnout_train$burn_rate + 0.5) /
                                       (1.5 - burnout_train$burn_rate))
burnout_test$burn_rate_trans <- log((burnout_test$burn_rate + 0.5) /
                                       (1.5 - burnout_test$burn_rate))
```



The transformed outcome basically resembles exactly the same properties as the untransformed one but the nice thing is that the bounds were removed. The further EDA will be based on the untransformed variable but the implications are the same due to the choice of the transformation via the empirical logit.

4.1.4 Distribution and main effects of the predictors

4.1.4.1 Employee ID

`employee_id` is just an ID variable and thus is not useful for any prediction model. But one has to check for duplicates.

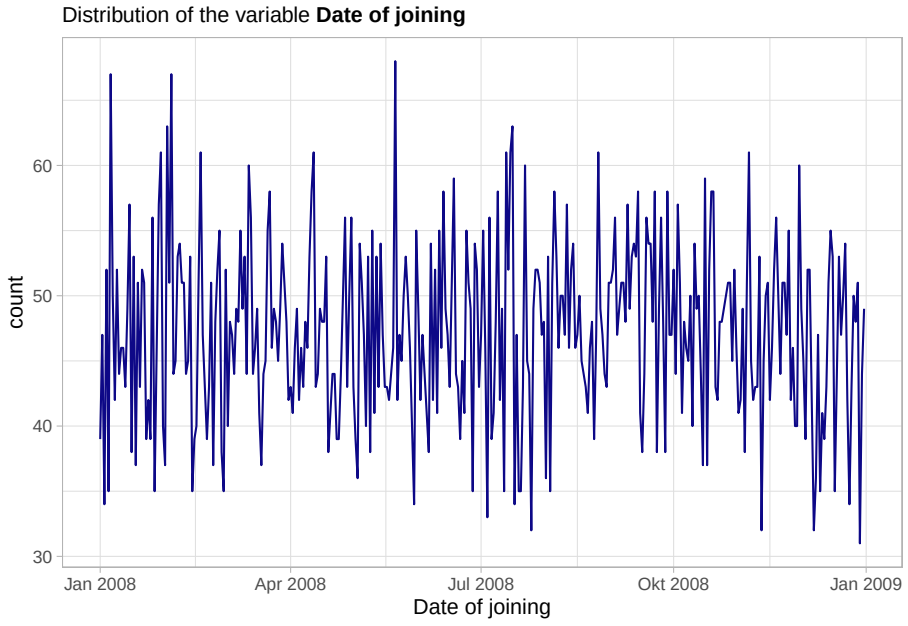
```
# TRUE if there are NO duplicates
burnout_train %>%
  group_by(employee_id) %>%
  summarise(n = n()) %>%
  nrow() == nrow(burnout_train)
```

```
## [1] TRUE
```

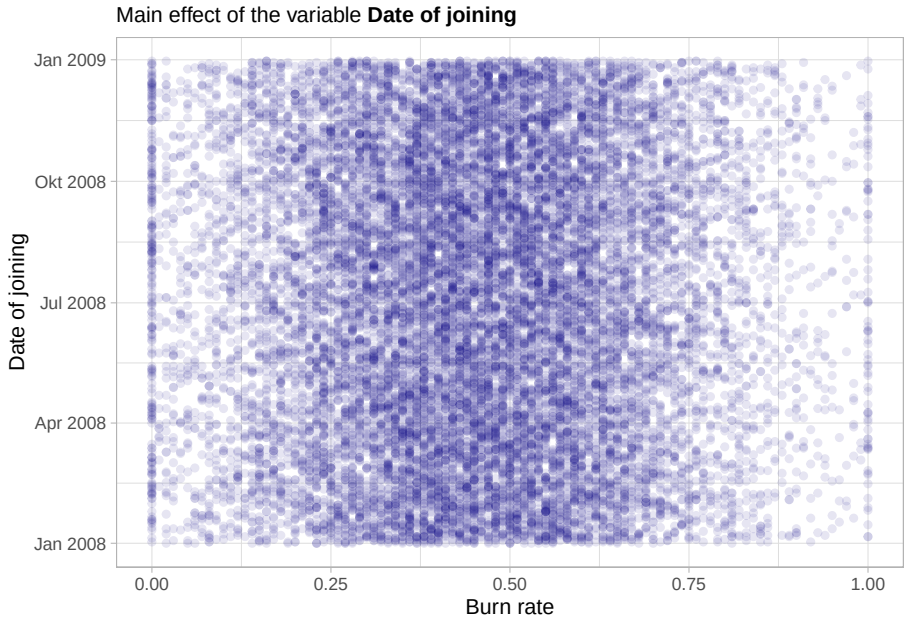
Thus there are no duplicates which is good.

4.1.4.2 Date of joining

`date_of_joining` is the date the employee has joined the company. Thus a continuous variable that most likely needs some kind of feature engineering.



Although there is a lot of variation no major trends in hirings are visible from this plot. Overall the variable seems to be quite equally distributed over the year 2008.



In its raw form the variable `date_of_joining` seems not to have a notable main effect on the outcome variable. Nevertheless the feature will be used in the model and as tree-based models have an in-built feature selection one can see after the fitting if the feature was helpful overall. The feature will not be included just as an integer (the default format how Dates are represented) but rather some more features like weekday or month will be extracted from the raw variable further down the road.

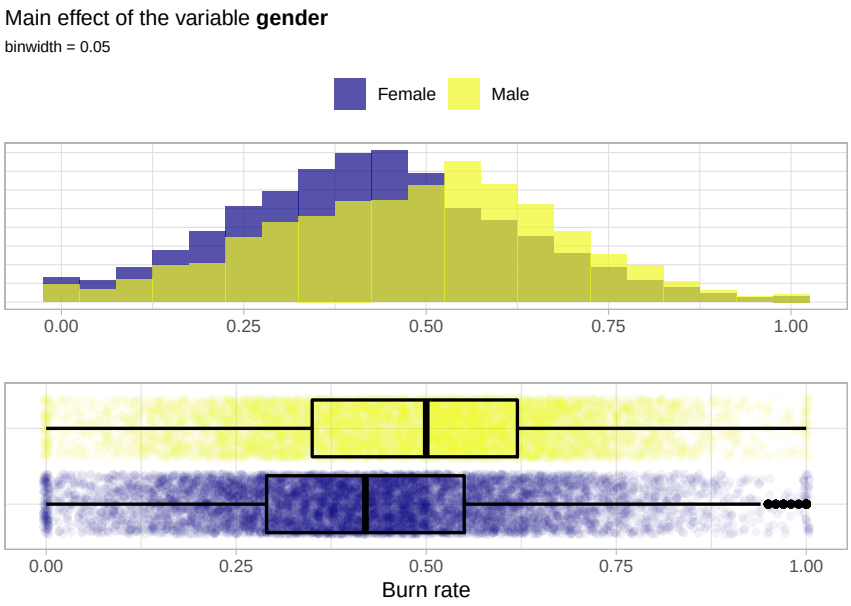
4.1.4.3 Gender

`gender` represents the gender of the employee. Definitely a categorical variable.

```
# have a look at the discrete distribution
summary(factor(burnout_train$gender))
```

```
## Female   Male
##   9039   8261
```

The two classes are well balanced. Now a look at the main effect of the feature.



For both classes the distributions are very similar and symmetrical. It seems like the male employees have overall a slightly higher risk of having a higher burn score i.e. a burnout.

4.1.4.4 Company type

`company_type` is a binary categorical variable that indicates whether the company is a service or product company.

```
# have a look at the discrete distribution
summary(factor(burnout_train$company_type))
```

```
## Product Service
##    6002   11298
```

In this case the classes are not fully balanced but each class is still well represented. Now a look at the main effect of the feature.

For both classes the distributions are almost identical and symmetrical. From an univariate point of view no notable main effect is visible from these visualizations.

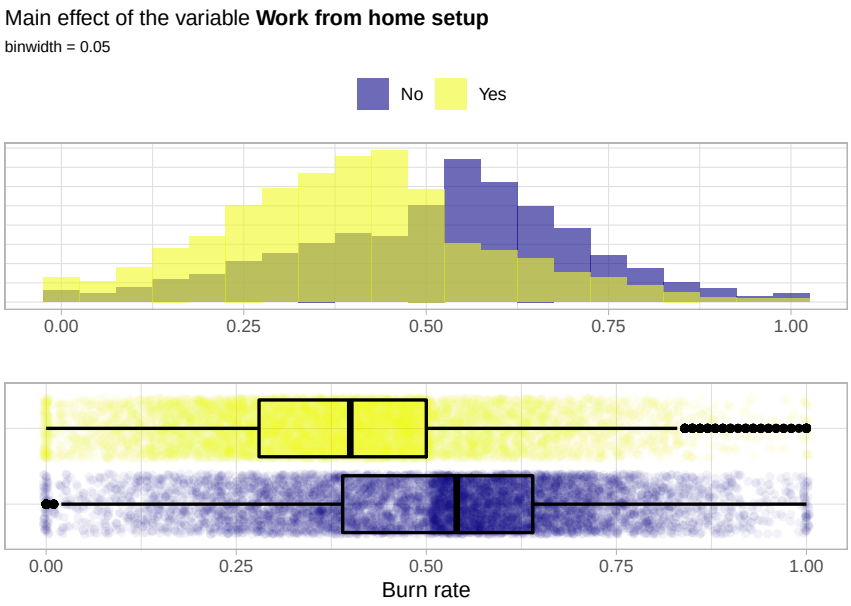
4.1.4.5 Work from home setup

`wfh_setup_available` indicates whether a working from home setup is available for the employee. So this is again a binary variable.

```
# have a look at the discrete distribution
summary(factor(burnout_train$wfh_setup_available))
```

```
##   No  Yes
## 7982 9318
```

The two classes are well balanced. Now a look at the main effect of the feature.



Again both distributions are quite similar i.e. bell shaped and symmetrical. Here quite a main effect is visible. A work from home setup most likely has a positive influence on the wellbeing and thus lowers the risk for a high burn rate.

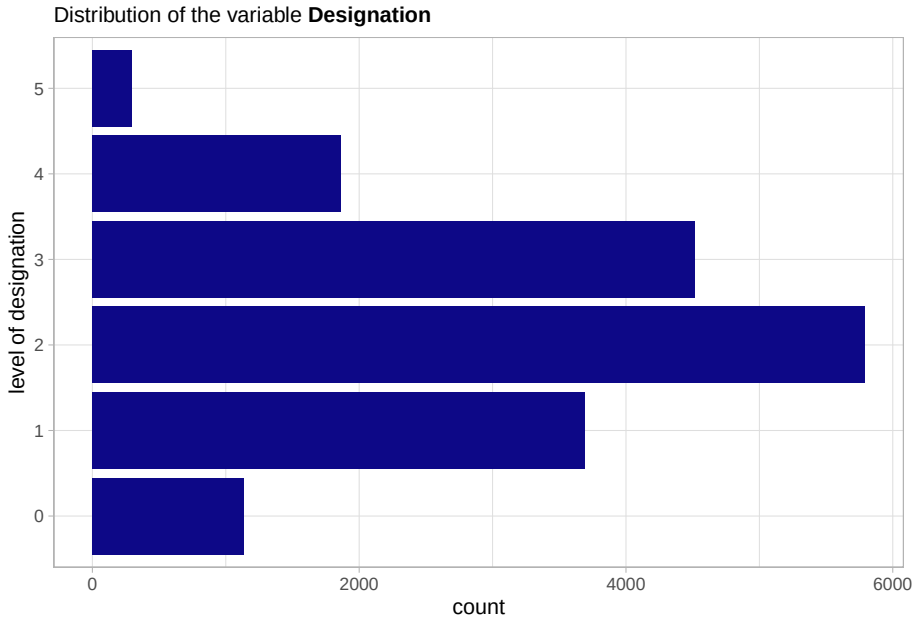
4.1.4.6 Designation

`designation` A rate within $[0, 5]$ that represents the designation in the company for the employee. High values indicate a greater amount of designation.

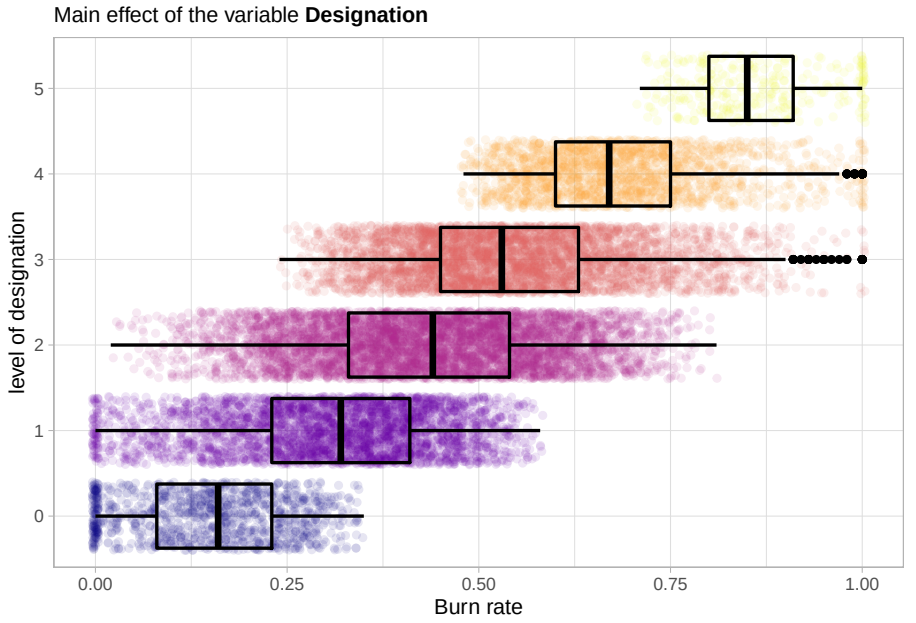
```
# unique values of the feature
unique(burnout_train$designation)
```

```
## [1] 0 2 3 1 4 5
```

As the feature has a natural ordering this variable will be treated as an ordinal one i.e. be encoded with the integers and not by one-hot-encoding.



Here clearly the more extreme levels of designation are less represented in the data. This makes total sense w.r.t. the meaning of the variable.



A strong main effect is visible in the plot. The plot also further strengthens the hypothesis that we should treat the feature as ordinal. A higher level of

designation seems to have an influence on the risk of having a burnout. For example employees from the training data set with a level of designation below 3 never even achieved a maximal burn score of one.

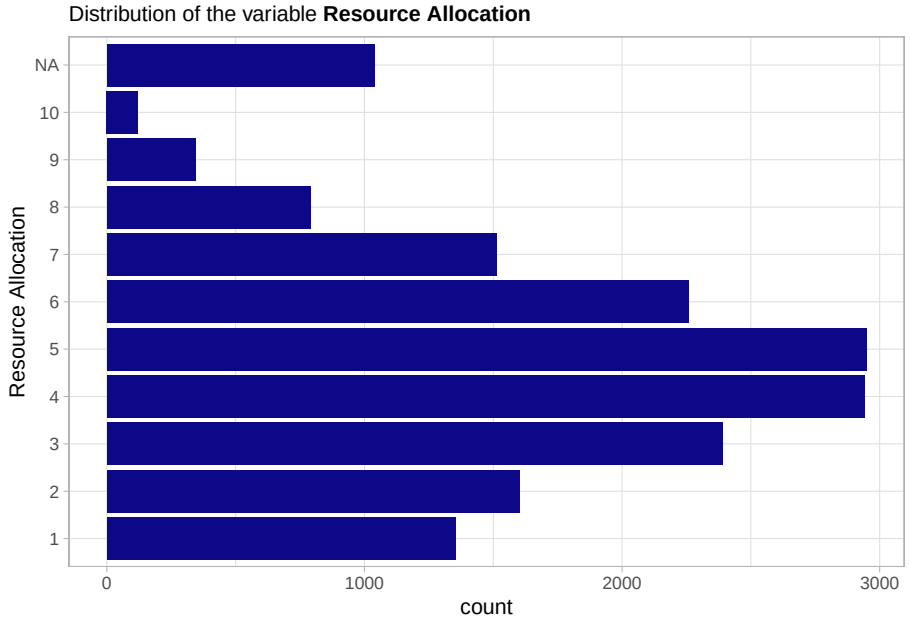
4.1.4.7 Resource allocation

`resource_allocation` A rate within $[1, 10]$ that represents the resource allocation to the employee. High values indicate more resources allocated to the employee.

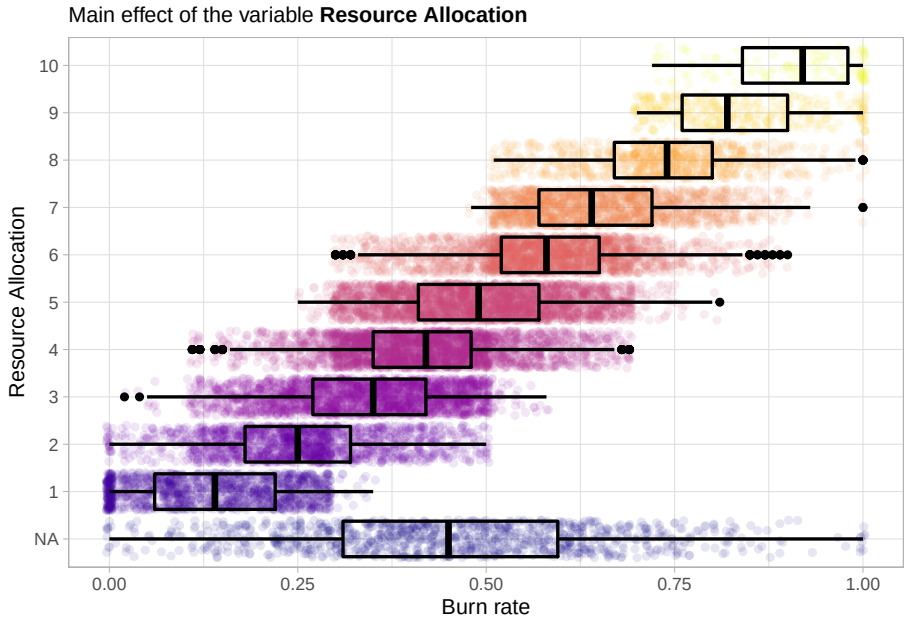
```
# unique values of the feature
unique(burnout_train$resource_allocation)
```

