Machine Learning 2: Prediction of the Individual Medical Cost

Conor Fallon; Dennis Fast; Leonhard Liu; Tassilo Henninger

17 1 2023

Table of contents

1 Project Goal and Outline

The task at hand is a regression problem. Broadly speaking, we wish to create a series of models, each of which will be able to make predictions on a withheld test set. The dataset in question is a Medical Cost dataset. The variable we wish to predict is the expected insurance premium for a given individual based on the following input variables:

- age
- sex: male or female
- Body mass index, is a measure of one's weight relative to their height
- children: Number of children/dependants also covered under this person's health insurance
- smoker: Whether the person smokes or not
- region: Four locations: northeast, southeast, southwest, and northwest. All refer to the United States
- charges: The premium billed by the health insurance company. This is the predictor variable

Of course, some of these variables seem obvious in how they affect a person's insurance premium; a 98 year-old obese smoker will have a higher insurance premium than a 24 year-old non-smoker with a healthy BMI. However, through some exploratory data analysis as well as by comparing our methods, a more discerning picture of how each variable impacts the value we wish to predict will be discussed.

There will be three models compared to each other: a baseline model, which consists of a simple linear regression; a regression tree; and a neural network. The best model is ultimately decided upon by using the Root Mean Squared error, although other metrics, where suitable, will be used to influence our decision making. For both the Neural Network and the Regression Tree models, we wish to find a 'best' Neural Network and a 'best' Regression Tree with which to work. For this purpose, these models are initially trained only on the train split (as outlined further on), and the 'best' model for each is decided upon based on their performance on a validation split. As such, the selection of the best model in our two algorithm classes is based on the validation error.

It is important to emphasise that this leaves the test data untouched at the stage when these models are created, leaving the test data purely for the final model comparison between the best Neural Net and the best Regression Tree. Then when ultimately comparing the two chosen models, these are retrained on the combined train and validation splits, and finally tested on the test set, which has been withheld for this purpose.

This report will conclude with a discussion of our findings, an examination of what worked and what did not work, and will also mention avenues for future work.

2 Dataset and Preprocessing

The dataset we used is included in a Book called "Machine Learning with R" by Brett Lantz and is also available on Kaggle. It can be downloaded via the following links:

- https://github.com/stedy/Machine-Learning-with-R-datasets
- https://www.kaggle.com/datasets/mirichoi0218/insurance

It contains 1338 rows, 7 columns and there are no missing values. The 6 predictor variables are like in the project outline introduced: age(numeric, integer), sex(female/male), BMI(numeric, float), children(numeric, integer), smoker(boolean) and region(categorical). The outcome variable is "charges" (numeric, float).

The only preprocessing we had to do was encoding the categorical features as factor variables. Next we split our dataset into a 60% train set, a 20% validation and a 20% test set, as it is asked in the project description.

3 EDA

One of the objectives of exploratory data analysis is to get a feel for the data you are dealing with by describing the key features of the data and summarizing the results.

The first thing we can do is to take a look at the descriptive statistics via the summary function and the histograms.

- we have individuals ranging from the age of 18 to 64 with a mean and median around 39 ages.
- the sex is almost perfectly balanced as 49.5% of the individuals are female and 50.5% are male.
- the BMI is ranging from 15.96 to 53.13 with a mean and median around 30. The BMI is calculated by the following formula:

$$BMI = \frac{mass_{kg}}{height_{m}^{\ 2}}$$

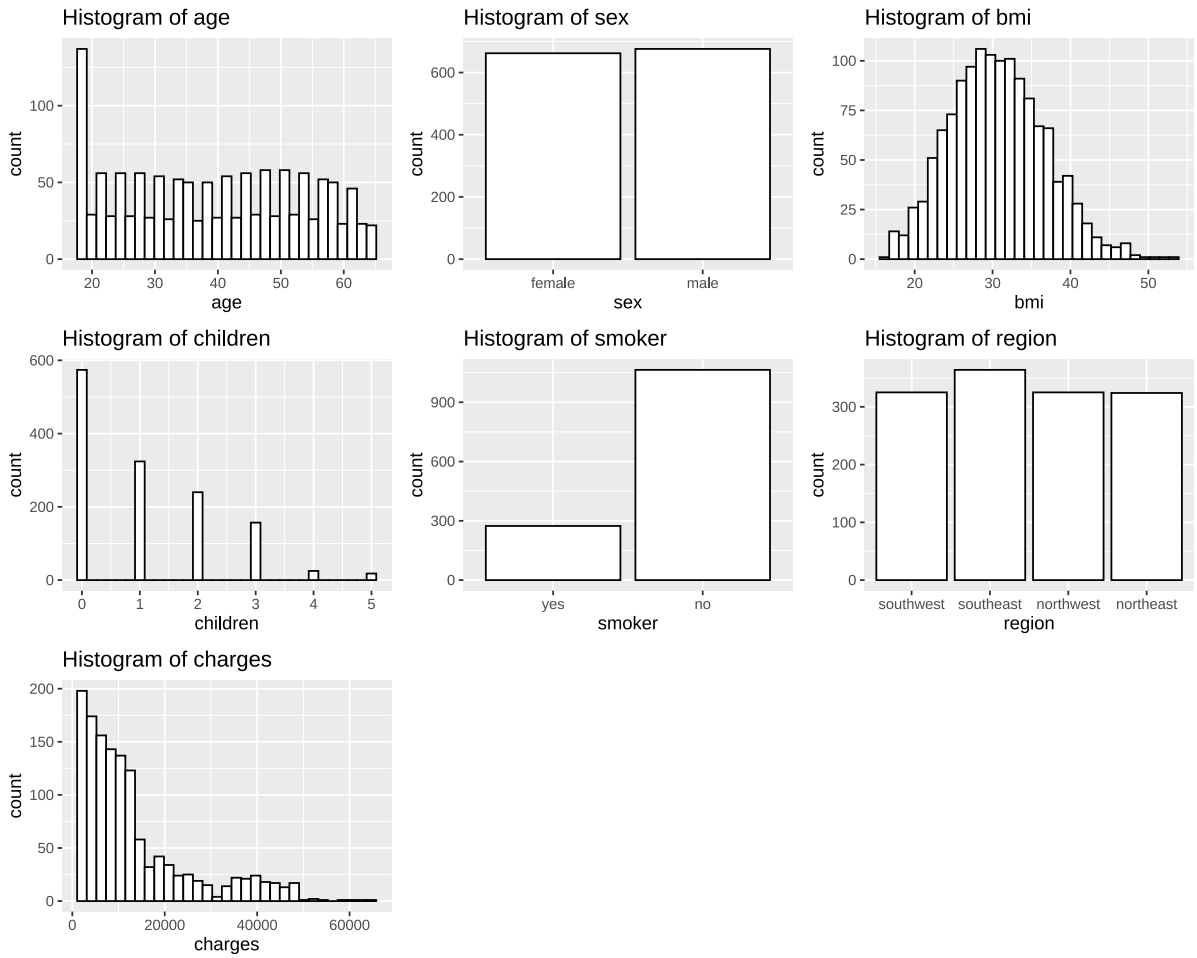
A common use of the BMI is to assess how far an individual's body weight departs from what is normal for a person's height. The WHO regards an adult BMI of less than 18.5 as underweight, a BMI of 25 or more as overweight and 30 or more is considered obese. From the summary and the histogram, we can clearly see, that our population of individuals is in mean considered obese and therefore an unhealthy population.

Most of the individuals have no children, the mean is 1 child and the maximum is 5. Regarding the factor feature smoker, the dataset is unbalanced. Only 20.5% of our individuals are smokers, respectively 79.5% are non smokers. The region is well balanced again between the four categories "southwest", "southeast", "northwest" and "northeast".

Last we have our predictor variable charges. The minimum of a health insurance charge is 1122$, the median is 9382$, the mean 13270$ and the maximum 63770$. From the difference of median and mean we can already tell that the distribution is skewed to the right (positively skewed). This means that there are a few large values that are pulling the mean higher, but the majority of the data is concentrated around the median. Looking at the boxplot for the charge we can confirm that. We can inspect a long tail of high values and a shorter tail of low values. We have no symmetric distribution.

```
summary(df.insurance)
```

```
      age            sex            bmi           children     smoker
 Min.   :18.00   female:662   Min.   :15.96   Min.   :0.000   yes: 274
 1st Qu.:27.00   male  :676   1st Qu.:26.30   1st Qu.:0.000   no :1064
 Median :39.00                Median :30.40   Median :1.000
 Mean   :39.21                Mean   :30.66   Mean   :1.095
 3rd Qu.:51.00                3rd Qu.:34.69   3rd Qu.:2.000
 Max.   :64.00                Max.   :53.13   Max.   :5.000
       region       charges
 southwest:325   Min.   : 1122
 southeast:364   1st Qu.: 4740
 northwest:325   Median : 9382
 northeast:324   Mean   :13270
                 3rd Qu.:16640
                 Max.   :63770
```



Next, we want to better understand the relationships between our different variables in the dataset. Therefore we use the correlation matrix to get insights into our data regarding the linearity and their respective strengths. A positive correlation means that as one variable increases, the other variable also increases, and a negative correlation means that as one variable increases, the other variable decreases. A correlation value of 1 or -1 indicates a perfect linear relationship between the variables. A correlation value of 0 indicates that there is no linear relationship between the variables.