```
##  [1]  2  4  3  6  5  8  1  7 NA 10  9
```

Here again the question is whether one should encode this variable as a categorical or an ordinal categorical feature. In this case as there are quite some levels and again a natural ordering the variable will be encoded as a continuous integer score.



A similar behavior as the one of the previous variable is visible. But here there are some missing values (NA's).



A strong main effect is visible in the plot. The plot again further strengthens the hypothesis that we should treat this feature as ordinal. A higher amount of resources assigned to an employee seems to have a positive influence on the risk of having a burnout. The missing values do not seem to have some structure as they replicate the base distribution of the outcome variable.

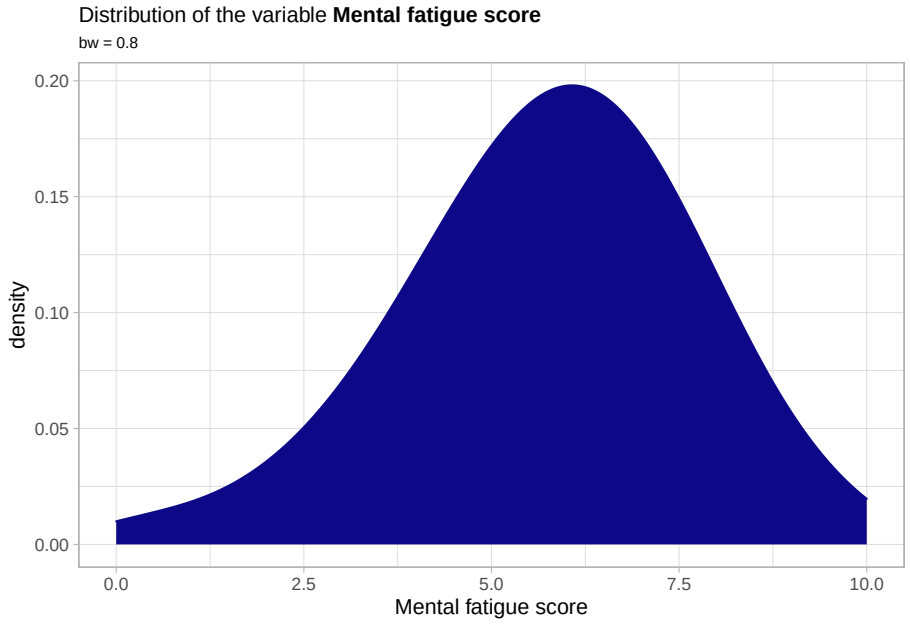
4.1.4.8 Mental fatigue score

`mental_fatigue_score` is the level of mental fatigue the employee is facing.

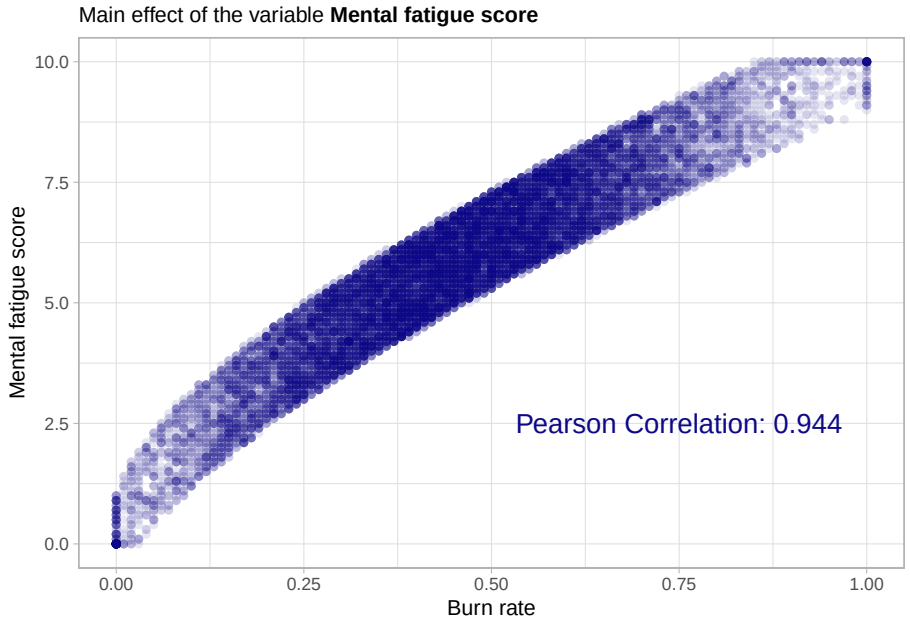
```
# number of unique values
length(unique(burnout_train$mental_fatigue_score))
```

```
## [1] 102
```

This variable will without a question be treated in a continuous way.



Although there is a very slight skew towards a higher mental fatigue score the overall distribution is still more or less bell shaped and quite symmetrical. Moreover the whole allowed range is covered and the bounds are not violated. Next the main effect of the variable.

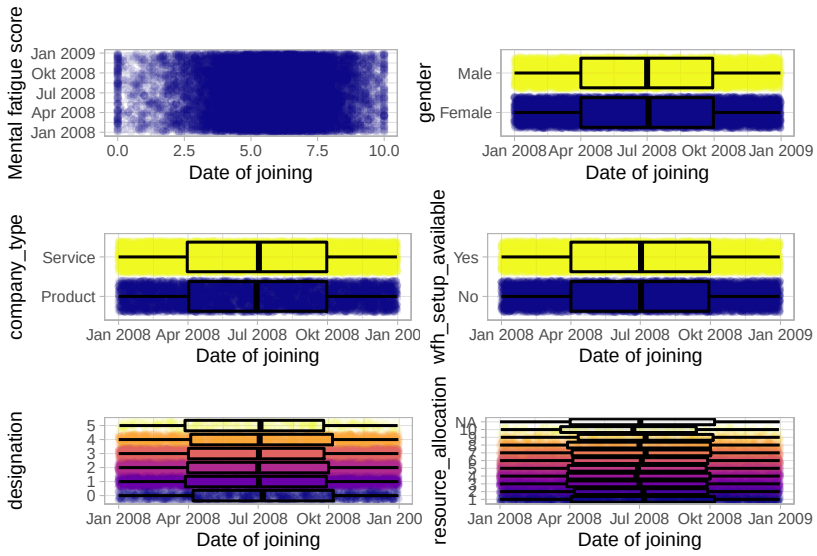


This scatterplot shows drastic results! The mental fatigue score has an almost perfect linear relationship with the outcome variable. This is also underlined by the very high pearson correlation. This indicates that mental fatigue score will be a most important predictor. If a communication with the data collector would be possible it would be important to check whether the two scores have common confounding variables as then one would have to question the practical usability of this predictor. This comes from the fact that no model would be needed if it was as hard to collect the data about the predictors as the outcome data. Moreover there are 1569 missing values in the feature so for those the model has to rely on the other maybe more weak predictors. It should be noted that when evaluating the final model one should consider to compare its performance to a trivial model (like a single intercept model). When constructing such a trivial model one could and maybe should also use this variable (when available) to get a trivial prediction by scaling the `mental_fatigue_score` feature by a simple scalar.

4.1.5 Relationships between the predictors

An exploration of the relationships between the predictors could also be done by having a look at a correlation and scatterplot matrix. This approach is much quicker than looking at each pairwise relationship individually but also not as precise as the one presented here. Especially for a lot of features such a matrix can get too big to grasp the subtle details. If this is necessary depends on the use case. A very good option if one wants an initial overview is the function `GGally::ggpairs`. So in the following each pairwise relationship will be covered.

4.1.5.1 Date of joining vs. the others



No major relationship can be detected here.

4.1.5.2 Gender vs. the remaining

Contingency tables for the comparison of two binary features:

```
# Gender vs Company type
table(burnout_train$gender, burnout_train$company_type)
```

```
##
##          Product Service
##   Female    3086    5953
##   Male      2916    5345
```

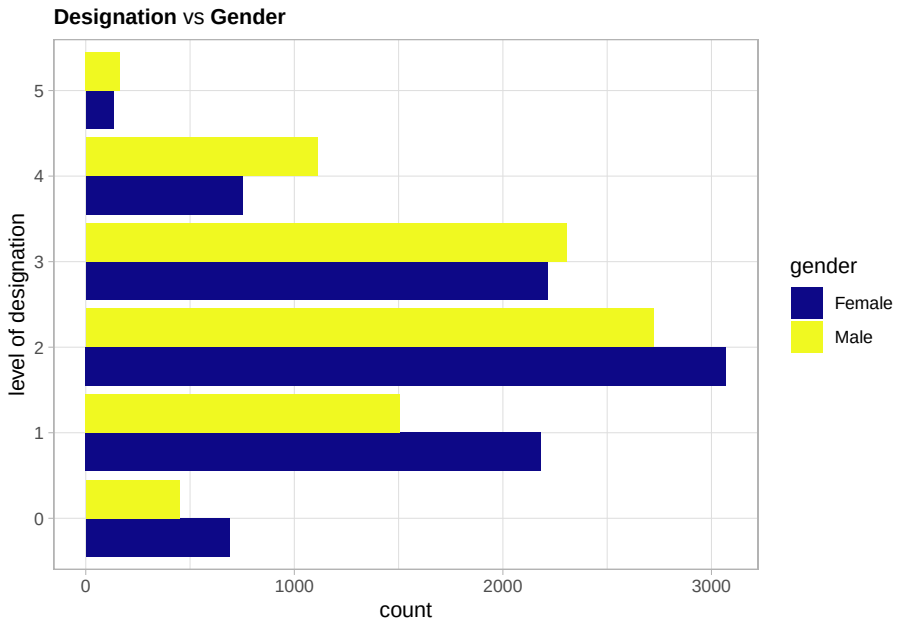
No huge tendency visible.

```
# Gender vs Work from home setup
table(burnout_train$gender, burnout_train$wfh_setup_available)
```

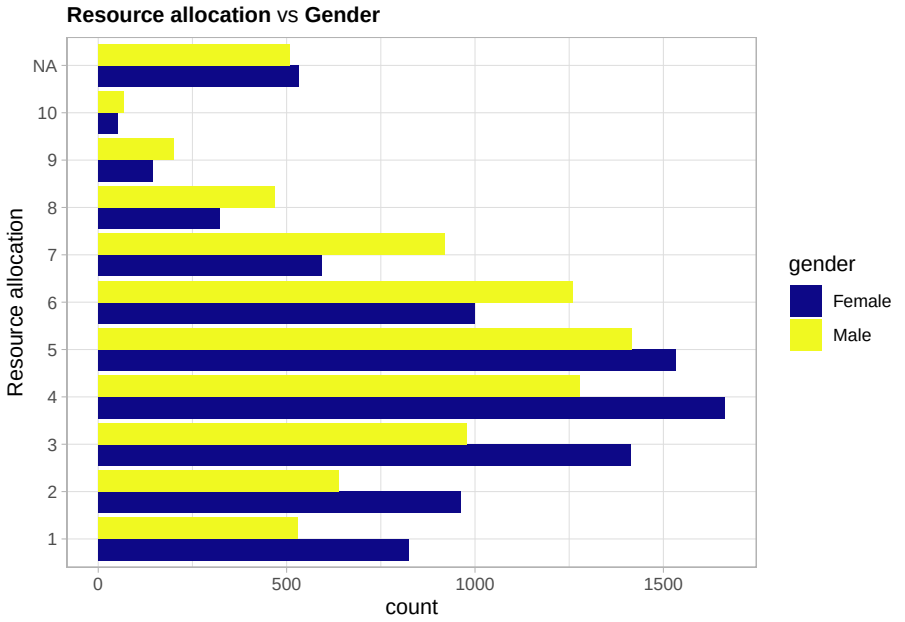
```
##
##            No  Yes
##   Female 3878 5161
##   Male   4104 4157
```

Slightly more women have a work from home setup available.

Now the ordinal variables:

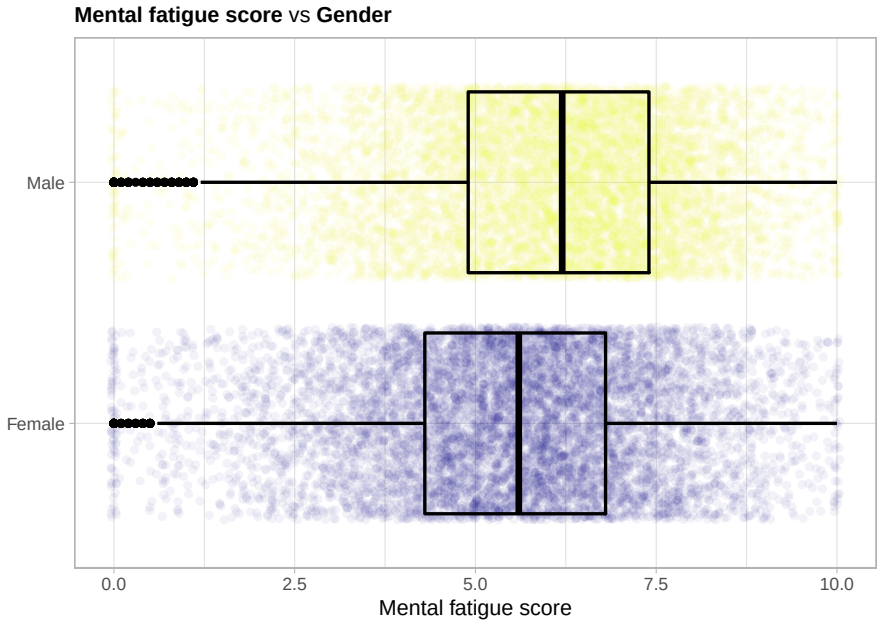


It has to be noted again that female and male emplyees are almost equally represented in the data set. Thus one can see from the above plot that the biggest difference in distribution is for the levels 1 and 4 with opposing effects. While male employees more often have a quite high designation of 4 females are the much more frequent employee with designation level 1.



Here a major shift in distribution is visible towards men getting more resources allocated to them. This reflects the society that still promotes men much more often to high paying jobs that most often come with resource responsibility.

Now the mental fatigue score:



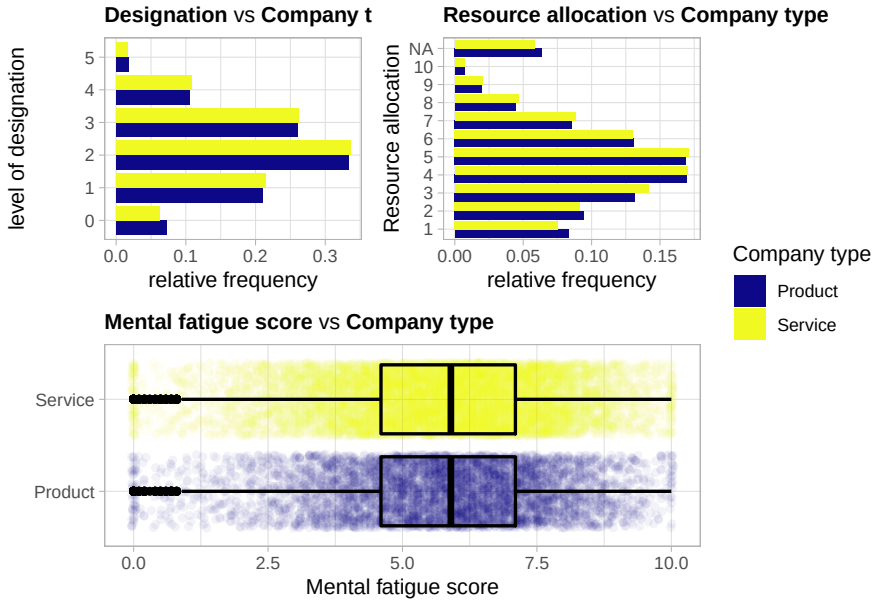
This is of course very similar to the main effect of the `gender` variable as the outcome and the feature `mental_fatigue_score` are highly linearly correlated.

4.1.5.3 Company type vs. the remaining