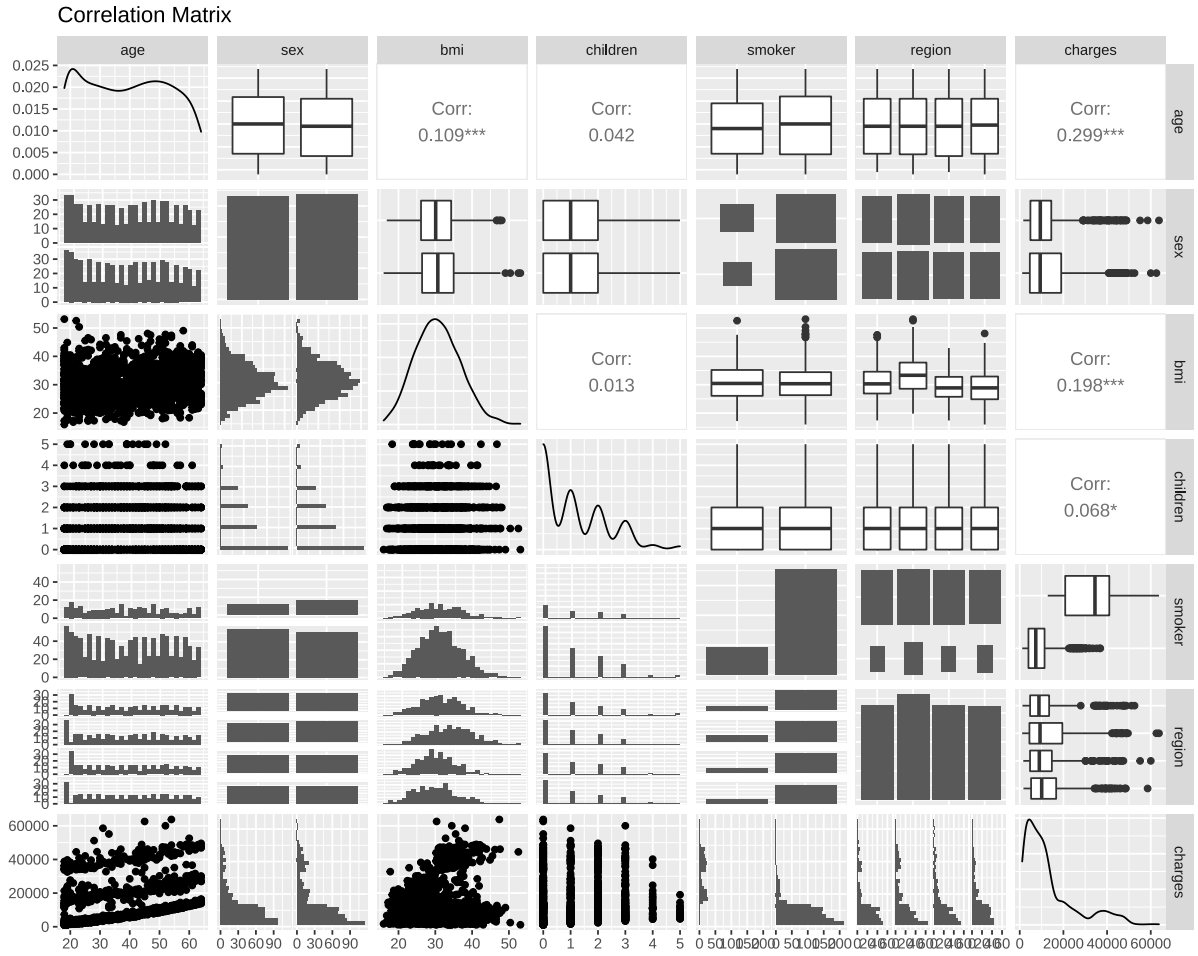
Note however, that correlation does not imply causality, it just indicates that two variables are related. That means the correlation between two variables may be high, but that the correlation is due to a relationship between the two variables and a third variable. This concept is called multicollinearity.

This has to be kept in mind for our baseline model. If we create a linear model and want to interpret the coefficients on how they influence our charge, multicollinearity can be a problem, as interpretability assumes that you can only change the value of one explanatory variable and not the others at the same time. This of course is only true if there are no correlations

between the explanatory variables. If this independence does not hold, we have a problem of multicollinearity.

Multicollinearity can result in the coefficients swinging wildly based on which other independent variables are in the model. Therefore the coefficients become very sensitive to small changes in the model and can not be easily interpreted. One way to detect multicollinearity is to examine the correlation matrix and look for high correlation coefficients between the multiple explanatory variables.

As the correlation metric can only be calculated for numeric values, we get the correlation scores for the features age, bmi, children and charges. Charges has the strongest correlation with Age (+0.299), then bmi (+0.198) and last children with only +0.068. The highest correlation between explanatory variables is +0.109 between bmi and age. As this is quite low, we don't have a problem of multicollinearity between the numeric explanatory variables.



One way to assess the influence of categorical variables on an outcome variable is through a one-way ANOVA, which compares the means of a numerical outcome variable across different

levels of a categorical variable. Thereby we could determine if there is a significant difference in the means of the outcome variable between the different levels of the categorical variable.

However, the ANOVA assumes that the data is normally distributed, independent and that the variances of the groups are equal. We already found that outcome "charges" is positively skewed and not normally distributed. Thereby, the assumption of the ANOVA fails. We can however use the non-parametric alternatives called Kruskal-Wallis test.

The output of Kruskal-Wallis test statistics gives us the p-value, through which we can determine the significance of the test. A small p-value (typically less than 0.05) indicates strong evidence against the null hypothesis, which in this case is that there is no difference in the medians of the outcome variable between the different levels of the categorical variable.

As only the p-value of the categorical variable "smoker" is below the significance level of 5%, we can only reject the null hypothesis for smoker and therefore say, there is a significant difference in the medians of the outcome variable between being a smoker and not. We can confirm the result also by looking at the boxplot, split up by category. This is also visible in small boxplots in the correlation matrix.

```
kruskal.test(df.insurance$charge , df.insurance$sex)
```

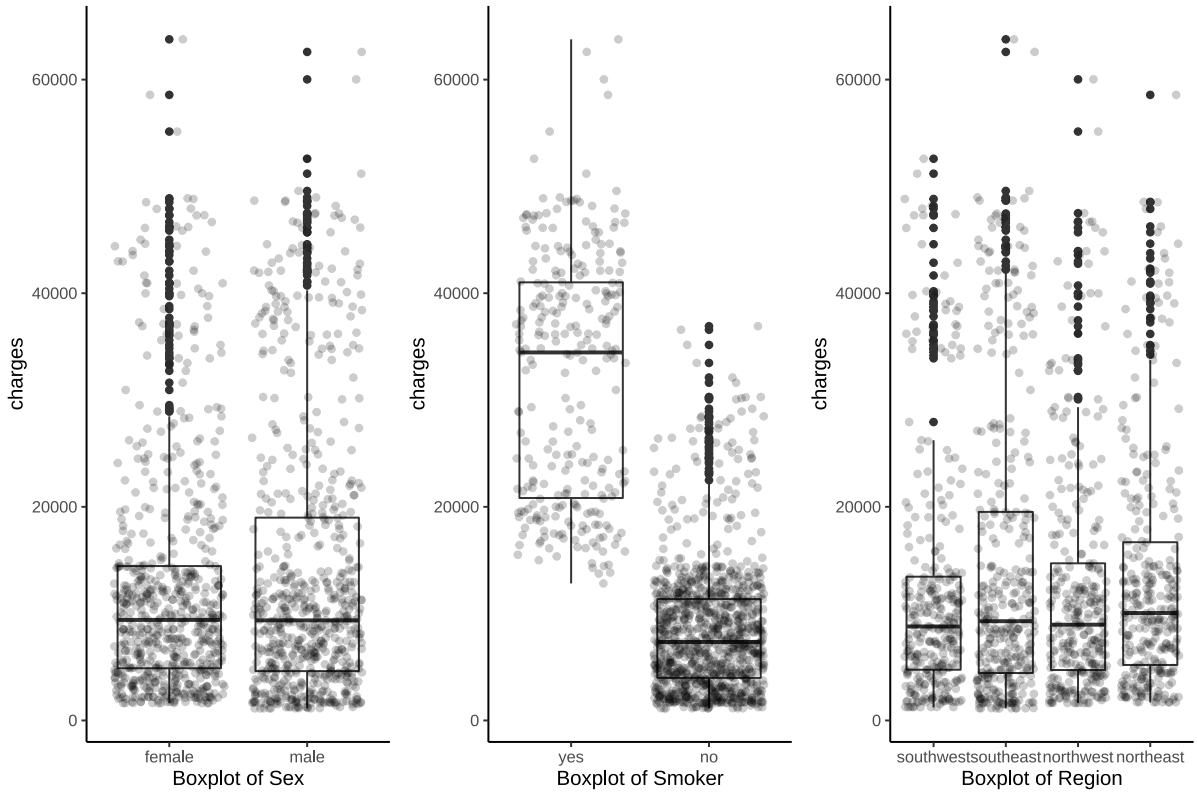
```
    Kruskal-Wallis rank sum test

data:  df.insurance$charge and df.insurance$sex
Kruskal-Wallis chi-squared = 0.1204, df = 1, p-value = 0.7286
```

```
kruskal.test(df.insurance$charge , df.insurance$smoker)
```

```
    Kruskal-Wallis rank sum test

data:  df.insurance$charge and df.insurance$smoker
Kruskal-Wallis chi-squared = 588.52, df = 1, p-value < 2.2e-16
```

```
kruskal.test(df.insurance$charge , df.insurance$region)
```

```
    Kruskal-Wallis rank sum test

data:  df.insurance$charge and df.insurance$region
Kruskal-Wallis chi-squared = 4.7342, df = 3, p-value = 0.1923
```



4 Forecasting

4.1 Baseline - Linear Regression

In order to check the performance of the chosen models, we first created a simple linear regression model as a baseline model, with 'charges' as the outcome variable and all other columns ('age', 'sex', 'BMI', 'children', 'smoker', and 'region') as predictor variables.

4.1.1 Mathematical Overview

The mathematical notation of a linear model represents a linear relationship between the outcome variable 'charges' and the predictor variables 'age', 'sex', 'BMI', 'children', 'smoker', and 'region'. The equation states that the outcome variable 'charges' is a linear combination of the predictor variables, with each predictor variable multiplied by a specific coefficient (beta). By default the categorical features are dummy encoded, to convert the categorical variable into multiple binary features, and then use these binary features as independent variables in the linear regression model. This allows the model to learn separate regression coefficients for each category of the original categorical variable.

$$charges = \beta_0 + \beta_1 * age + \beta_2 * sexmale + \beta_3 * BMI + \beta_4 * children + \beta_5 * smokeryes + \beta_6 * regionnorthwest + \beta_7 * regionsoutheast + \beta_8 * regionsouthwest$$

In this equation, 'charges' is represented as charges, the coefficients are represented as β_0, β_1, β_2, β_3, β_4, β_5, β_6, β_7, β_8 and predictor variables 'age', 'sex', 'BMI', 'children', 'smoker', and 'region' are represented as age, sexmale, BMI, children, smokeryes, regionnorthwest, regionsoutheast and regionsouthwest respectively. Note, that sexfemale is indirectly encoded in sexmale being 0, smokerno is indirectly encoded in smokeryes being 0 and regionnortheast in all other region variables being 0.

The equation can be used to make predictions about the outcome variable 'charges' based on the values of the predictor variables. For example, if we know the values of 'age', 'sex', 'BMI', 'children', 'smoker', and 'region' for a specific individual, we can plug those values into the equation and solve for 'charges'. The result of this calculation would be an estimate of the individual's 'charges'.

If our linear model has good predictability, we can also interpret the coefficients on how they influence the outcome. As we found no strong multicollinearity during EDA, we can do that in the following. This is called regression analysis.