```
# Company type vs Work from home setup
table(burnout_train$company_type, burnout_train$wfh_setup_available)
```

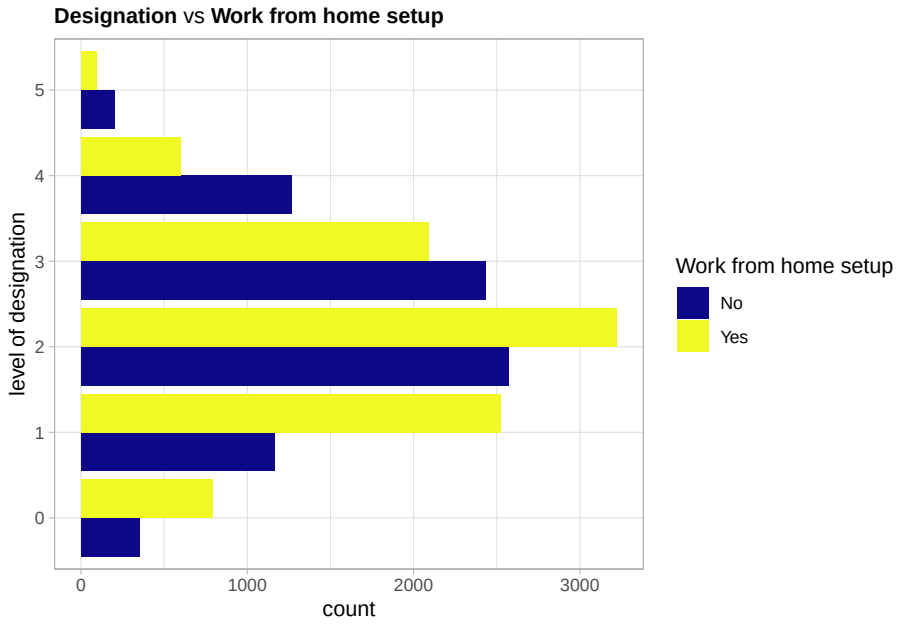
```
##
##              No  Yes
##   Product 2805 3197
##   Service 5177 6121
```

No notable trend.

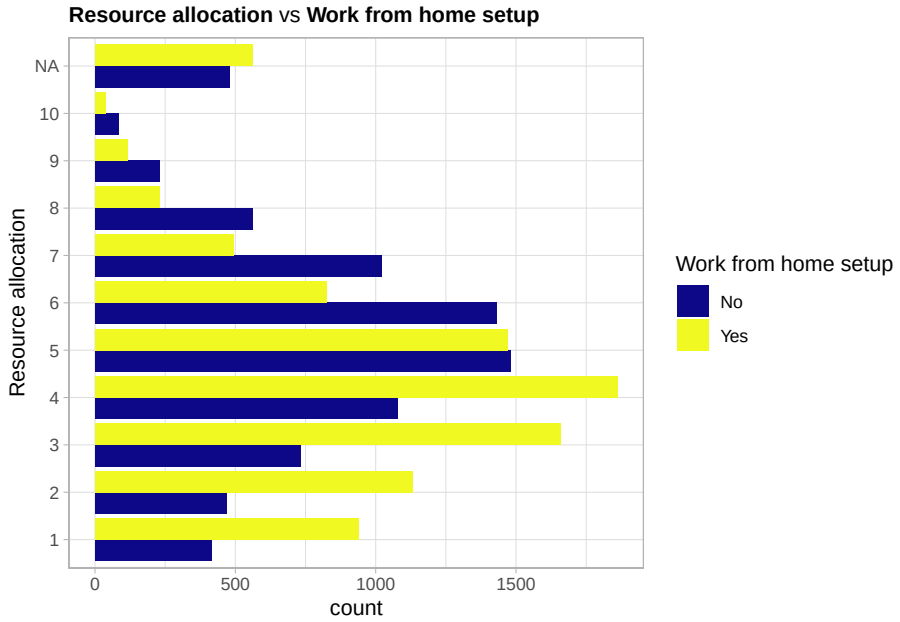


No trend here either.

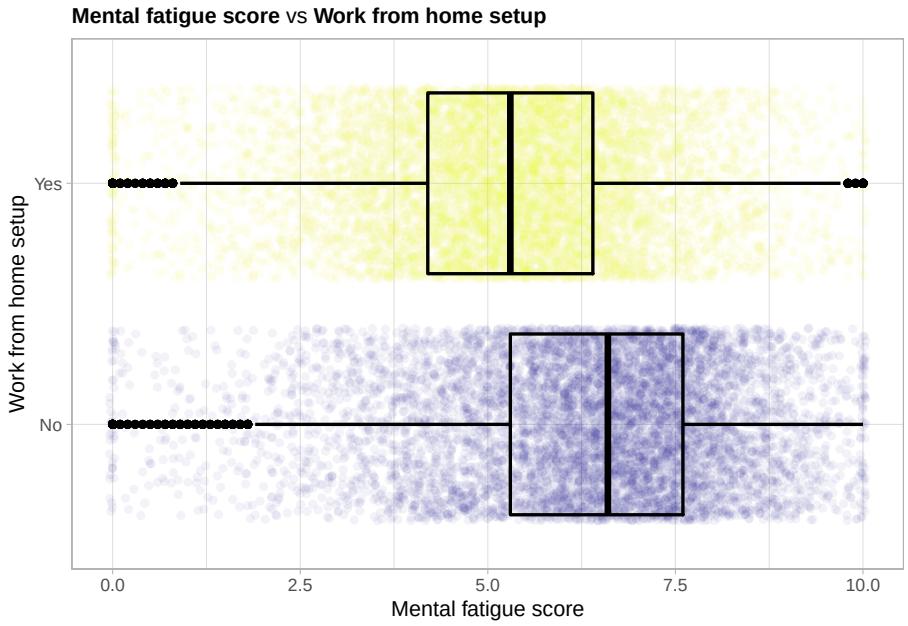
4.1.5.4 Work from home setup vs. the remaining



A work from home setup is way more often available for employees with a lower designation (≤ 2).

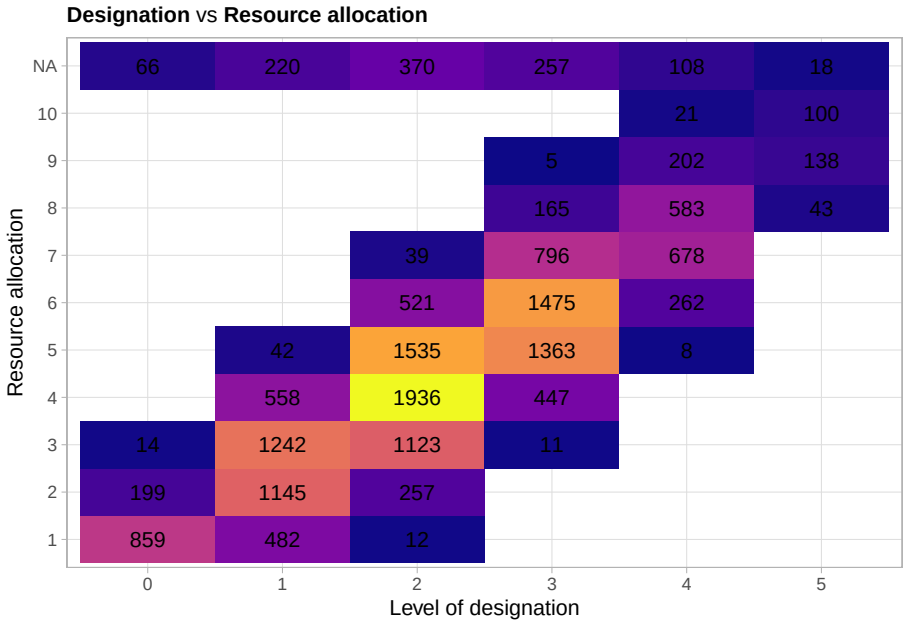


The same structure as in the previous comparison is visible here again. Employees with a lower amount of resources allocated to them have more often a work from home setup available. This could be due to the fewer responsibilities they have in the business.



Again this is of course very similar to the main effect of the `wfh_setup_available` variable as the outcome and the feature `mental_fatigue_score` are highly linearly correlated.

4.1.5.5 Designation vs the remaining



Here a strong quite linear relationship is visible. This is sensible as often more resource responsibility is given to employees with high designation.

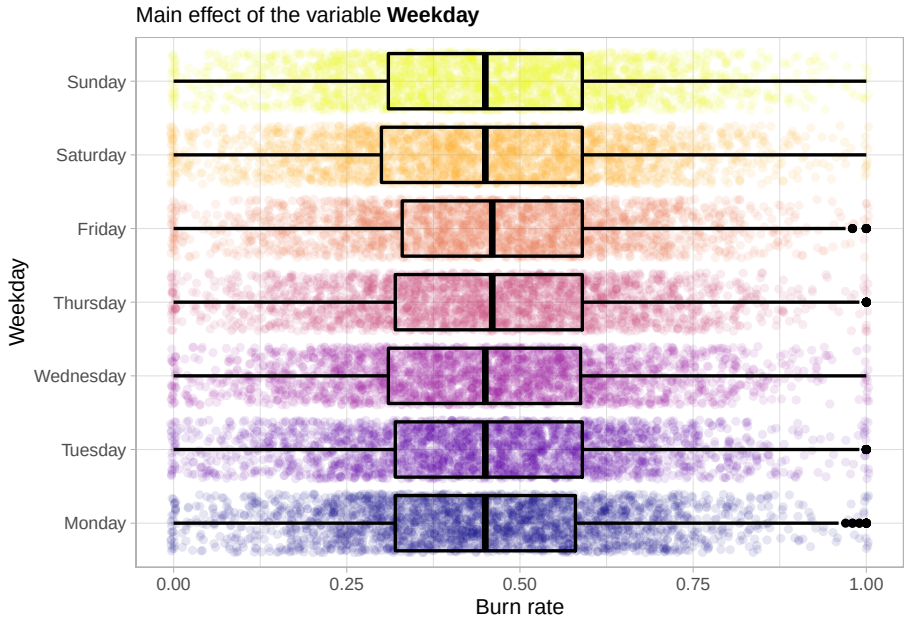
The last two relationships will be omitted here as both for the variable `designation` as well as for `resource_allocation` the comparison with `mental_fatigue_score` will be very similar to the main effect of the two variables. This comes again from the high correlation of the latter with the outcome.

Overall some stronger and mainly less strong relationships between the predictors could be detected. Not like in ordinary least squares regression for gradient tree boosting no decorrelation and normalization of the features is needed. But before one fixes the pre-processing one can try to extract some more information from some features through some feature engineering.

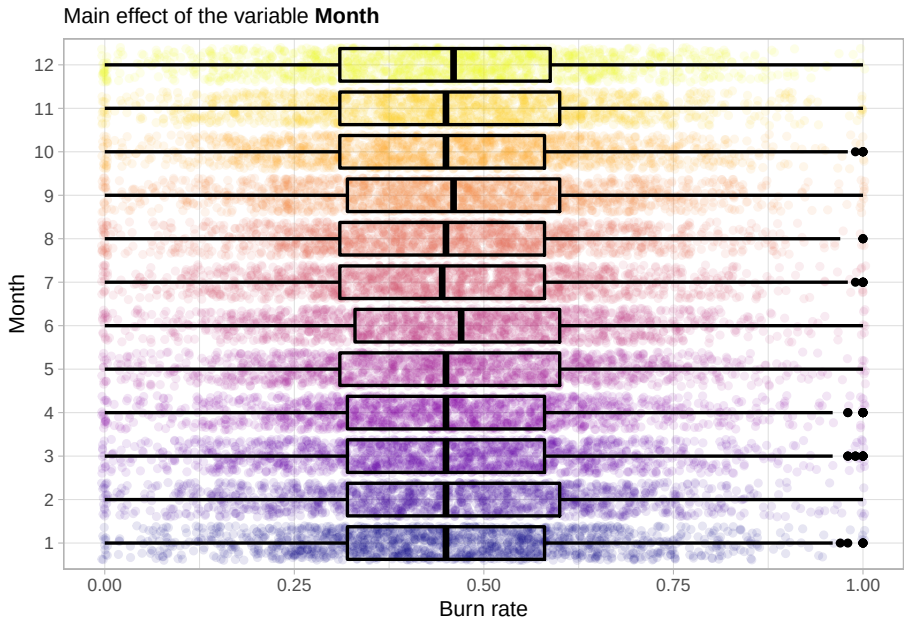
4.1.6 Some feature engineering

The only variable that allows for reasonable feature engineering is the date of joining predictor. One can try to extract some underlying patterns and see if an effect on the outcome is visible.

First extract the day of the week:



No main effect is visible here. So try the month next.



Again no main effect is visible.

Nevertheless one can include those two variables into the model because as mentioned before tree-based models actually perform a feature selection at each

split. So including these just comes at a small computational cost. This minimal pre-processing that is needed when dealing with tree-based models is actually one of its biggest strengths. It is very robust against any kind of weird selection of features with different scales for example. This is one of the reasons, beside the strong predictive power, for the heavy use of such models in data mining applications.(Hastie et al., 2009)

4.1.7 Create the recipe

A recipe is an object that defines a series of steps for data pre-processing and feature engineering.

```
### recipe for xgboost (nominal variables must be dummy variables)
# define outcome, predictors and training data set
burnout_rec_boost <- recipe(burn_rate ~ date_of_joining + gender +
                            company_type + wfh_setup_available +
                            designation + resource_allocation +
                            mental_fatigue_score,
                            data = burnout_train) %>%
  # extract the date features day of the week and month
  step_date(date_of_joining, features = c("dow", "month")) %>%
  # dummify all nominal features
  step_dummy(all_nominal()) %>%
  # encode the date as integers
  step_mutate(date_of_joining = as.integer(date_of_joining))

### recipe for xgboost (nominal variables must be dummy variables)
### HERE the TRANSFORMED target
burnout_rec_boost_trans <- recipe(burn_rate_trans ~ date_of_joining + gender +
                            company_type + wfh_setup_available +
                            designation + resource_allocation +
                            mental_fatigue_score,
                            data = burnout_train) %>%
  # extract the date features day of the week and month
  step_date(date_of_joining, features = c("dow", "month")) %>%
  # dummify all nominal features
  step_dummy(all_nominal()) %>%
  # encode the date as integers
  step_mutate(date_of_joining = as.integer(date_of_joining))

### recipe for a random forest model for comparison (no dummy encoding needed)
# same as above without dummification but here the na's have to be
# imputed (here via knn)
burnout_rec_rf <- recipe(burn_rate ~ date_of_joining + gender +
                         company_type + wfh_setup_available +
```

```
                        designation + resource_allocation +
                        mental_fatigue_score,
                        data = burnout_train) %>%
  step_string2factor(all_nominal()) %>%
  step_impute_knn(resource_allocation, neighbors = 5) %>%
  step_impute_knn(mental_fatigue_score, neighbors = 5) %>%
  step_date(date_of_joining, features = c("dow", "month")) %>%
  step_mutate(date_of_joining = as.integer(date_of_joining))
```

4.2 Insurence data

The insurance data set is part of the book *Machine Learning with R* by Brett Lantz. It can be downloaded here: https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/insurance.csv

```
# load the data
insurance_data <- read_csv("_data/insurance.csv")
```

4.2.1 Train-test split

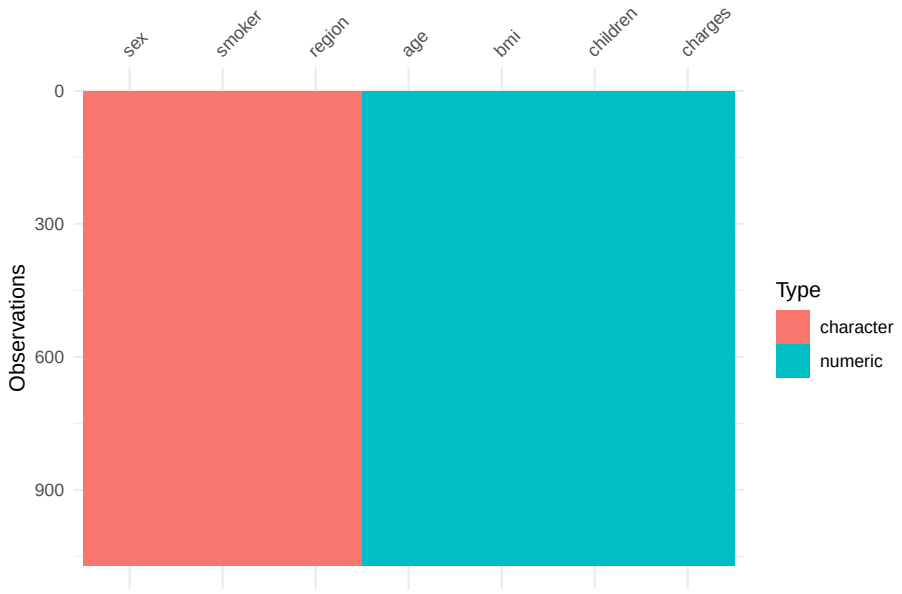
```
set.seed(2)
ins_split <- rsample::initial_split(insurance_data, prop = 0.80)
ins_train <- rsample::training(ins_split)
ins_test  <- rsample::testing(ins_split)
```

The training data set contains 1070 rows and 7 variables.

The test data set contains 268 observations and naturally also 7 variables.

4.2.2 A general overview

A general visualization of the whole data set in order to detect missing values.



So there are no missing values!

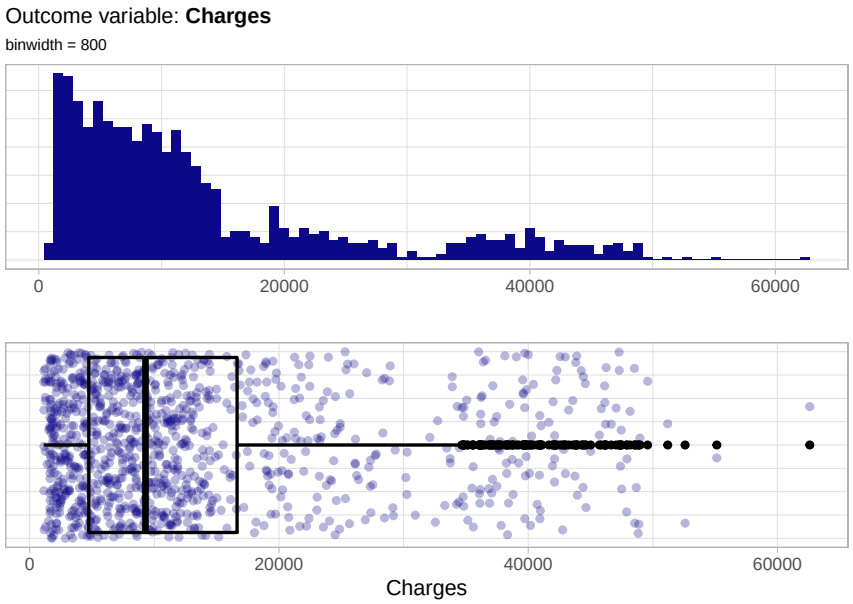
4.2.3 What about the outcome?

`charges`: Individual medical costs billed by health insurance.

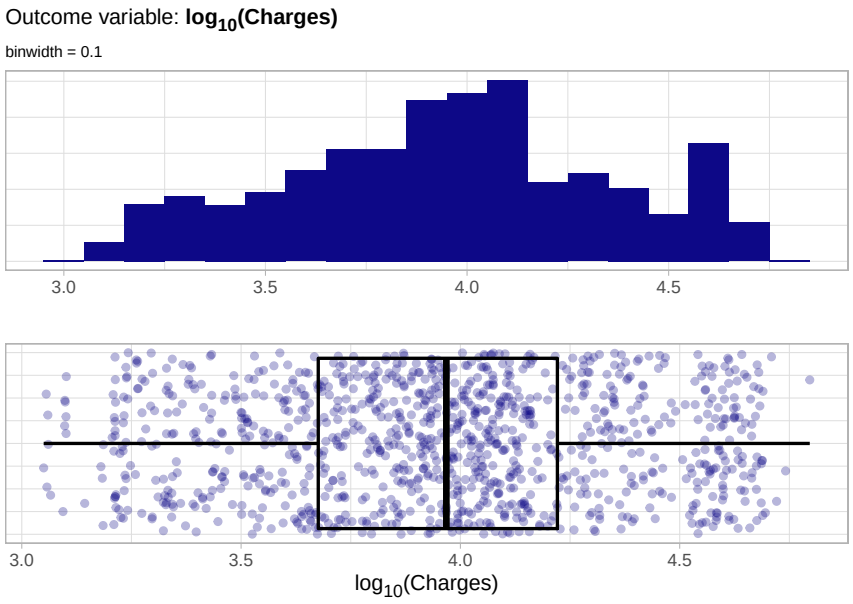
The five point summary below shows that the no invalid values i.e. negative ones are in the data.

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1122    4741    9289   13316   16640   62593
```

Now the distribution of the outcome.



Not like the `burn_rate` previously this target distribution is not at all symmetrical but highly right skewed. A natural thing to do would be a log transformation of the outcome. The resulting `log10_charges` outcome variable is shown below.



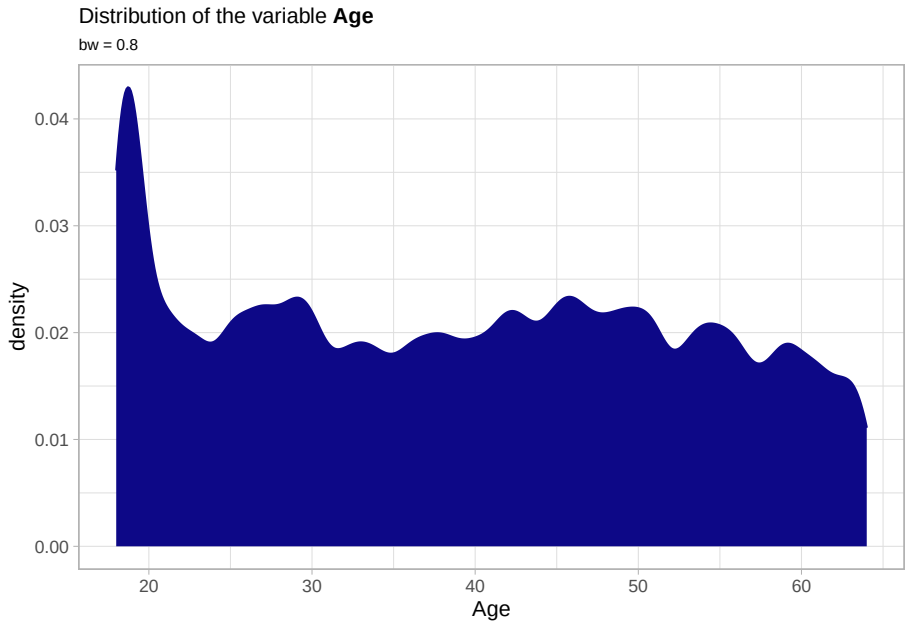
Although such a transformation of the outcome variable is not needed for tree-

based modeling it can make the job of the algorithm somewhat easier.

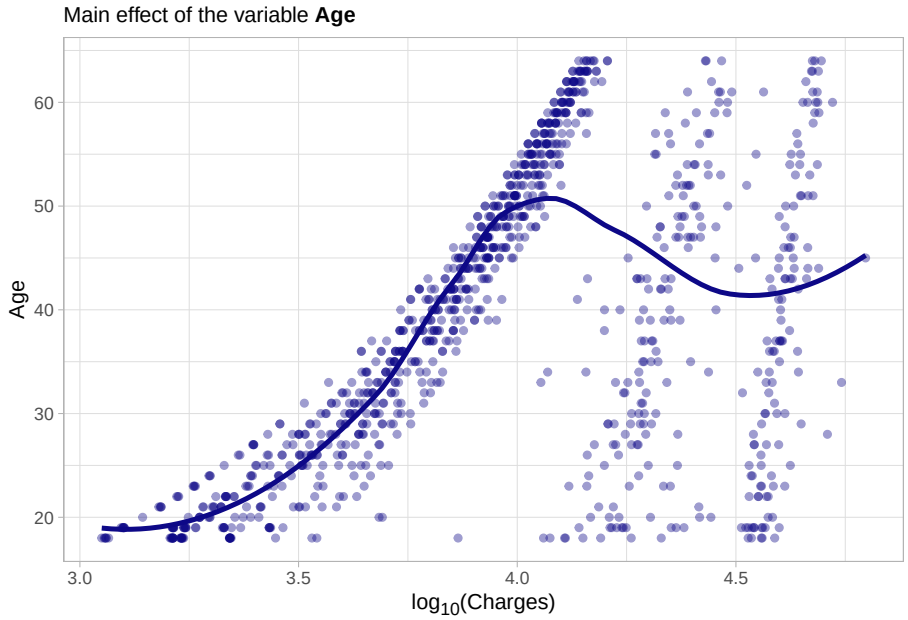
4.2.4 Distribution and main effects of the predictors

4.2.4.1 Age

`age`: The age of the insurance contractor. This is naturally a continuous variable.