4.1.2 Hyperparameter Optimization

Since we use the linear model as a baseline, we don't conduct the hyperparameter optimization on that model.

```
lm1 <- lm(charges ~ ., train)
summary(lm1)
```

```
Call:
lm(formula = charges ~ ., data = train)

Residuals:
   Min     1Q Median     3Q    Max
-11078  -3021  -1085   1448  29854

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)     -11854.00    1282.48  -9.243  < 2e-16 ***
age                259.98      15.47  16.805  < 2e-16 ***
sexmale             42.63     429.42   0.099  0.92095
bmi                327.40      36.92   8.867  < 2e-16 ***
children           477.80     174.61   2.736  0.00635 **
smokeryes        23594.95     536.18  44.005  < 2e-16 ***
```

```
regionnorthwest    242.35     597.11   0.406  0.68494
regionsoutheast   -683.86     615.71  -1.111  0.26705
regionsouthwest  -1391.54     612.99  -2.270  0.02347 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6042 on 793 degrees of freedom
Multiple R-squared:  0.7468,    Adjusted R-squared:  0.7442
F-statistic: 292.3 on 8 and 793 DF,  p-value: < 2.2e-16
```

4.1.3 Performance

Finally, we predict the outcome on the validation set and calculate RMSE and R2 scores.

```
pred.lm.valid <- predict(lm1, valid, se.fit = TRUE)
RMSE(pred.lm.valid$fit, valid$charges)
```

```
[1] 5914.225
```

```
R2(pred.lm.valid$fit, valid$charges)
```

```
[1] 0.7952659
```

Model	$RMSE_{valid}$	$R2_{valid}$
Linear Model (Baseline)	5914.225	0.7953

The R2 score of 0.79 is good enough for interpreting the coefficients. From the model summary we can see hat the following explanatory variables are statistically significant:

- Age has an positive effect on the charges. One unit change on age results in an increase of the charges of **259.98$**
- BMI has an positive effect on the charges. One unit change on BMI results in an increase of the charges of **327.40$**
- Children has an positive effect on the charges. One unit change on Children results in an increase of the charges of **477.80$**
- Smoker has an positive effect on the charges. Being a smoker results in an increase of the charges of **23594.95$**
- Age has an positive effect on the charges. One unit change on age results in an increase of the charges of **259.98$**

- BMI has an positive effect on the charges. One unit change on BMI results in an increase of the charges of **327.40$**
- Children has an positive effect on the charges. One unit change on Children results in an increase of the charges of **477.80$**
- BMI has an positive effect on the charges. One unit change on BMI results in an increase of the charges of **327.40$**
- Smoker has an positive effect on the charges. Being a smoker results in an increase of the charges of **23594.95$**

4.2 Algorithm 1 - Regression Tree

4.2.1 Mathematical Overview

A regression tree is a specific class of tree which will predict a numerical dependent variable based on a number of explanatory variables. They can deal with the situation in which the explanatory variables interact with one another (BMI and age are in all likelihood in someway dependent).

This can be difficult to model through simpler models like linear regression; however, by partitioning the data into smaller regions, the interactions between the explanatory variables, potentially very complex, can become more manageable. In a tree model, this partitioning and sub-partitioning (i.e. recursively partitioning each partition) is done by the tree.

We want a set of predictor variables $X_1, ..., X_P$, into some number J distinct and mutually exclusive regions (partitions) $R_1, ..., R_J$. In this description, each observation that falls into the region R_j will have the same value assigned to it. These regions are decided upon as follows: we want to find a selection of regions $R_1, ..., R_J$ such that the residual squared error (RSS) is minimised. I.e. minimise:

$\Sigma_{j=1}^{J} \Sigma_{i \in R_j} (y_i - \hat{y}_{R_j})^2$

where $\hat{y}_{R_j}$ is the mean value for the predictor variable in that given region based on the training data.

The splitting is done in a top-down, greedy manner, where the splitting starts at the top of the tree (i.e. all data-points belong to the same region), then this is split in 2, and this process is repeated for each split in the tree. It is greedy in the sense that it does not look ahead steps into the future; it simply predicts the best split for the split in question. The actual splitting is done as follows.

A predictor variable X_j and a splitting point s such that the splitting regions $X|X_j < s$ and $X|X_j \geq s$ provides the smallest possible RSS. This is then repeated for the other predictor variables until the predictor variable which gave the value of s which gave the smallest RSS is found and this is chosen as the variable for this split in the tree. We do this until a stopping criteria is found, one which will ideally neither underfit nor overfit the tree.

In our case, we are examining the complexity parameter in order to do this. Of course, if we completely overfit the tree, there will be the smallest RSS; however, we want to penalise overfitting in some way. The approach used here is penalised least squares (PLS).

Let us call our full tree T_0. We want to find a pruned tree $T_\alpha \subset T_0$ such that

$PLS(\alpha) = \Sigma_{j \in |T_\alpha|}^{J} \Sigma_{i \in R_j} (y_i - \hat{y}_{R_j})^2 + \alpha |T_\alpha|$

with α greater than or equal to zero, and the other variables as defined already. α is the complexity parameter. It can be found using K-fold cross validation.

The rule of thumb that is used by the rpart package is to take the largest value of the complexity parameter that is within 1 standard deviation of the smallest value out of the cross validation setup.

4.2.2 Hyperparameter Optimization

This is how the rpart regression tree works out of the box: the standard hyperparameter settings form a baseline of the regression trees. Rpart uses 10-fold cross validation as its default to find the complexity parameter. The tree looks as follows. To explain how to interpret this output, the first split is made on whether or not one is a smoker; if one is not a smoker, the next split is based on age; otherwise, the next split is made on BMI.