A wide range of ages is covered. Notably there is a peak at roughly 18 which means that many fresh adults were observed in this data set.



There seems to be a strong main effect although it does not seem to be linear. The general trend is that older contractors generally accumulate more medical costs. This is very intuitive.

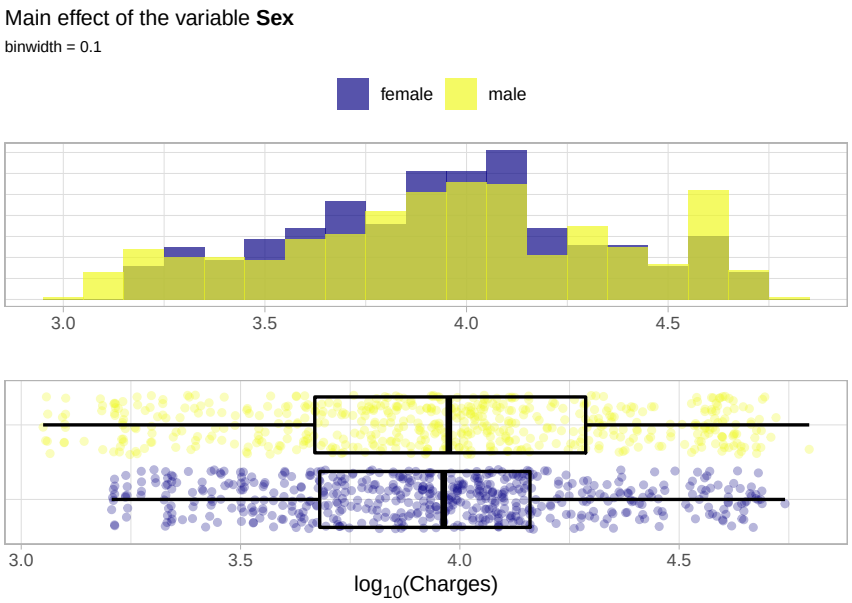
4.2.4.2 Sex

`sex`: The insurance contractors gender. Here either female or male. This means it is a binary variable and will be treated as such.

```
summary(as.factor(ins_train$sex))
```

```
## female   male
##    544    526
```

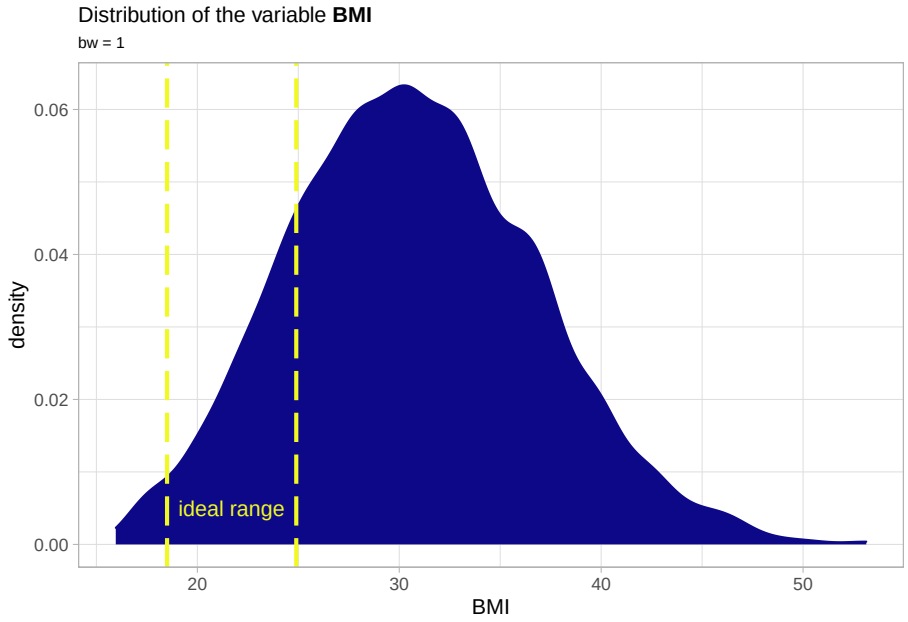
The classes are very well balanced. Now the main effect.



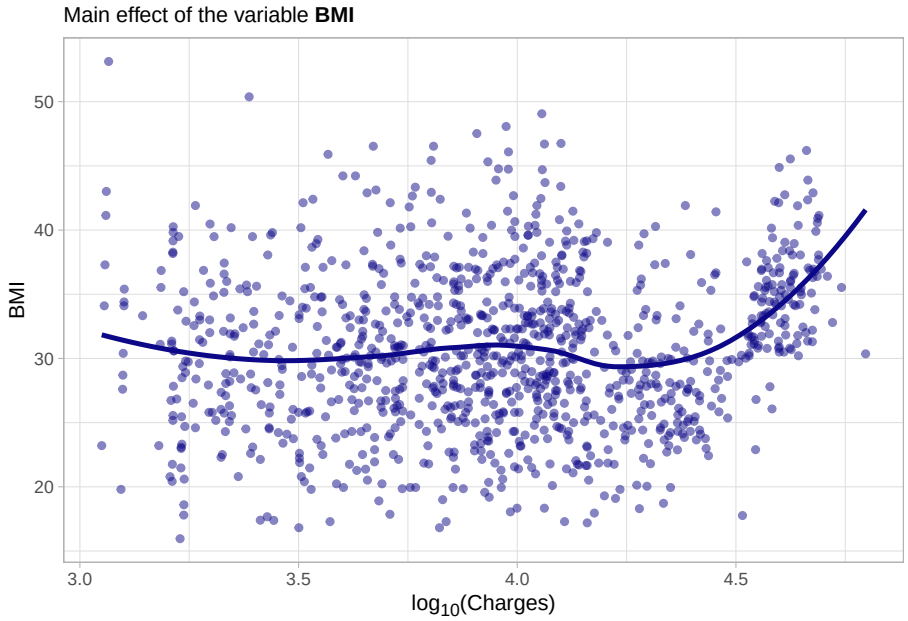
No notable difference can be detected here.

4.2.4.3 Body mass index

`bmi`: The body mass index is providing an understanding of the body composition. It is a ratio composed out of the weight which is divided by the height $\frac{kg}{m^2}$. Ideally the ratio is between 18.5 and 24.9. The variable is obviously a continuous variable.



The distribution is bell-shaped and symmetrical roughly around a bmi of 30 which is above the ideal range. Actually only a small amount of the data falls into the normal range here. Moreover the right tail is heavier than the left one. Now a look at the main effect of the variable.



With some fantasy one can grasp some non-linear patterns on the right side of the plot but beside that no strong main effect is visible here.

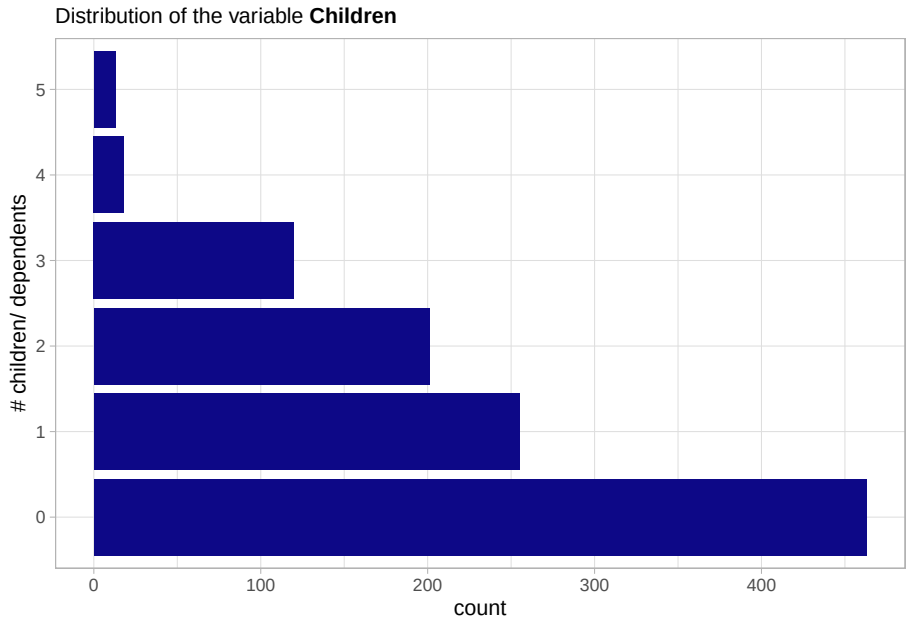
4.2.4.4 Number of children

`children`: The number of children or dependents covered by the health insurance.

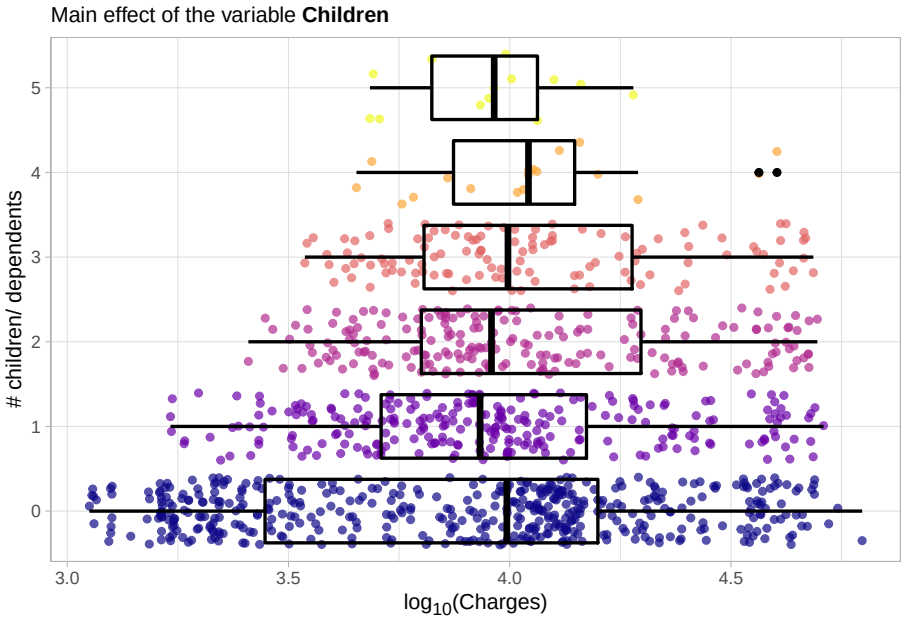
```
# unique values of the feature
unique(ins_train$children)
```

```
## [1] 0 1 2 3 4 5
```

This could be treated as categorical but as there is a natural ordering it will be encoded by the integers so the treatment is like the one of a continuous feature.



As one would also think the more children the lower the number of observed values. Especially the numbers greater than 3 are not well represented. If encoded by one-hot-encoding one would have to think about removing these then near-zero-variance variables. But as they will be encoded in a continuous way this is no problem at all. A look at the main effects can now strengthen or weaken this hypothesis of a natural ordering.



This plot is quite similar to the main effect plot for the `age` feature. As most likely (will be checked later) the age is positively correlated with the number of children one can observe a rise of the minimal observed charges towards a greater amount of children. The upper two boxplots are built with just a few observations so they should not be interpreted in great detail. Overall there seems to be some kind of main effect.

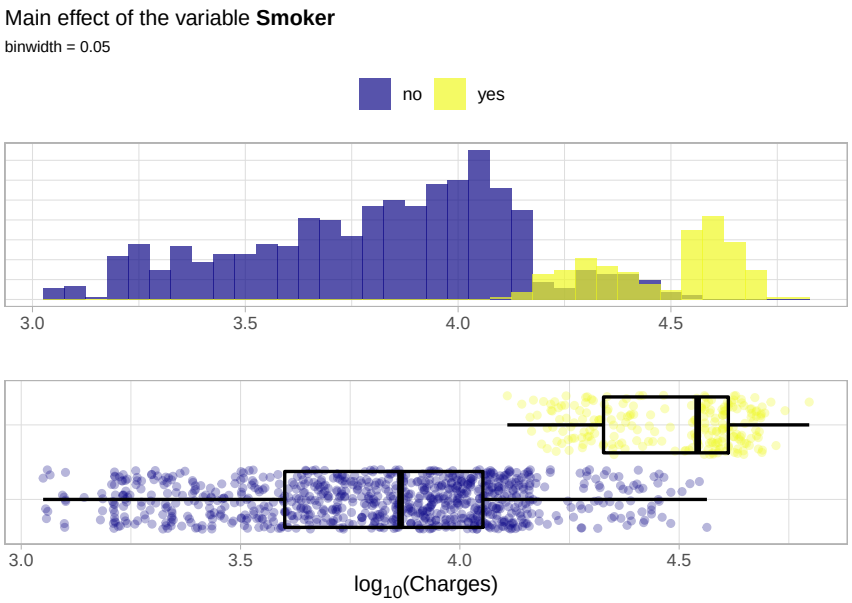
4.2.4.5 Smoking

`smoker`: Is the contractor smoking? Of course a binary variable.

```
summary(as.factor(ins_train$smoker))
```

```
##  no yes
## 849 221
```

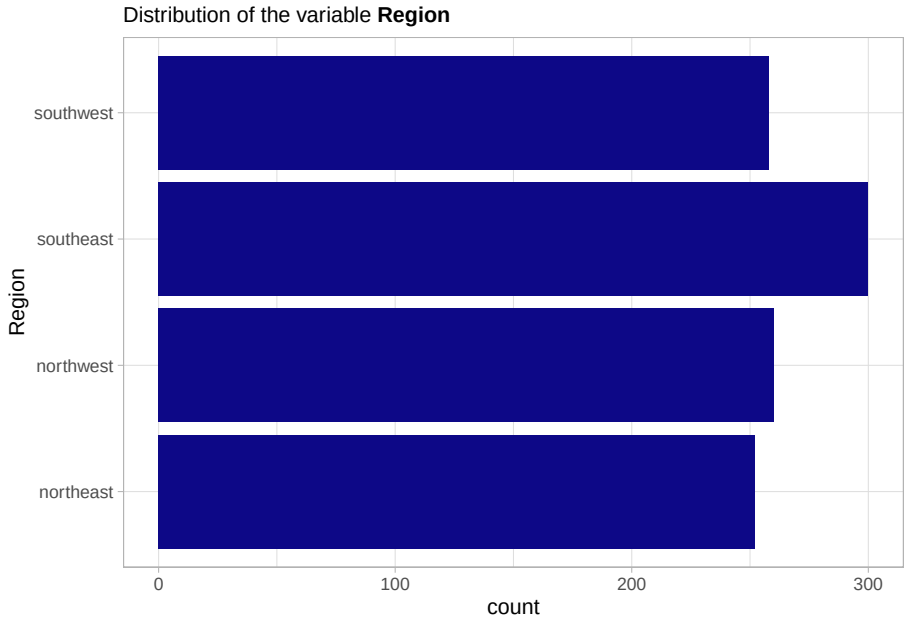
The classes are not balanced but the class of the smokers is still represented with a good amount of observations. Now a look at the main effect.



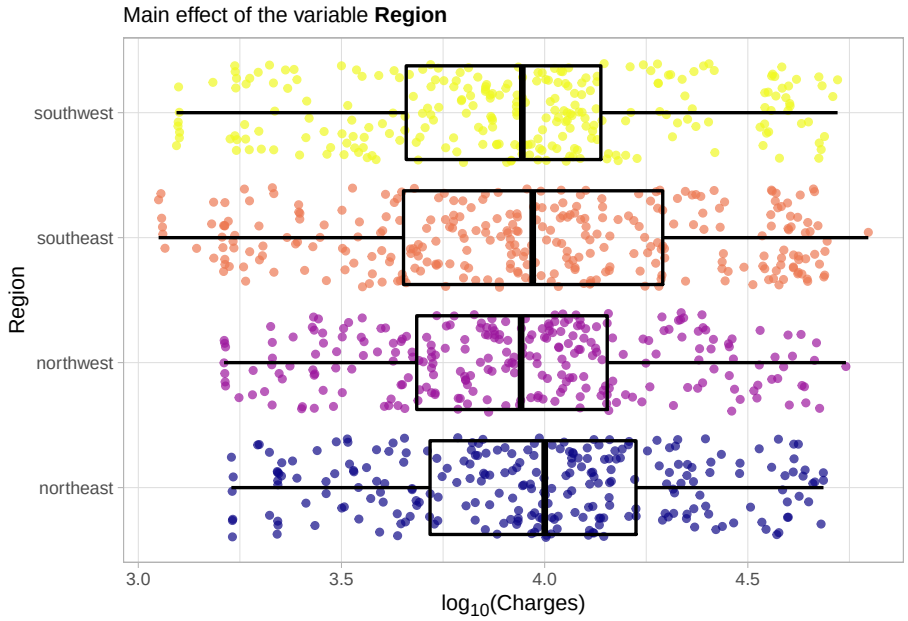
This main effect is as drastic as it is intuitive. Smoking seems to definitely increases the charges. This means that this variable has probably a lot of predictive power.

4.2.4.6 Region

`region`: The beneficiary's residential area in the US. Either northeast, southeast, southwest or northwest. This definitely is a categorical variable.



The four regions are balanced. Now to the main effect.

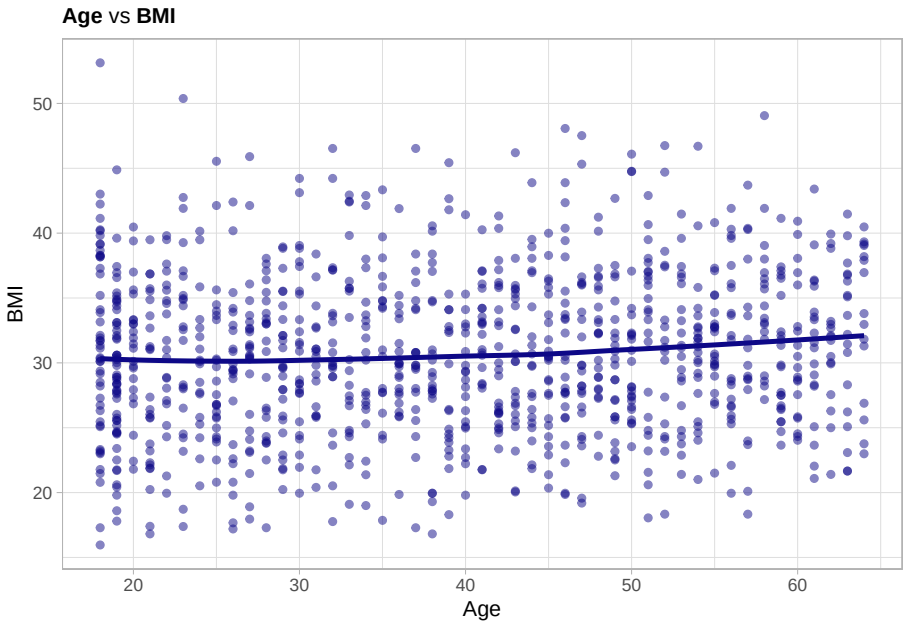


No important main effect is detectable from this plot.