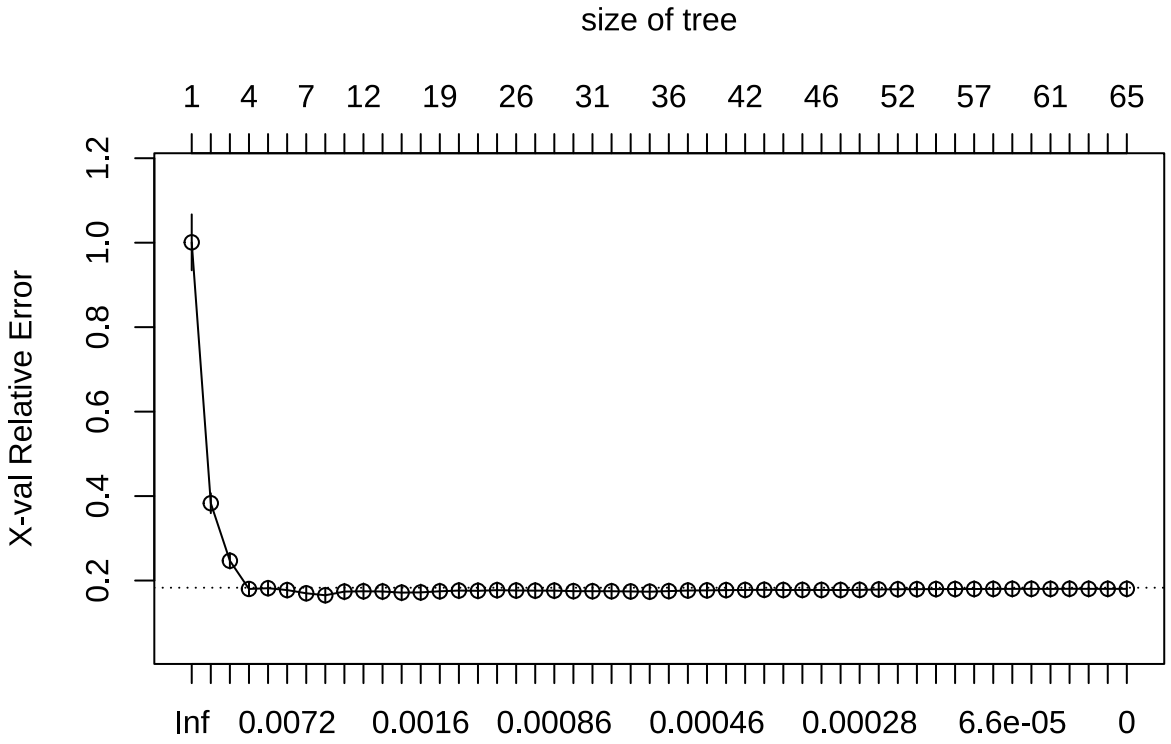
```
n= 802

node), split, n, deviance, yval
      * denotes terminal node

1) root 802 114336300000 13335.670
  2) smoker=no 641  23372910000  8626.180
    4) age< 42.5 355   8058769000  5518.603 *
    5) age>=42.5 286   7630553000 12483.490 *
  3) smoker=yes 161  20143560000 32085.890
    6) bmi< 30.01 77   1671806000 21685.210 *
    7) bmi>=30.01 84   2507060000 41619.850 *
```

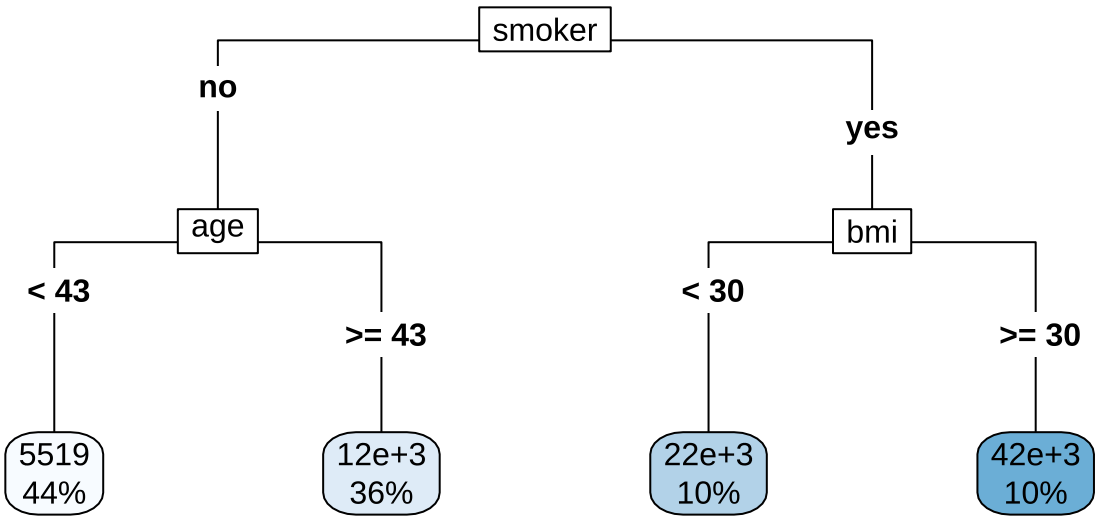
The next tree to be formed will be a full tree (i.e. totally overfitted) as discussed above, with complexity parameter equaling 0, which will then be pruned back.

We can see in the below plot how the error decreases as the complexity parameter decreases.