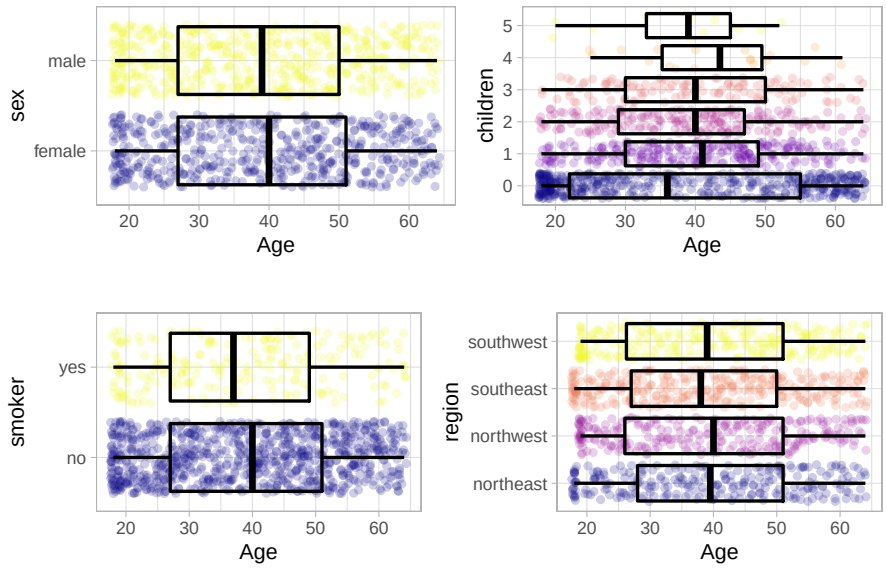
4.2.5 Relationships between the predictors

4.2.5.1 Age vs the others

First the continuous one: `bmi`

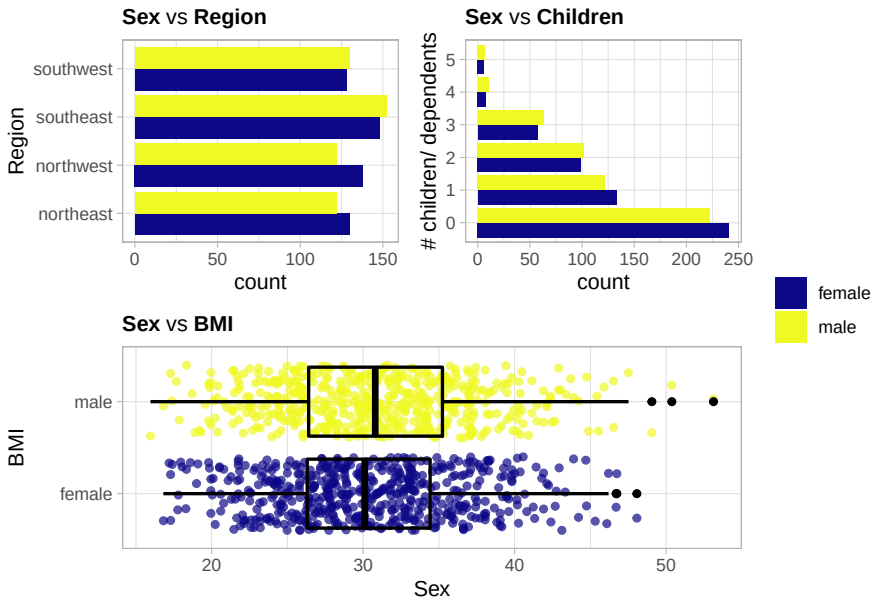


No relationship detectable. The pearson correlation is with 0.089 also low.



The most interesting take away from these four plots is that the hypothesis about the age of the contractors with children seem to be okish except for the ones with more than 3 children. But again this counterintuitive behavior could also be due to the few samples. At this point one might think about encoding the children variable as a categorical but in the following it will be left continuous.

4.2.5.2 Sex vs the remaining



No notable differences.

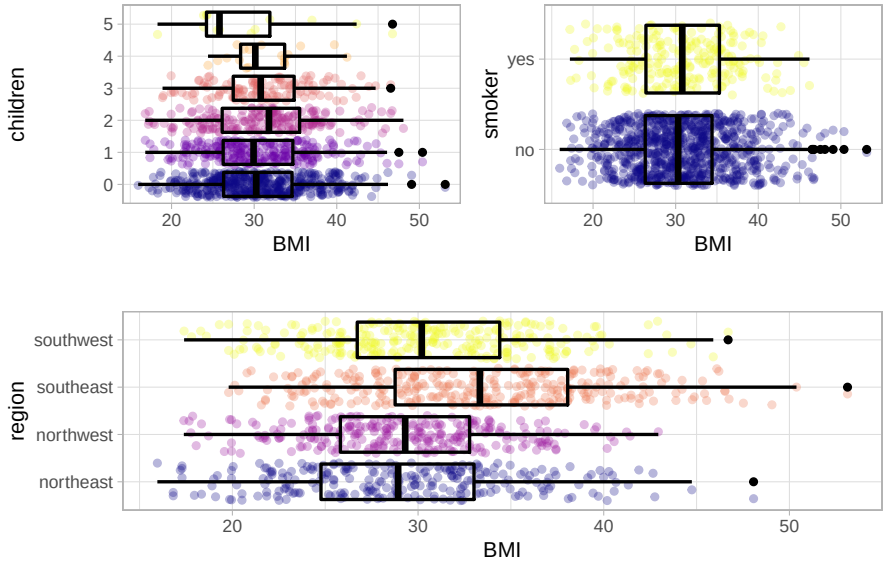
Contingency table for the binary variable `smoker`:

```
# sex vs smoker
table(ins_train$sex, ins_train$smoker)
```

```
##
##           no yes
##   female 448  96
##   male   401 125
```

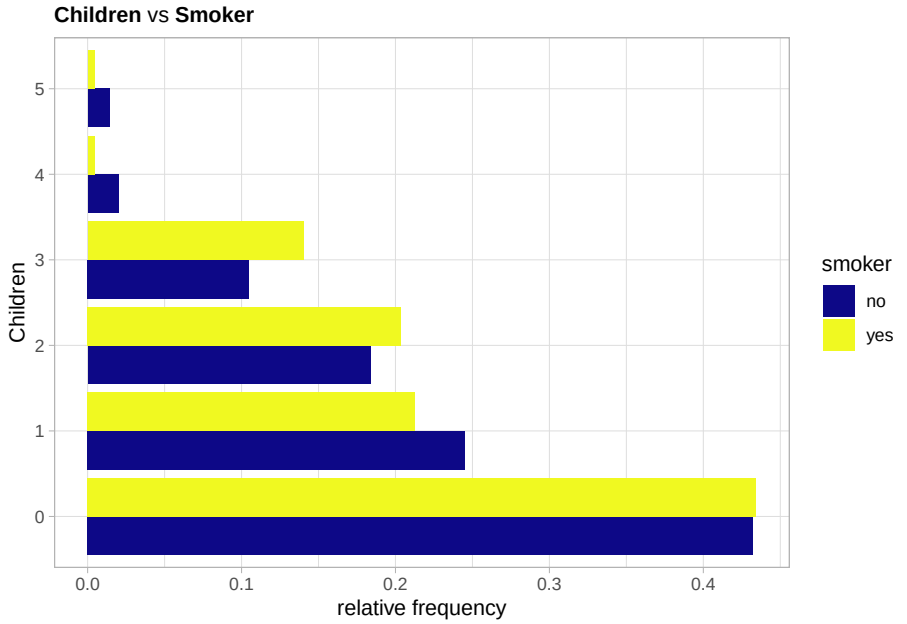
Slightly more men smoke but the difference is smallish.

4.2.5.3 BMI vs the remaining

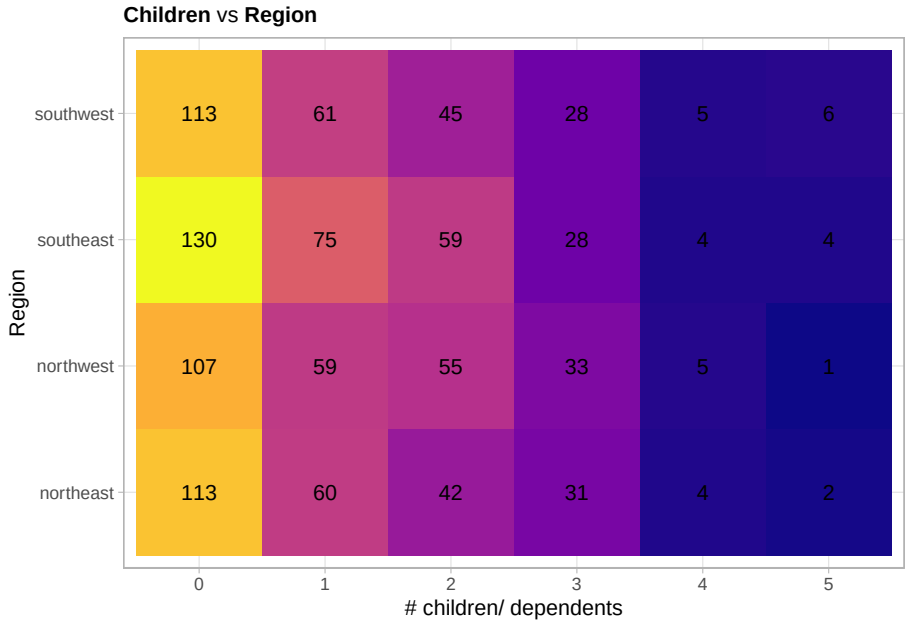


The most notable fact here is that southeast of the US seems to be a little more overweight than the rest.

4.2.5.4 Children vs the remaining

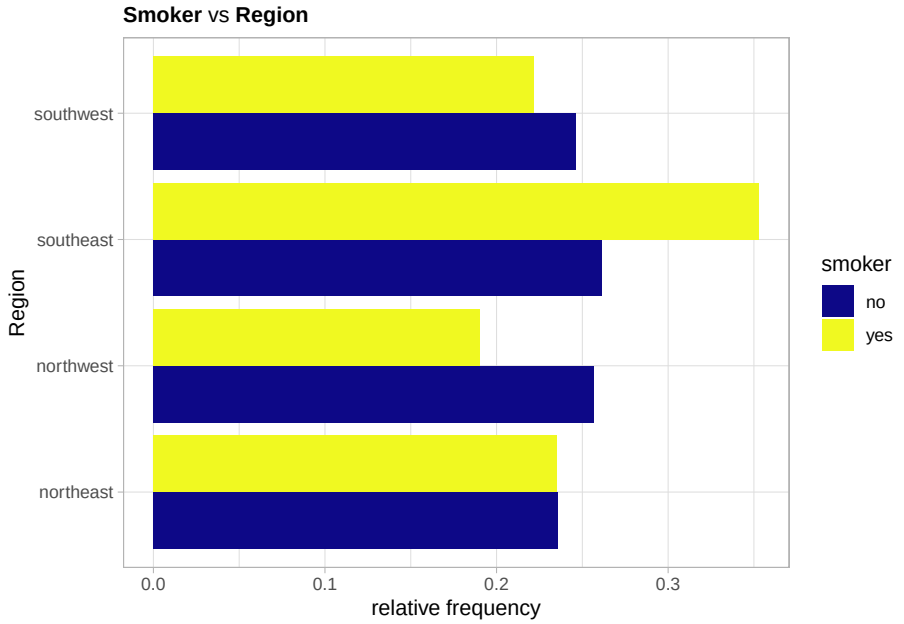


In relative terms actually the contractors with three children smoke the most but the other levels seem quite balanced.



Here no trend is visible.

4.2.5.5 Smoker vs Region



So the southeast is not only the most overweight region but also the one with the most smokers in relative terms.

This concludes the tour of the pairwise relationships. Of course such an in detail look at all pairwise relationships for both data sets was only possible because there are quite few predictors and is not always needed in this extend. Besides a crystal clear understanding of the data one sees that there is not much room left for feature engineering for the insurance data set. Thus one can go on and define the recipe.

4.2.6 Create the recipe

Transformations on the outcome variable are not good practice within a recipe thus this will be done now before hand by adding a new feature i.e. `log10_charges` to the train and test data set.

```
# add the log transformed outcome variable to the data
ins_train$log10_charges <- log10(ins_train$charges)
ins_test$log10_charges <- log10(ins_test$charges)
```

```
### recipe for xgboost (nominal variables must be dummy variables)
# define outcome, predictors and training data set
ins_rec_boost <- recipe(log10_charges ~ age + sex +
                            bmi + children +
                            smoker + region,
                            data = ins_train) %>%
  # dummify all nominal features (sex, smoker, region)
  step_dummy(all_nominal())

### recipe for a random forest model for comparison (no dummy encoding needed)
# same as above without dummification
ins_rec_rf <- recipe(log10_charges ~ age + sex +
                           bmi + children +
                           smoker + region,
                           data = ins_train)
```

Having now all the recipes ready one can proceed with modeling. Finally!

Chapter 5

Let's boost the models

Beside the XGBoost model also a random forest model will be fitted here for comparison. The whole modeling approach and procedure is closely following the principles that are outlined in the **great** book *Tidy modeling with R* by Max Kuhn and Julia Silge.

For the modeling three additional packages are required.(Chen et al., 2021; Wright and Ziegler, 2017; Greenwell and Boehmke, 2020)

```
library(xgboost) # for the xgboost model
library(ranger) # for the random forest model
# + the vip package but it will not be loaded into
# the namespace
```

For parallel computations the `doParallel` package is used.(Corporation and Weston, 2019) This is most useful for the tuning part.

```
library(doParallel)

# Create a cluster object and then register:
cl <- makePSOCKcluster(2)
registerDoParallel(cl)
```

First one has to set a **metric** for the evaluation of the final performance. As one knows from the EDA that not a lot of outliers are present in the data one can leave the default for XGBoost as the optimization metric for regression which is the root mean squared error (RMSE) which basically corresponds to the L_2 loss. Besides also the mean average error (MAE) which is kind of the L_1 norm will be covered but the optimization and tuning will focus on the RMSE.

5.1 Burnout data

5.1.1 Baseline models

The first thing is to set up the trivial **baseline models**.

```
# the trivial intercept only model:
bout_predict_trivial_mean <- function(new_data) {
  rep(mean(burnout_train$burn_rate), nrow(new_data))
}

# the trivial scoring of mental fatigue score (if missing intercept model)
bout_predict_trivial_mfs <- function(new_data) {
  # these two scoring parameters are correspondint to the
  # simple linear regression model containing only one predictor
  # i.e. mfs
  pred <- new_data[["mental_fatigue_score"]] * 0.097 - 0.1
  pred[is.na(pred)] <- mean(burnout_train$burn_rate)
  pred
}
```

The predictions of these baseline models on the test data set will be compared with the predictions of the tree-based models that will be constructed.

5.1.2 Model specification

```
# Models:

# Random forest model for comparison
bout_rf_model <- rand_forest(trees = tune(),
                             mtry = tune()) %>%
  set_engine("ranger") %>%
  set_mode("regression")

# XGBoost model
bout_boost_model <- boost_tree(trees = tune(),
                               learn_rate = tune(),
                               loss_reduction = tune(),
                               tree_depth = tune(),
                               mtry = tune(),
                               sample_size = tune(),
                               stop_iter = 10) %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

```
# Workflows: model + recipe

# Random Forest workflow
bout_rf_wflow <-
  workflow() %>%
  add_model(bout_rf_model) %>%
  add_recipe(burnout_rec_rf)

# XGBoost workflow with the untransformed target
bout_boost_wflow <-
  workflow() %>%
  add_model(bout_boost_model) %>%
  add_recipe(burnout_rec_boost)

# XGBoost workflow with the transformed target (empirical logit)
bout_boost_wflow_trans <-
  workflow() %>%
  add_model(bout_boost_model) %>%
  add_recipe(burnout_rec_boost_trans)
```

5.1.3 Tuning

For the hyperparameter tuning one needs validation sets to monitor the models on unseen data. To do this 5-fold cross validation (CV) is used here.

```
# Create Resampling object
set.seed(2)
burnout_folds <- vfold_cv(burnout_train, v = 5)
```

Now adjust or check the hyperparameter ranges that the tuning will use. First the Random forest model.

```
# Have a look at the hyperparameters that have to be tuned and finalize them
bout_rf_params <- bout_rf_wflow %>%
  parameters()

bout_rf_params
```

```
## Collection of 2 parameters for tuning
##
##  identifier  type    object
##        mtry  mtry nparam[?]
##       trees trees nparam[+]
```

```
##
## Model parameters needing finalization:
##    # Randomly Selected Predictors ('mtry')
##
## See `?dials::finalize` or `?dials::update.parameters` for more information.
```

This shows that the `mtry` hyperparameter has to be adjusted depending on the data. Moreover with the `dials::update` function one can manually set the ranges that should be used for tuning.

```
# default range for tuning is 1 up to 2000 for the trees argument
# set the lower bound of the range to 100. Then finalize the
# parameters using the training data.
bout_rf_params <- bout_rf_params %>%
  update(trees = trees(c(100, 2000))) %>%
  finalize(burnout_train)
bout_rf_params
```

```
## Collection of 2 parameters for tuning
##
##  identifier  type    object
##        mtry  mtry nparam[+]
##       trees trees nparam[+]
```

```
bout_rf_params %>% pull_dials_object("mtry")
```

```
## # Randomly Selected Predictors (quantitative)
## Range: [1, 10]
```

```
bout_rf_params %>% pull_dials_object("trees")
```

```
## # Trees (quantitative)
## Range: [100, 2000]
```

Now this parameter object is ready for tuning and the same steps have to be performed on the main boosting workflow. The parameter set for the untransformed target can then also be used for the transformed one.

```
bout_boost_params <- bout_boost_wflow %>%
  parameters()

bout_boost_params
```

```
## Collection of 6 parameters for tuning
##
##      identifier           type    object
##            mtry           mtry nparam[?]
##           trees          trees nparam[+]
##      tree_depth     tree_depth nparam[+]
##      learn_rate     learn_rate nparam[+]
##  loss_reduction loss_reduction nparam[+]
##     sample_size    sample_size nparam[+]
##
## Model parameters needing finalization:
##    # Randomly Selected Predictors ('mtry')
##
## See `?dials::finalize` or `?dials::update.parameters` for more information.
```

```
# first a look at the default ranges
trees()
```

```
## # Trees (quantitative)
## Range: [1, 2000]
```

```
tree_depth()
```

```
## Tree Depth (quantitative)
## Range: [1, 15]
```

```
learn_rate()
```

```
## Learning Rate (quantitative)
## Transformer:  log-10
## Range (transformed scale): [-10, -1]
```

```
loss_reduction()
```

```
## Minimum Loss Reduction (quantitative)
## Transformer:  log-10
## Range (transformed scale): [-10, 1.5]
```

```
sample_size()
```

```
## # Observations Sampled (quantitative)
## Range: [?, ?]
```

So `sample_size` must also be finalized. Again the lower bound on the number of trees will be raised to 100. The other scales are really sensible and will be left as is.

```
bout_boost_params <- bout_boost_params %>%
  update(trees = trees(c(100, 2000))) %>%
  finalize(burnout_train)

bout_boost_params
```

```
## Collection of 6 parameters for tuning
##
##      identifier           type    object
##            mtry           mtry nparam[+]
##           trees          trees nparam[+]
##      tree_depth     tree_depth nparam[+]
##      learn_rate     learn_rate nparam[+]
##  loss_reduction loss_reduction nparam[+]
##     sample_size    sample_size nparam[+]
```

```
bout_boost_params %>%
  pull_dials_object("sample_size")
```

```
## Proportion Observations Sampled (quantitative)
## Range: [0.1, 1]
```

```
bout_boost_params %>%
  pull_dials_object("mtry")
```

```
## # Randomly Selected Predictors (quantitative)
## Range: [1, 10]
```

```
# use the same object for the tuning of the boosted model
# with the transformed target
bout_boost_params_trans <- bout_boost_params
```

Now also these parameter objects are ready to be used. The next step is to define the metrics used for evaluation while tuning.