Based on this plot a suitable complexity parameter is selected, and this quite interpretable decision tree is shown.

Note, there exist other error measures for the splitting criteria and this will be mentioned in the section on future work. Note that the cp=0.019 was gotten from reading the cpmatrix and applying the rule of picking a cp within 1sd of the smallest error; cpmatrix has been commented out to avoid it being printed in the report. The splitting rule, now shown in an easy-to-read plot, is similar to the one outputted by rpart's default settings.





Let us predict and get some metrics as was done for the baseline model. Do this for the validation set.

This is initially done on the full, overfitted tree, then on the default one, and then on the one with the selected complexity parameter

```
pred.train.full <- predict(tree.data.full, valid, se.fit = TRUE)

#Validation data
RMSE(pred.train.full, valid$charges)
```

```
[1] 4770.11
```

```
R2(pred.train.full, valid$charges)
```

```
[1] 0.8643816
```

Repeat for the other two models Default:

```
# out of the box tree model
pred.train.default <- predict(tree.data, valid, se.fit = TRUE)
RMSE(pred.train.default, valid$charges)
```

```
[1] 5076.103
```

```
R2(pred.train.default, valid$charges)
```

```
[1] 0.8478985
```

Pruned:

```
# Chosen complexity parameter based on the theory and graph
pred.train.pruned <- predict(prune.data, valid, se.fit = TRUE)
RMSE(pred.train.pruned, valid$charges)
```

```
[1] 5076.103
```

```r
R2(pred.train.pruned, valid$charges)
```

```
[1] 0.8478985
```

Note that the default method and the one found via our rule of thumb find the same value and thus the same RMSE and R2 values.

We will do some quick hyperparameter optimisation to finding the optimal complexity parameter via grid search:

```r
param_grid <- expand.grid(cp = seq(0.01, 0.5, 0.01))

regtree <- train(
  charges ~ .,
  data = train,
  method = "rpart",
  trControl = trainControl(method = "cv", number = 10),
  tuneGrid = param_grid
)

print(regtree$bestTune)
```

```
    cp
6 0.06
```

```r
predictions <- predict(regtree, newdata = valid)

RMSE(predictions, valid$charges)
```

```
[1] 5076.103
```

```r
R2(predictions, valid$charges)
```

```
[1] 0.8478985
```

It clearly finds a different cp than before; however, it provides the same RMSE and R2 as the other two trees discussed. This is perhaps unsurprising as they all are creating the same tree (compare first and last plots). Hence this tree seems to offer us the best solution.

4.2.3 Performance

For a single tree you can interpret a fitted model by inspecting the text output of the tree or the tree diagram itself. We could also calculate the Increase in Node Purity and Mean decrease in Accuracy, but that is most often done for ensemble methods, as you e.g can not plot all trees of a random forest. Given that Smoking, BMI and Age are the only variables considered by our 'best' model, we can consider that these variables are the most important.

Finally, we predict the outcome of the final tree, found via hyperparameter optimisation, on the validation set and calculate RMSE and R2 scores.

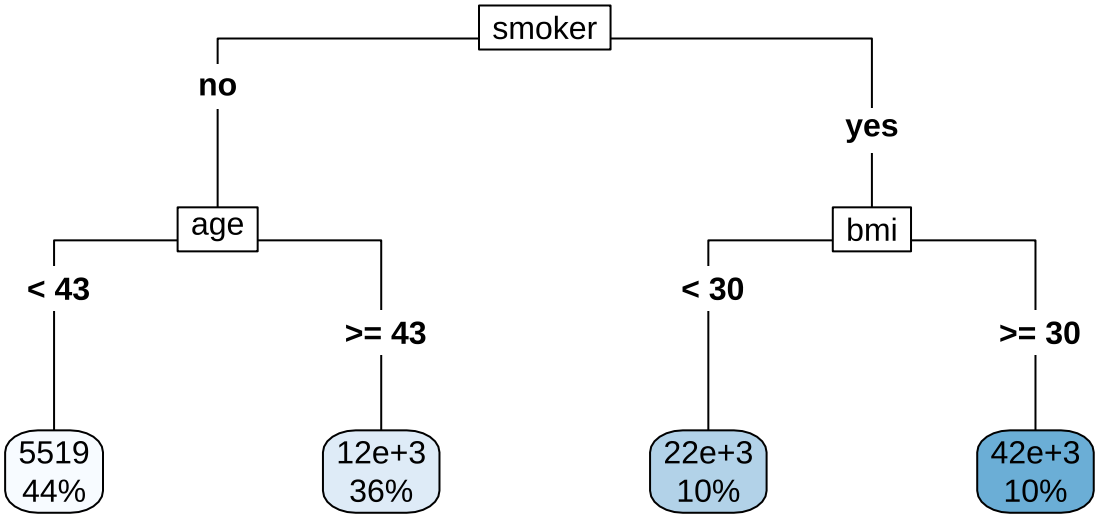
Model	$RMSE_{valid}$	$R2_{valid}$
Regression Tree	5076.103	0.8479

The final tree itself looks as follows:

```
printcp(rpart(charges~.,train))
```

```
Regression tree:
rpart(formula = charges ~ ., data = train)

Variables actually used in tree construction:
[1] age    bmi    smoker

Root node error: 1.1434e+11/802 = 142563982

n= 802

        CP nsplit rel error  xerror     xstd
1 0.619399      0   1.00000 1.00447 0.066367
2 0.139629      1   0.38060 0.38362 0.023519
3 0.067202      2   0.24097 0.24615 0.018445
4 0.010000      3   0.17377 0.17898 0.016712
```

```
rpart.plot(rpart(charges~.,train),type=5)
```



4.3 Algorithm 2 - Neural Networks

4.3.1 Mathematical Overview

A neural network is a type of machine learning model that is inspired by the structure and function of the human brain. It is composed of layers of interconnected nodes, or "neurons," that process and transmit information. Neural networks are capable of handling complex and non-linear relationships between input and output variables.