```
# define a metrics set used for evaluation of the hyperparameters
regr_metrics <- metric_set(rmse, mae)
```

Now the actual tuning begins. The models will be tuned by a space filling grid search. As one has defined ranges for each parameter one can then use different algorithms to construct a grid of combinations of parameter values that tries to best fill the defined ranges by random sampling also in a high dimensional setting. In the following the function `tune::grid_latin_hypercube` is used for this. So basically one tries out all hyperparameter values for each fold and saves the performance of the performance metrics on the hold out fold. These metrics are then aggregated for each combination and with the resulting estimates of performance one can choose the final set of hyperparameters. The more hyperparameters the model has the more tuning rounds might be needed to refine the grid. Besides the classical grid search there are also iterative methods for tuning. These are mainly good to tune a single hyperparameter at once and not for a bunch of them simultaneously. They will not be used in the following.

The **random forest model** goes first with the tuning.

Here there are as seen above only two hyperparameters to be tuned thus 30 combinations should suffice.

```
# took roughly 30 minutes
system.time({
  set.seed(2)
  bout_rf_tune <- bout_rf_wflow %>%
    tune_grid(
      resamples =  burnout_folds,
      grid = bout_rf_params %>%
        grid_latin_hypercube(size = 30, original = FALSE),
      metrics = regr_metrics
    )
})

# visualization of the tuning results (snapshot of the output below)
autoplot(bout_rf_tune) + theme_light()
# this functions shows the best combinations wrt the rmse metric of all the
# combinations in the grid
show_best(bout_rf_tune, metric = "rmse")
```

The visualization alongside the best performing results suggest that a value of `mtry`of 3 and 1000 `trees` should give good results. Thus one can finalize and fit this model.

```
final_bout_rf_wflow <-
  bout_rf_wflow %>%
  finalize_workflow(tibble(
    trees = 1000,
    mtry = 3
```

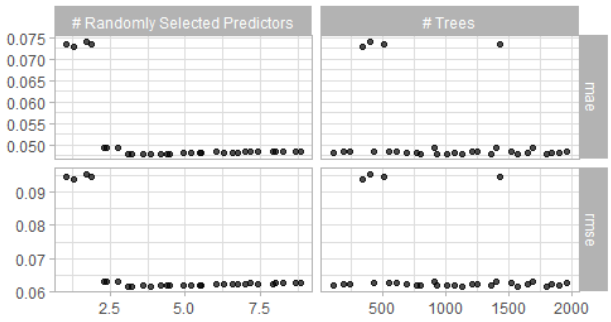


Figure 5.1: Result of a spacefilling grid search for the random forest model.

```
  )) %>%
  fit(burnout_train)
```

Now the main **boosting model**.

First tune only the number of trees in order to detect a number of trees that is 'large enough'. Then tune the tree specific arguments. If one tunes all parameters at the same time the grid grows to large. Moreover a tuning of the trees argument could encourage overfitting. See the book *Hands-on Machine Learning with R* for a detailed explenation of the tuning strategies.(Boehmke and Greenwell, 2019)

The first tuning round:

To display the discussed impact of different maximum tree depths the first grid search for a good number of trees will besides the kind of standard value 6 also be performed for regression stumps i.e. a value of 1 for the maximum tree depth.

```
# tuning grid just for the #trees one with max tree depth 6 and one with
# only stumps for comparison
first_grid_boost_burn_depth6 <- crossing(
  trees = seq(250, 2000, 250),
  mtry = 9,
  tree_depth = 6,
  loss_reduction = 0.000001,
  learn_rate = 0.01,
  sample_size = 1
)

first_grid_boost_burn_stumps <- crossing(
  trees = seq(250, 2000, 250),
  mtry = 9,
```

```
  tree_depth = 1,
  loss_reduction = 0.000001,
  learn_rate = 0.01,
  sample_size = 1
)
```

```
# took roughly 1 minute
system.time({
  set.seed(2)
  bout_boost_tune_first_stumps <- bout_boost_wflow %>%
    tune_grid(
      resamples =  burnout_folds,
      grid = first_grid_boost_burn_stumps,
      metrics = regr_metrics
    )
})


autoplot(bout_boost_tune_first_stumps) + theme_light()
show_best(bout_boost_tune_first_stumps)
```

```
# took roughly 1 minute
system.time({
  set.seed(2)
  bout_boost_tune_first_depth6 <- bout_boost_wflow %>%
    tune_grid(
      resamples =  burnout_folds,
      grid = first_grid_boost_burn_depth6,
      metrics = regr_metrics
    )
})

# plot output is shown in the figure below
autoplot(bout_boost_tune_first_depth6) + theme_light()
show_best(bout_boost_tune_first_depth6)
```

Compare the two different first tuning rounds:

```
# output of this code shown in the figure below
bout_boost_tune_first_stumps %>%
  collect_metrics() %>%
  mutate(tree_depth = 1) %>%
  bind_rows(
    bout_boost_tune_first_depth6 %>%
```

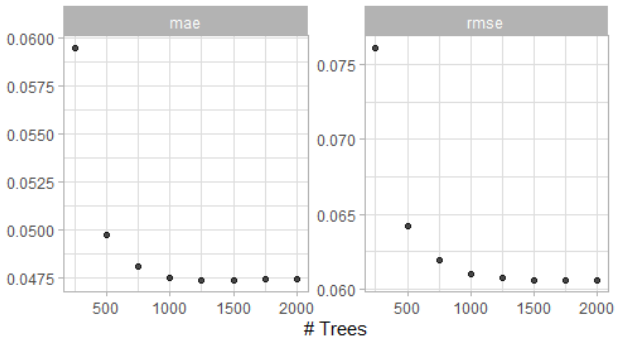


Figure 5.2: Result of the first grid search for the XGBoost model (max depth 6).

```
    collect_metrics() %>%
    mutate(tree_depth = 6)
) %>%
ggplot(aes(x = trees, y = mean, col = factor(tree_depth))) +
geom_point() +
geom_line() +
geom_linerange(aes(ymin = mean - std_err, ymax = mean + std_err)) +
labs(col = "Max tree depth", y = "", x = "Number of trees",
     title = "") +
scale_color_viridis_d(option = "C") +
facet_wrap(~ .metric) +
theme_light()
```

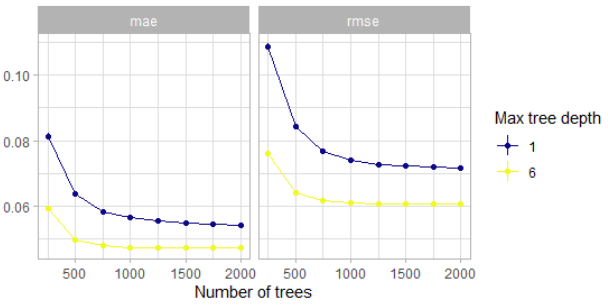


Figure 5.3: Comparison of the initial grid search for the number of trees w.r.t. the maximum tree depth. Actually with tiny confidence bands around the estimates.

This shows that the model with just stumps converges much slower or may even

never reach the low level of when using deeper trees. One reason for this could be that these stumps do not account for interactions and of course the same number of deep trees encode much more information than the same number of stumps.

So 1500 trees should suffice here. Thus the workflow and model will be adjusted accordingly.

```
# fix the number of trees by redefining the boosted model with a
# fixed number of trees.
bout_boost_model <- boost_tree(trees = 1500,
                               learn_rate = tune(),
                               loss_reduction = tune(),
                               tree_depth = tune(),
                               mtry = tune(),
                               sample_size = tune(),
                               stop_iter = 10) %>%
  set_engine("xgboost") %>%
  set_mode("regression")

# update the workflow
bout_boost_wflow <-
  bout_boost_wflow %>%
  update_model(bout_boost_model)

bout_boost_params <- bout_boost_wflow %>%
  parameters() %>%
  finalize(burnout_train)

# reduced hyperparameter space
bout_boost_params
```

```
## Collection of 5 parameters for tuning
##
##      identifier           type    object
##            mtry           mtry nparam[+]
##      tree_depth     tree_depth nparam[+]
##      learn_rate     learn_rate nparam[+]
##  loss_reduction loss_reduction nparam[+]
##     sample_size    sample_size nparam[+]
```

Now perform the first major grid search over all the other hyperparameters.

```
# now tune all the remaining hyperparameters with a large space filling grid
```

```
# took roughly 1.5 hours
system.time({
  set.seed(2)
  bout_boost_tune_second <- bout_boost_wflow %>%
    tune_grid(
      resamples = burnout_folds,
      grid = bout_boost_params %>%
        grid_latin_hypercube(
          size = 200),
      metrics = regr_metrics
    )
})

show_best(bout_boost_tune_second, metric = "rmse")
```

Now refine the grid i.e. the parameter space according to the results of the last grid search and perform a third one.

```
# now tune all the remaining hyperparameters with a refined space filling grid

# took roughly 2 hours
system.time({
  set.seed(2)
  bout_boost_tune_third <- bout_boost_wflow %>%
    tune_grid(
      resamples = burnout_folds,
      grid = bout_boost_params %>%
        update(
          mtry = mtry(c(5,9)),
          tree_depth = tree_depth(c(4,5)),
          learn_rate = learn_rate(c(-1.7, -1.3)),
          loss_reduction = loss_reduction(c(-8,-3)),
          sample_size = sample_prop(c(0.4, 0.9))
        ) %>%
        grid_latin_hypercube(
          size = 200),
      metrics = regr_metrics
    )
})

show_best(bout_boost_tune_third, metric = "rmse")
```

With this final grid search one is ready to finalize the model.

The results of the three grid searches suggest that no column-subsampling should be applied, the maximum tree depth should be 4, the learning rate

should be small but not extremely small (~ 0.02), the loss reduction regularization effect is not needed here (very small) and the sample size for each tree should be set to 0.8.

```
final_bout_boost_wflow <-
  bout_boost_wflow %>%
  finalize_workflow(tibble(
    mtry = 9,
    tree_depth = 4,
    learn_rate = 0.02,
    loss_reduction = 0.0000003,
    sample_size = 0.8
  )) %>%
  fit(burnout_train)
```

Now tune the **transformed outcome boosting model**.

Again start with the `trees`.

```
first_grid_boost_burn_trans <- crossing(
  trees = seq(250, 2000, 250),
  mtry = 9,
  tree_depth = 6,
  loss_reduction = 0.000001,
  learn_rate = 0.01,
  sample_size = 1
)
```

```
# took roughly 1 minute
system.time({
  set.seed(2)
  bout_boost_tune_first_trans <- bout_boost_wflow_trans %>%
    tune_grid(
      resamples =  burnout_folds,
      grid = first_grid_boost_burn_trans,
      metrics = regr_metrics
    )
})

# plot output in the figure below
autoplot(bout_boost_tune_first_trans) + theme_light()
show_best(bout_boost_tune_first_trans)
```

Again 1500 trees should easily suffice.

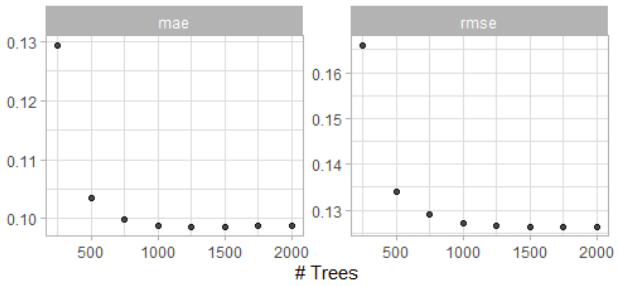


Figure 5.4: Result of the first grid search for the XGBoost model with transformed outcome.

```
# fix the number of trees by redefining the boosted model with a
# fixed number of trees.
bout_boost_model_trans <- boost_tree(trees = 1500,
                                     learn_rate = tune(),
                                     loss_reduction = tune(),
                                     tree_depth = tune(),
                                     mtry = tune(),
                                     sample_size = tune(),
                                     stop_iter = 10) %>%
  set_engine("xgboost") %>%
  set_mode("regression")

# update the workflow
bout_boost_wflow_trans <-
  bout_boost_wflow_trans %>%
  update_model(bout_boost_model_trans)

bout_boost_params_trans <- bout_boost_wflow_trans %>%
  parameters() %>%
  finalize(burnout_train)

# reduced hyperparameter space
bout_boost_params_trans
```

```
## Collection of 5 parameters for tuning
##
##      identifier           type    object
##            mtry           mtry nparam[+]
##      tree_depth     tree_depth nparam[+]
##      learn_rate     learn_rate nparam[+]
##  loss_reduction loss_reduction nparam[+]
```

```
##     sample_size    sample_size nparam[+]
```

Now perform the first major grid search over all the other hyperparameters and the second one overall.

```
# now tune all the remaining hyperparameters with a large space filling grid

# took roughly 2 hours
system.time({
  set.seed(2)
  bout_boost_tune_second_trans <- bout_boost_wflow_trans %>%
    tune_grid(
      resamples = burnout_folds,
      grid = bout_boost_params_trans %>%
        grid_latin_hypercube(
          size = 200),
      metrics = regr_metrics
    )
})

show_best(bout_boost_tune_second_trans, metric = "rmse")
```

From the tuning results one can conclude that in this setting actually the same hyperparameter setting as for the raw target variable seems appropriate. So one finalizes the workflow with the same hyperparameters and fits the model.

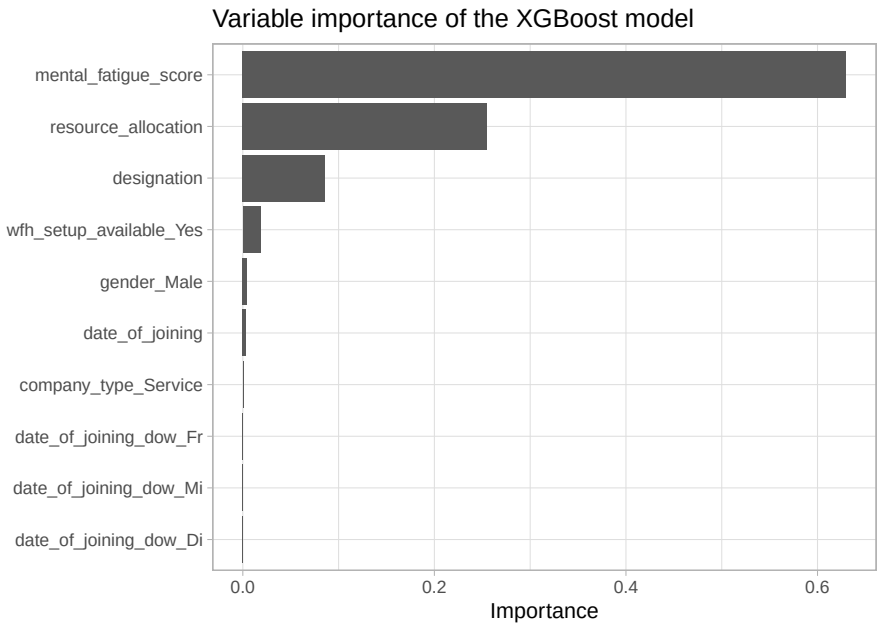
```
final_bout_boost_wflow_trans <-
  bout_boost_wflow_trans %>%
  finalize_workflow(tibble(
    mtry = 9,
    tree_depth = 4,
    learn_rate = 0.02,
    loss_reduction = 0.0000003,
    sample_size = 0.8
  )) %>%
  fit(burnout_train)
```

Now all models w.r.t. the burnout data set are fitted.

5.1.4 Evaluate and understand the model

First a visualization of the variable importance. The variable importance is basically calculated by measuring the gain w.r.t. the loss of each feature in the single trees and splits.

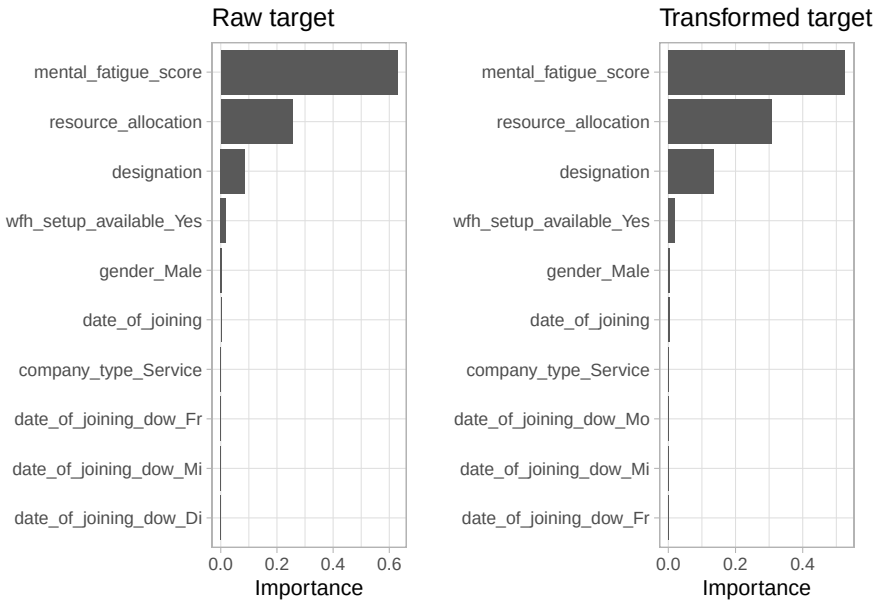
```
vip_burn <- final_bout_boost_wflow %>%
  pull_workflow_fit() %>%
  vip::vip() +
  theme_light() +
  labs(title = "Variable importance of the XGBoost model")
vip_burn
```




```
ggsave("_pictures/vip_burn.png",plot = vip_burn)
remove(vip_burn)
```

This shows that indeed the `mental_fatigue_score` is the most influential predictor by far followed by the `ressource_allocation` and `designation` features. These results are not at all surprising as the EDA exactly came to these conclusions. Especially the very few appearances of the features connected to `company_type` and `date_of_joining` are most likely just some minor overfitting.

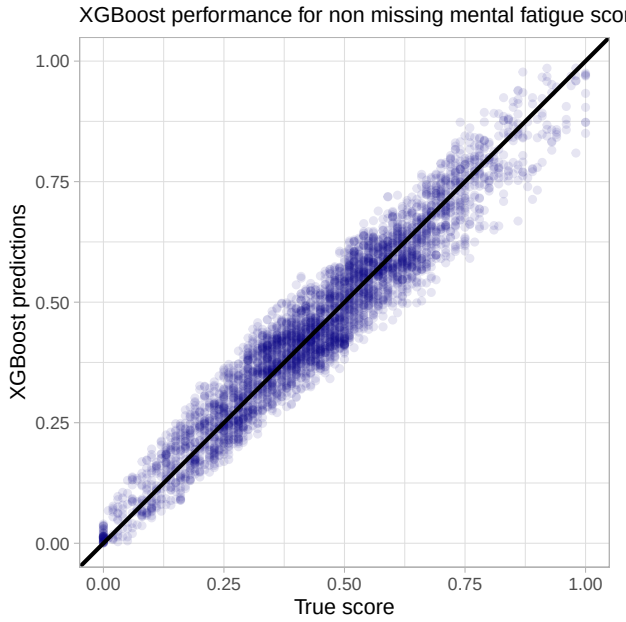
Now compare the variable importance of the model with the raw and the transformed target variable.



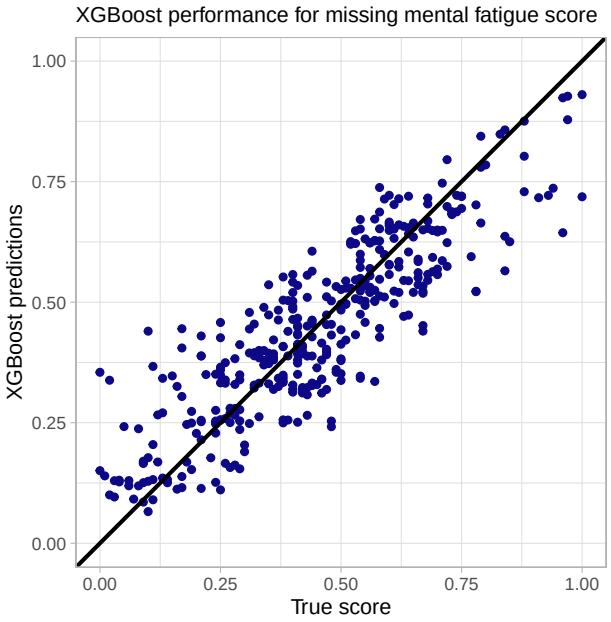
One can observe the exact same ordering. The only difference seems to be a higher influence of the designation variable for the transformed target. This is not surprising as the resource allocation and designation variable are highly correlated.

Now have a look at the performance of the main boosting model w.r.t. missing values in the most influential variable.

```
miss_vals_burn_no <- burnout_test %>%
  mutate(boost_pred = predict(final_bout_boost_wflow,
                              new_data = .)[[".pred"]]) %>%
  filter(!is.na(mental_fatigue_score)) %>%
  ggplot(aes(x = burn_rate, y = boost_pred)) +
  geom_point(col = plasma(1), alpha = 0.1, na.rm = TRUE) +
  geom_abline(slope = 1, intercept = 0, size = 1) +
  xlim(0,1) + ylim(0,1) +
  theme_light() +
  labs(y = "XGBoost predictions",
       x = "True score",
       title = "XGBoost performance for non missing mental fatigue score") +
  coord_equal() +
  theme(plot.title = ggtext::element_markdown(size = 11))
miss_vals_burn_no
```




```
miss_vals_burn_yes <- burnout_test %>%
  mutate(boost_pred = predict(final_bout_boost_wflow,
                              new_data = .)[[".pred"]]) %>%
  filter(is.na(mental_fatigue_score)) %>%
  ggplot(aes(x = burn_rate, y = boost_pred)) +
  geom_point(col = plasma(1)) +
  geom_abline(slope = 1, intercept = 0, size = 1) +
  xlim(0,1) + ylim(0,1) +
  theme_light() +
  labs(y = "XGBoost predictions",
       x = "True score",
       title = "XGBoost performance for missing mental fatigue score") +
  coord_equal() +
  theme(plot.title = ggtext::element_markdown(size = 11))
miss_vals_burn_yes
```



The performance for the missing values is indeed not that precise but still quite good. There is at least no huge outlier detectable here. While at first glance the fact that outliers are handled naturally by the model is not astonishing it really is one the core strengths of the model to deal with messy data that includes missing and sparse data. So there is no need for imputation or a second fallback model for missing values.

Now it is interesting to check which model performed the best on the test data set. The results can be viewed below in tabular and graphical form.

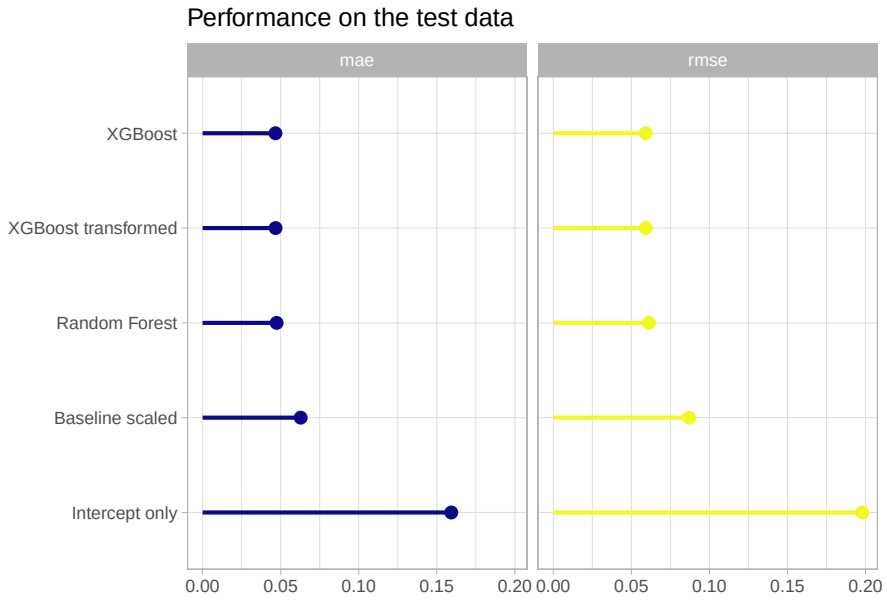
```
# calculate the mae and the rmse for all models (random forest and xgboost)
bout_test_perf <- burnout_test %>%
  mutate(rf_pred = predict(final_bout_rf_wflow,
                           new_data = .)[[".pred"]],
         boost_pred = predict(final_bout_boost_wflow,
                              new_data = .)[[".pred"]],
         boost_trans_pred = predict(final_bout_boost_wflow_trans,
                              new_data = .)[[".pred"]],
         # transform the predictions back
         boost_trans_pred = (2 / (exp(-boost_trans_pred) + 1)) - 0.5,
         intercept_pred = bout_predict_trivial_mean(.),
         mfs_scored_pred = bout_predict_trivial_mfs(.)) %>%
  mutate(across(ends_with("pred"), function(col) {
    case_when(
      col < 0 ~ 0,
      col > 1 ~ 1,
```

Table 5.1: Performance on the test data

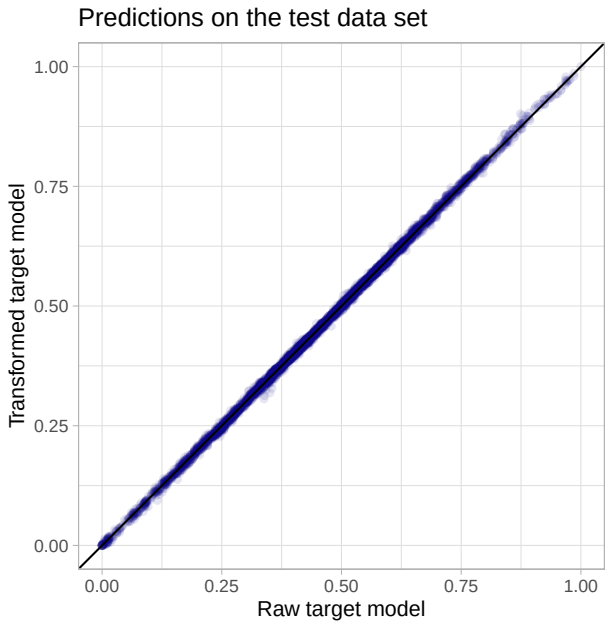
model	mae	rmse
boost_trans_pred	0.0467	0.0592
boost_pred	0.0467	0.0593
rf_pred	0.0474	0.0612
mfs_scored_pred	0.0628	0.0870
intercept_pred	0.1593	0.1980

```
      TRUE ~ col
      )
  })) %>%
  select(burn_rate, rf_pred, boost_pred, boost_trans_pred,
         intercept_pred, mfs_scored_pred) %>%
  pivot_longer(-burn_rate, names_to = "model") %>%
  group_by(model) %>%
  summarise(
    mae = mae_vec(
      truth = burn_rate,
      estimate = value
    ),
    rmse = rmse_vec(
      truth = burn_rate,
      estimate = value
    )
  ) %>%
  arrange(rmse)

knitr::kable(bout_test_perf, digits = 4,
             booktabs = TRUE,
             caption = "Performance on the test data")
```



It can be observed that already the really simple baseline model that just scales the mental fatigue score (so it reflects just a single linear influence) has a really low MAE and RMSE. Still both random forest as well as the boosting model can further improve this metric but the huge linear influence of the mental fatigue score obviously leaves not much room for improvement. By the way the simple linear model with just this one predictor has already an R^2 of more than 0.92. XGBoost manages to get a slightly better performance on the test data set but this would only be nice in a machine learning competition as in real life this difference would negligible. Actually in this use case for a real life application a very easy explainable linear model might be somewhat better than a complex model like XGBoost as the difference in performance is not too big. Nevertheless in my opinion the predictor `mental_fatigue_score` should be treated with extreme care as in a real life situation the collection of this score could be as costly or hard as the one of the outcome. There might be even latent variables that are highly linearly correlated to both scores. But this data was not intended to be used in a real life application but was shared at a machine learning competition and actually many of the best submissions there used XGBoost models. The transformation of the outcome variable did actually not change the predictive power of the model as the result for both the model with the raw target as well as the one with the transformed one performed equally good. Below one can see a scatterplot comparing the individual predictions of these two models on the test set which perfectly underlines the hypothesis that the models are basically the same as there is almost no variation around the identity slope.



So all in all for this data set it was not a way better predictive performance (if one is not in an artificial machine learning setup) that was the core strength of the tree-based gradient boosting model but the minimal pre-processing and exploratory work (for example no interactions have to be detected manually or be tested for significance) that was needed and the natural handling of missing values to achieve the model. This of course came to a quite high computational price. The famous predictive performance could probably be displayed better when having a data set with more complex non-linear patterns.

This finishes the analysis of the burnout data set. But no worries there is still one data set and boosting model left to explore namely the insurance data set.

5.2 Insurance data

5.2.1 Baseline models

The first thing is to set up the trivial **baseline models**.

```
# the trivial intercept only model:
ins_predict_trivial_mean <- function(new_data) {
  rep(mean(ins_train$log10_charges), nrow(new_data))
}

# the trivial linear model without any interactions (as we do not have
```

```
# missing values does not have to be dealt with them)
ins_baseline_lm <- lm(log10_charges ~ age + sex + bmi +
                        children + smoker + region,
                      data = ins_train %>%
                        mutate(across(where(is.character), as.factor)))
summary(ins_baseline_lm)
```

```
##
## Call:
## lm(formula = log10_charges ~ age + sex + bmi + children + smoker +
##     region, data = ins_train %>% mutate(across(where(is.character),
##     as.factor)))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.46395 -0.08466 -0.01955  0.02643  0.93557
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)      3.0685262  0.0357167  85.913  < 2e-16 ***
## age              0.0149190  0.0004316  34.566  < 2e-16 ***
## sexmale         -0.0251970  0.0119527  -2.108 0.035260 *
## bmi              0.0052551  0.0010243   5.130 3.44e-07 ***
## children         0.0413979  0.0050235   8.241 5.00e-16 ***
## smokeryes        0.6841462  0.0147963  46.238  < 2e-16 ***
## regionnorthwest -0.0178125  0.0172137  -1.035 0.301003
## regionsoutheast -0.0613744  0.0172229  -3.564 0.000382 ***
## regionsouthwest -0.0512729  0.0172868  -2.966 0.003084 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1945 on 1061 degrees of freedom
## Multiple R-squared:  0.764,  Adjusted R-squared:  0.7622
## F-statistic: 429.4 on 8 and 1061 DF,  p-value: < 2.2e-16
```

The predictions of these baseline models on the test data set will be compared with the predictions of the tree-based models that will be constructed.

5.2.2 Model specification

```
# Models:
```

```
# Random forest model for comparison
ins_rf_model <- rand_forest(trees = tune(),
                            mtry = tune()) %>%
  set_engine("ranger") %>%
  set_mode("regression")

# XGBoost model
ins_boost_model <- boost_tree(trees = tune(),
                              learn_rate = tune(),
                              loss_reduction = tune(),
                              tree_depth = tune(),
                              mtry = tune(),
                              sample_size = tune(),
                              stop_iter = 10) %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

```
# Workflows: model + recipe

ins_rf_wflow <-
  workflow() %>%
  add_model(ins_rf_model) %>%
  add_recipe(ins_rec_rf)

ins_boost_wflow <-
  workflow() %>%
  add_model(ins_boost_model) %>%
  add_recipe(ins_rec_boost)
```

5.2.3 Tuning

For the hyperparameter tuning one needs validation sets to monitor the models on unseen data. To do this 5-fold cross validation (CV) is used here.

```
# Create Resampling object
set.seed(2)
ins_folds <- vfold_cv(ins_train, v = 5)
```

Now the parameter objects will be fixed for the grid searches analogously to above.

```
ins_rf_params <- ins_rf_wflow %>%
  parameters() %>%
```

```
  update(trees = trees(c(100, 2000))) %>%
  finalize(ins_train)
ins_rf_params %>% pull_dials_object("trees")
```

```
## # Trees (quantitative)
## Range: [100, 2000]
```

```
ins_rf_params %>% pull_dials_object("mtry")
```

```
## # Randomly Selected Predictors (quantitative)
## Range: [1, 8]
```

```
ins_boost_params <- ins_boost_wflow %>%
  parameters() %>%
  update(trees = trees(c(100, 2000))) %>%
  finalize(ins_train)
ins_boost_params
```

```
## Collection of 6 parameters for tuning
##
##      identifier           type    object
##            mtry           mtry nparam[+]
##           trees          trees nparam[+]
##      tree_depth     tree_depth nparam[+]
##      learn_rate     learn_rate nparam[+]
##  loss_reduction loss_reduction nparam[+]
##     sample_size    sample_size nparam[+]
```

The **random forest model** goes first with the tuning.

Here there are as seen above only two hyperparameters to be tuned thus 30 combinations should suffice.

```
# took roughly 2 minutes
system.time({
  set.seed(2)
  ins_rf_tune <- ins_rf_wflow %>%
    tune_grid(
      resamples =  ins_folds,
      grid = ins_rf_params %>%
        grid_latin_hypercube(size = 30, original = FALSE),
      metrics = regr_metrics
    )
```

```
})

# visualization of the tuning results (snapshot of the output below)
autoplot(ins_rf_tune) + theme_light()
# this functions shows the best combinations wrt the rmse metric of all the
# combinations in the grid
show_best(ins_rf_tune, metric = "rmse")
```

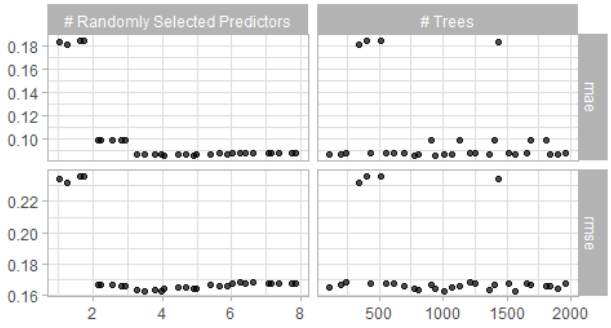


Figure 5.5: Result of a spacefilling grid search for the random forest model.

The visualization alongside the best performing results suggest that a value of `mtry`of 4 and 1000 `trees` should give good results. Thus one can finalize and fit this model.

```
final_ins_rf_wflow <-
  ins_rf_wflow %>%
  finalize_workflow(tibble(
    trees = 1000,
    mtry = 4
  )) %>%
  fit(ins_train)
```

Now the **boosting model**.

Again one starts to tune over the number of trees first (this time we also use three different tree depths to visualize the impact of this parameter too)

```
first_grid_boost_ins <- crossing(
  trees = seq(250, 2000, 250),
  mtry = 8,
  tree_depth = c(1, 4, 6),
  loss_reduction = 0.000001,
```

```
  learn_rate = 0.01,
  sample_size = 1
)
```

```
# took roughly half a minute
system.time({
  set.seed(2)
  ins_boost_tune_first <- ins_boost_wflow %>%
    tune_grid(
      resamples =  ins_folds,
      grid = first_grid_boost_ins,
      metrics = regr_metrics
    )
})


show_best(ins_boost_tune_first)
```

```
# visualize the tuning results (output in the figure below)
ins_boost_tune_first %>%
  collect_metrics() %>%
  ggplot(aes(x = trees, y = mean, col = factor(tree_depth))) +
  geom_point() +
  geom_line() +
  geom_linerange(aes(ymin = mean - std_err, ymax = mean + std_err)) +
  labs(col = "Max tree depth", y = "", x = "Number of trees",
       title = "") +
  scale_color_viridis_d(option = "C") +
  facet_wrap(~ .metric) +
  theme_light()
```

This visualization clearly shows that one has to have a closer look at the region around 500 trees. More than 500 trees might lead to overfitting as seen here.

```
second_grid_boost_ins <- crossing(
  trees = seq(250, 750, 50),
  mtry = 8,
  tree_depth = c(3, 4, 5, 6),
  loss_reduction = 0.000001,
  learn_rate = 0.01,
  sample_size = 1
)
```

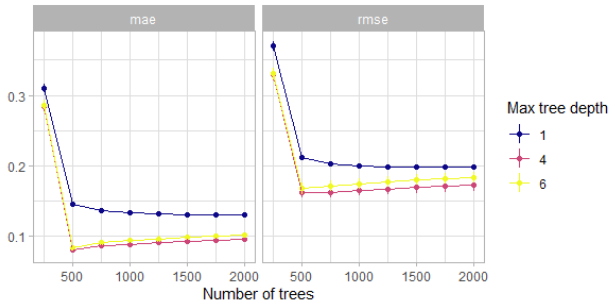


Figure 5.6: Comparison of the initial grid search for the number of trees w.r.t. the maximum tree depth. Actually with tiny confidence bands around the estimates.

```
# took roughly 15 seconds
system.time({
  set.seed(2)
  ins_boost_tune_second <- ins_boost_wflow %>%
    tune_grid(
      resamples =  ins_folds,
      grid = second_grid_boost_ins,
      metrics = regr_metrics
    )
})


show_best(ins_boost_tune_second)
```

```
# visualize the tuning results (output in the figure below)
ins_boost_tune_second %>%
  collect_metrics() %>%
  ggplot(aes(x = trees, y = mean, col = factor(tree_depth))) +
  geom_point() +
  geom_line() +
  geom_linerange(aes(ymin = mean - std_err, ymax = mean + std_err)) +
  labs(col = "Max tree depth", y = "", x = "Number of trees",
       title = "") +
  scale_color_viridis_d(option = "C") +
  facet_wrap(~ .metric) +
  theme_light()
```

So from this visualization one can conclude that a number of trees of 600 should suffice to get a decent model. So the workflow will be updated and the number of

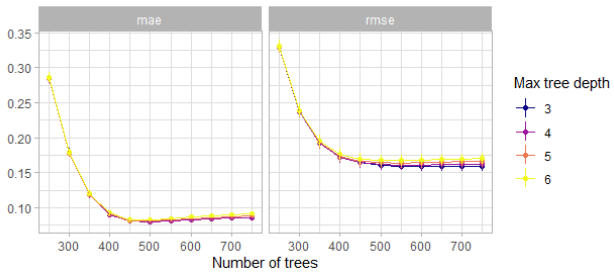


Figure 5.7: Comparison of the second grid search for the number of trees w.r.t. the maximum tree depth. Actually with tiny confidence bands around the estimates.

trees fixed. After that the first major space filling grid search will be performed. Moreover with the previous visualizations in mind one can reduce the bound of the parameter space of the maximum tree depth quite a bit from 15 to 9. A minimum tree depth of 2 seems also appropriate. Before one does that mainly for a view on the influence of the learning rate one can tune one round only with the learning rate and some tree numbers for comparison.

```
# tune mainly learn rate
lrate_grid_boost_ins <- crossing(
  trees = seq(200, 1500, 100),
  mtry = 8,
  tree_depth = c(3),
  loss_reduction = c(0.000001),
  learn_rate = c(0.1, 0.01, 0.001, 0.0001),
  sample_size = 1
)
```

```
# took roughly 20 seconds
system.time({
  set.seed(2)
  ins_boost_tune_lrate <- ins_boost_wflow %>%
    tune_grid(
      resamples =  ins_folds,
      grid = lrate_grid_boost_ins,
      metrics = regr_metrics
    )
})


show_best(ins_boost_tune_lrate)
```

```
# visualize the tuning results (output in the figure below)
ins_boost_tune_lrate %>%
  collect_metrics() %>%
  ggplot(aes(x = trees, y = mean, col = factor(learn_rate))) +
  geom_point() +
  geom_line() +
  geom_linerange(aes(ymin = mean - std_err, ymax = mean + std_err)) +
  labs(x = "Number of trees", y = "", col = "Learning rate",
       title = "") +
  scale_color_viridis_d(option = "C") +
  facet_wrap(~ .metric) +
  theme_light()
```

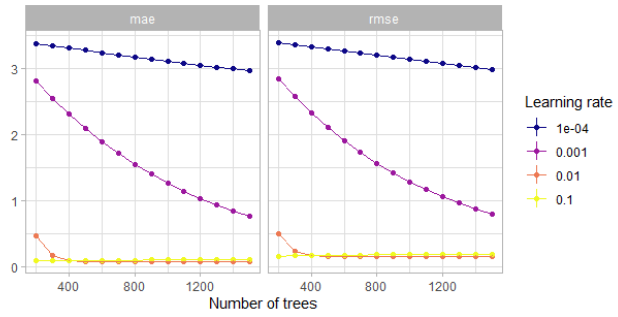


Figure 5.8: Comparison of the grid search for the number of trees w.r.t. the learning rate. Actually with tiny confidence bands around the estimates.

This showcases the fact that indeed a learning rate that is too small can blow up the computational costs.

Now fix the number of the `trees` hyperparameter.

```
# fix the number of trees by redefining the boostin model with a
# fixed number of trees.
ins_boost_model <- boost_tree(trees = 600,
                              learn_rate = tune(),
                              loss_reduction = tune(),
                              tree_depth = tune(),
                              mtry = tune(),
                              sample_size = tune(),
                              stop_iter = 10) %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

```
# update the workflow
ins_boost_wflow <-
  ins_boost_wflow %>%
  update_model(ins_boost_model)

ins_boost_params <- ins_boost_wflow %>%
  parameters() %>%
  update(tree_depth = tree_depth(c(2, 9))) %>%
  finalize(ins_train)

# reduced hyperparameter space
ins_boost_params
```

```
## Collection of 5 parameters for tuning
##
##      identifier           type    object
##            mtry           mtry nparam[+]
##      tree_depth     tree_depth nparam[+]
##      learn_rate     learn_rate nparam[+]
##  loss_reduction loss_reduction nparam[+]
##     sample_size    sample_size nparam[+]
```

```
# now tune all the remaining hyperparameters with a large space filling grid

# took roughly 10 minutes
system.time({
  set.seed(2)
  ins_boost_tune_major1 <- ins_boost_wflow %>%
    tune_grid(
      resamples = ins_folds,
      grid = ins_boost_params %>%
        grid_latin_hypercube(
          size = 300),
      metrics = regr_metrics
    )
})

show_best(ins_boost_tune_major1, metric = "rmse")
```

The most prominent suggestions of the visualizations and the table of the best performing ones are combinations of hyperparameters with a very low `loss_reduction` , a low `learn_rate`, a medium to high value of `mtry`, a not too big `tree_depth` as well as a rather high value for the `sample_size`. These observations will be used to refine the search space and perform once more a space filling grid search.

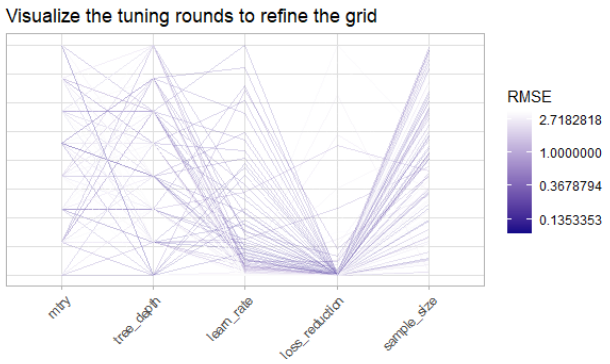


Figure 5.9: Visualize the tuning results with a parallel coordinate plot. The y axis represents the scaled range for each of the hyperparameter spaces.

```
# now tune all the remaining hyperparameters with a refined space filling grid

# took roughly 15 minutes
system.time({
  set.seed(2)
  ins_boost_tune_major2 <- ins_boost_wflow %>%
    tune_grid(
      resamples = ins_folds,
      grid = ins_boost_params %>%
        update(
          mtry = mtry(c(6,8)),
          tree_depth = tree_depth(c(3,4)),
          learn_rate = learn_rate(c(-1.7, -1.5)),
          loss_reduction = loss_reduction(c(-4, -1.5)),
          sample_size = sample_prop(c(0.3, 0.9))
        ) %>%
        grid_latin_hypercube(
          size = 300),
      metrics = regr_metrics
    )
})

show_best(ins_boost_tune_major2, metric = "rmse")
```

The results from this grid search were quite clear. One will further use a `mtry` value of 7, a `tree_depth` of 3, a `learn_rate` of 0.02, a `loss_reduction` of 0.03 and a `sample_size` of 0.8.

```
final_ins_boost_wflow <-
  ins_boost_wflow %>%
  finalize_workflow(tibble(
    mtry = 7,
    tree_depth = 3,
    learn_rate = 0.02,
    loss_reduction = 0.03,
    sample_size = 0.8
  )) %>%
  fit(ins_train)
```

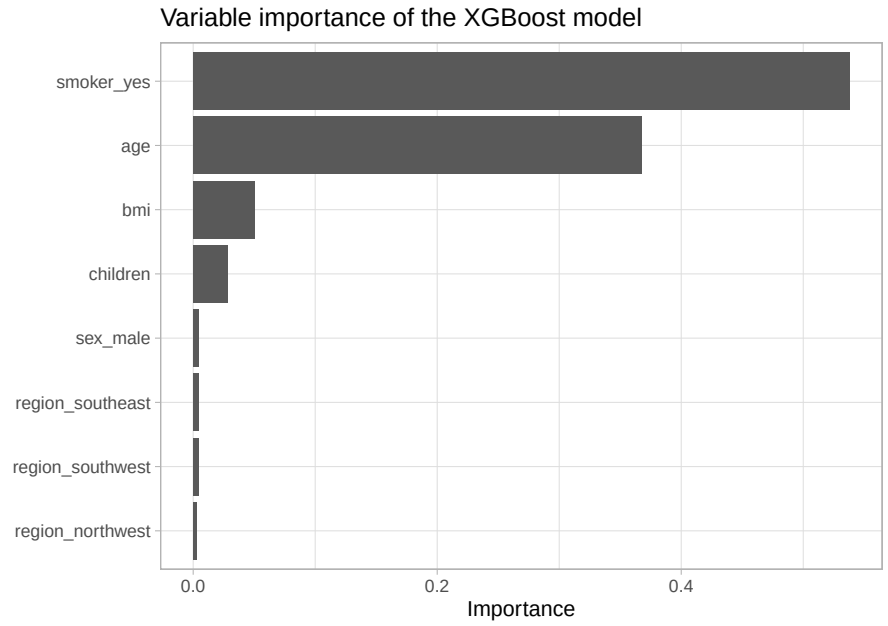
From here no computational intensive task will be performed so one stops the cluster.

```
# stop cluster
stopCluster(cl)
```

5.2.4 Evaluate and understand the model

First a visualization of the variable importance. The variable importance is again calculated by measuring the gain w.r.t. the loss of each feature in the single trees and splits.

```
vip_ins <- final_ins_boost_wflow %>%
  pull_workflow_fit() %>%
  vip::vip() +
  theme_light() +
  labs(title = "Variable importance of the XGBoost model")
vip_ins
```




```
ggsave("_pictures/vip_ins.png", plot = vip_ins)
remove(vip_ins)
```

Here one can clearly see that the trends that were observable during the EDA are reflected here again. The most influential variable is indeed the smoker variable, closely followed by the age variable. Also the bmi and the number of children seem to be relevant. To get an even better sense for where which variable was mostly used one can have a look at the visualization below which shows the final model in a compressed form as a single tree. At each node the most important (again by the feature importance score like above) 5 variables are listed with their respective raw importance scores in brackets. The 'Leaf' variable just corresponds to the case where trees ended at this point.

```
# Visualization of the ensemble of trees as a single collective unit
final_ins_boost_fit <- final_ins_boost_wflow %>%
  pull_workflow_fit()
final_ins_boost_fit$fit %>%
  xgb.plot.multi.trees(render = T)
# output as figure below as this returns a htmlwidget which is not that
# intuitively handled by latex (pdf output)
```

It is quite interesting to see that most of the time the smoker feature was used just once in the first node but not multiple times. This suggests a quite linear influence of smokers and of course the binary character of the feature can support

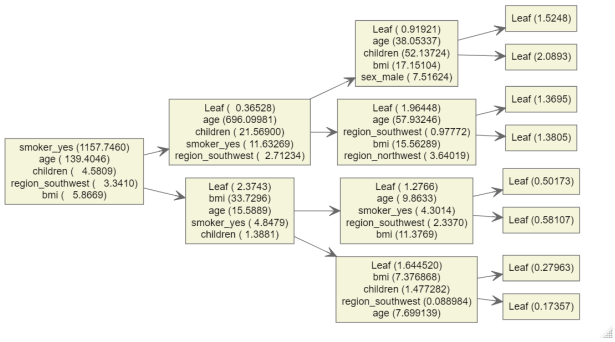


Figure 5.10: Visualization of the final ensemble of trees as a single collective unit.

this behavior. Age on the other hand is represented in multiple depths with high importance which suggests a non-linear influence which was also visible during the EDA. Interestingly the children variable for example mostly is used at depth 3 for a split.

Also a residual plot of the XGBoost predictions on the test data set can shed some light on the way the model works. This 3D plot is an interactive html widget which works not in the pdf output but just on the website. Besides the residuals on the test set also the residuals on the training data set are shown below

```
# prepare the data for plotting i.e. calculate raw residuals on the test
# and the training data
resid_plot_ins_data_test <- ins_test %>%
  bind_cols(pred = predict(final_ins_boost_wflow, new_data = ins_test)[[".pred"]]) %>%
  mutate(resid = charges - 10^pred)
resid_plot_ins_data_train <- ins_train %>%
  bind_cols(pred = predict(final_ins_boost_wflow, new_data = ins_train)[[".pred"]]) %>%
  mutate(resid = charges - 10^pred)

save(resid_plot_ins_data_test,
     resid_plot_ins_data_train,
     file = "_data/resid_data_ins.RData")

# test residuals
resid_plot_ins_data_test %>%
  plotly::plot_ly(x = ~age, z = ~resid, y = ~bmi,
                  color = ~factor(smoker),
                  symbol = ~factor(smoker),
                  text = ~paste("Age:", age, "BMI:", bmi),
                  hoverinfo = "text",
```

```
                  symbols = c('x','circle'),
                  colors = plasma(3)[2:1]) %>%
  plotly::add_markers(opacity = 0.9,
                      size = 2) %>%
  plotly::layout(scene = list(
    xaxis = list(title = "Age"),
    yaxis = list(title = "BMI"),
    zaxis = list(title = "Raw residuals")),
    title = "Raw test residuals of the insurance XGBoost model",
    legend = list(title = list(text = "Smoker")))
```

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is
```

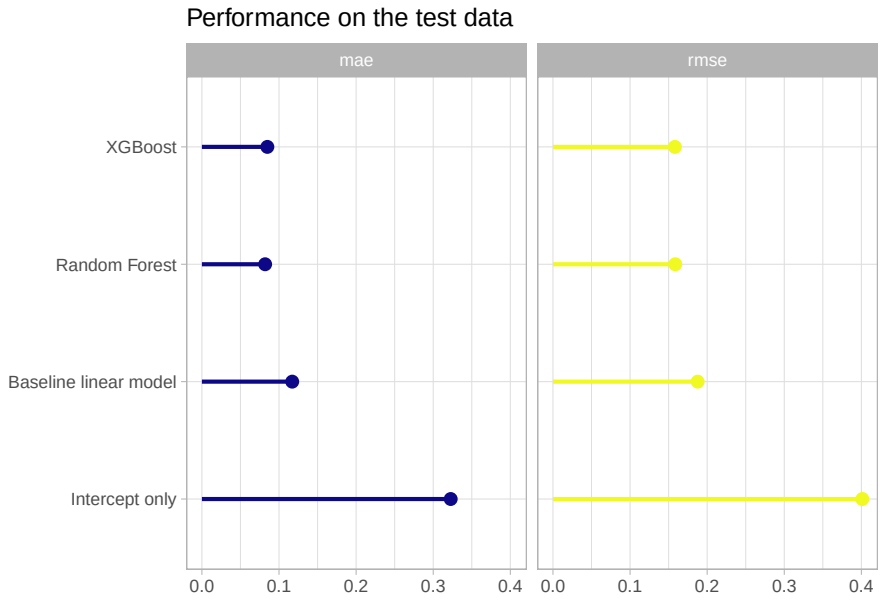
```
# train residuals
resid_plot_ins_data_train %>%
  plotly::plot_ly(x = ~age, z = ~resid, y = ~bmi,
                  color = ~factor(smoker),
                  symbol = ~factor(smoker),
                  text = ~paste("Age:", age, "BMI:", bmi),
                  hoverinfo = "text",
                  symbols = c('x','circle'),
                  colors = plasma(3)[2:1]) %>%
  plotly::add_markers(opacity = 0.9,
                      size = 2) %>%
  plotly::layout(scene = list(
    xaxis = list(title = "Age"),
    yaxis = list(title = "BMI"),
    zaxis = list(title = "Raw residuals")),
    title = "Raw train residuals of the insurance XGBoost model",
    legend = list(title = list(text = "Smoker")))
```

For the train residuals no clear pattern in the residuals is visible. Looking at the test residuals one can see where the model has the most difficulties. Interesting enough the model has the biggest outliers with non smokers of either quite low or quite high age with a more or less not that extreme BMI. This could be due to chronic diseases especially for the younger ones but also some randomness that life holds. For example a car accident could cause such high charges. Of course it could be also due to not observed latent variables.

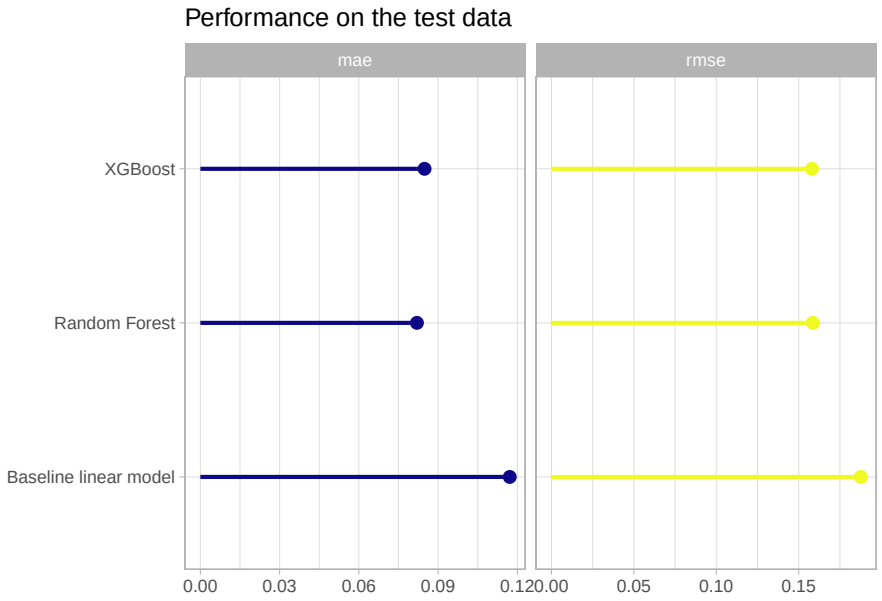
Which model performed the best on the test data set is the next interesting question. The results can be viewed below in tabular and graphical form.

Table 5.2: Performance on the test data

model	mae	rmse
boost_pred	0.0849	0.1581
rf_pred	0.0820	0.1586
baseline_lm	0.1171	0.1877
intercept_pred	0.3227	0.4012



Once without the intercept only model.



So overall the XGBoost model again delivered the best performance on the test data set w.r.t. the main optimization metric RMSE. The baseline linear model that takes all features without interactions into account also goes a long way and the performance w.r.t. the RMSE metric is even comparable to the tree-based models. So again the data did not leave much room to explore more complex non-linear patterns. Whether the rather small difference in performance is worth considering such a model is of course up to the use case. Again the XGBoost model displayed that it reaches a very competitive performance with minimal pre-processing, integrated feature selection and in this case the computational effort was quite small. The major pros and cons of the tree-based gradient boosting models will be shortly reviewed in the next and final section.

Chapter 6

Conclusion

Now after having a solid theoretical understanding of gradient boosting methods for regression and the implementation XGBoost as well as seeing the application of such models to two data sets one can wrap up this project by discussing the major pros and cons of this approach with this specific implementation in mind.

6.1 Pros

- Minimal pre-processing
- Flexible enough to detect complex non-linear patterns
- Handling of missing values
- Integrated feature selection
- Good generalization due to lots of regularization options
- Strong predictive power

6.2 Cons

- Not as explainable as for example a linear model
- Computationally demanding (especially the hyperparameter tuning)
- Preferably lots of observations

These pros and cons show that tree-based gradient boosting is a very powerful learning algorithm but still not suitable to any application. Due to its very robust nature and good handling of missing values and features of different scales it is one of the dominant algorithms in data mining alongside models like random forest. But in critical applications with just few observations such models should be treated with care as they are not that explainable. Still if one handles tabular data in order to perform a regression or classification task one should take the pros and cons that were mentioned above into account and in many cases a consideration of a tree-based gradient boosting model like XGBoost is advisable.

Thanks for sticking with it!

Chapter 7

References

Bibliography

Boehmke, B. and Greenwell, B. (2019). *Hands-On Machine Learning with R*.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754.

Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., Mitchell, R., Cano, I., Zhou, T., Li, M., Xie, J., Lin, M., Geng, Y., and Li, Y. (2021). *xgboost: Extreme Gradient Boosting*. R package version 1.3.2.1.

Corporation, M. and Weston, S. (2019). *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.15.

Garnier, S. (2018a). *viridis: Default Color Maps from 'matplotlib'*. R package version 0.5.1.

Garnier, S. (2018b). *viridisLite: Default Color Maps from 'matplotlib' (Lite Version)*. R package version 0.3.0.

Greenwell, B. M. and Boehmke, B. C. (2020). Variable importance plots—an introduction to the vip package. *The R Journal*, 12(1):343–366.

Grolemund, G. and Wickham, H. (2011). Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3):1–25.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning (12th printing)*. Springer New York.

Kuhn, M. and Wickham, H. (2020). *Tidymodels: a collection of packages for modeling and machine learning using tidyverse principles.*

Meschiari, S. (2021). *latex2exp: Use LaTeX Expressions in Plots*. R package version 0.5.0.

Pedersen, T. L. (2020). *patchwork: The Composer of Plots*. R package version 1.1.1.

Schloerke, B., Cook, D., Larmarange, J., Briatte, F., Marbach, M., Thoen, E., Elberg, A., and Crowley, J. (2021). *GGally: Extension to 'ggplot2'*. R package version 2.1.1.

Sievert, C. (2020). *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC.

Tierney, N. (2017). visdat: Visualising whole data frames. *JOSS*, 2(16):355.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wilke, C. O. (2020). *ggtext: Improved Text Rendering Support for 'ggplot2'*. R package version 0.1.1.

Wolf, M. M. (2020). Lecture notes in mathematical foundations of supervised learning.

Wright, M. N. and Ziegler, A. (2017). ranger: A fast implementation of random forests for high dimensional data in C++ and R. *Journal of Statistical Software*, 77(1):1–17.