A neural network can be represented mathematically as a series of layers, with each layer performing a set of mathematical operations on the input it receives. The output of one layer serves as the input to the next layer. The first layer, called the input layer, receives the input variables, and the last layer, called the output layer, produces the output variable(s). In between the input and output layers, there are one or more hidden layers that perform computations on the input data.

The mathematical notation for a neural network can be quite complex as it depends on the architecture of the network, the number of layers and the activation functions used. In general, a neural network can be represented mathematically as:

$y = f(W_n * f(W_{n-1} * f(...f(W_1 * x + b_1) + b_2)... + b_n) + b_{n+1})$

Where:

- y = output variable
- x = input variable
- f = activation function (such as sigmoid, ReLU, etc.)
- W = weight matrix
- b = bias term

This equation is showing a simplified version of a feedforward neural network with one hidden layer, the input, hidden and output layers.

4.3.2 Implementation

4.3.2.1 Python libraries To train the neural network model, we use Python and the following libraries:

- **pandas** and **numpy** for wrangling data
- **sklearn** for preprocessing data
- **torch** for defining the neural network model
- **pytorch_lightning** for running and logging the training
- **optuna** for hyperparameter optimization

```
import pandas as pd
import numpy as np

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD, Adam

from pytorch_lightning.loggers import TensorBoardLogger, logger
import pytorch_lightning as pl
from pytorch_lightning.callbacks import ModelCheckpoint
from pytorch_lightning.callbacks.early_stopping import EarlyStopping

import optuna
```

4.3.2.2 Data preprocessing To preprocess the data, we transformed the categorical variables using **LabelEncoder** and scaled the numerical data using **MinMaxScaler** from **sklearn** library:

```
def labelEncoder(df):
    for col in df.columns:
        if df.dtypes[col] == "object":
            df[col] = LabelEncoder().fit_transform(df[col])
```

```
            df[col] = df[col].astype('category')
    return df

cols_to_scale = ['age', 'bmi', 'children']
scaler = MinMaxScaler()
train[cols_to_scale] = scaler.fit_transform(train[cols_to_scale])
val[cols_to_scale] = scaler.transform(val[cols_to_scale])
test[cols_to_scale] = scaler.transform(test[cols_to_scale])
```

4.3.2.3 Dataset Architecture

To run the training process, we have to define custom data classes first, **trainDataset**, **valDataset**, and **testDataset** that are used to load and preprocess the data for training, validation and testing respectively.

Each dataset class takes one parameter, which is a DataFrame **df** containing the data, and creates two attributes, **X** and **y** from the dataframe. **X** is a tensor containing the input features 'age', 'sex', 'bmi', 'children', 'smoker', 'region' and **y** is a tensor containing the target variable 'charges'.

The **len** method returns the number of data points in the dataset, and the **getitem** method is used to retrieve a specific data point at a given index.

```
class trainDataset(torch.utils.data.Dataset):
    def __init__(self, df):
        self.X = torch.tensor(df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].value
        self.y = torch.tensor(df['charges'].values, dtype=torch.float32)

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

class valDataset(torch.utils.data.Dataset):
    def __init__(self, df):
        self.X = torch.tensor(df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].value
        self.y = torch.tensor(df['charges'].values, dtype=torch.float32)

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
class testDataset(torch.utils.data.Dataset):
    def __init__(self, df):
        self.X = torch.tensor(df[['age', 'sex', 'bmi', 'children', 'smoker', 'region']].value
        self.y = torch.tensor(df['charges'].values, dtype=torch.float32)

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

4.3.2.4 Model Architecture

The following section describes the model architecture of the neural network model.

The **RegressionModel** class is a subclass of PyTorch's LightningModule, which is a high-level wrapper for PyTorch's neural network module.

The class takes four parameters:

- layers: an integer representing the number of hidden layers in the network
- num_neurons: an integer representing the number of neurons in each hidden layer
- learning_rate: a float representing the learning rate for the optimizer
- batch_size: an integer representing the batch size to use during training

The **init** method is called when the class is first instantiated and sets up the neural network architecture. It creates a series of hidden layers using the helper function hidden_layer. The input layer of the network is defined with 6 input features, and the output layer has 1 output feature.

Each hidden layer is defined as a linear layer followed by a ReLU activation function.

The **forward** method defines the computation that takes place within the neural network, where input data is passed through the layers and transformed into output predictions.

The **training_step**, **validation_step**, and **test_step** methods define what happens during each step of the training, validation, and test processes respectively. They calculate the mean squared error loss between the predictions and the true values, log the loss and return it for optimization.

The **configure_optimizers** method sets the optimizer to use for training, in this case, the **Adam** optimizer with the specified learning rate.

The **train_dataloader**, **val_dataloader** and **test_dataloader** methods return the data loaders for training, validation, and test dataset respectively.

```
def hidden_layer(in_f, out_f):
    return torch.nn.Sequential(
        torch.nn.Linear(in_f, out_f),
        torch.nn.ReLU()
    )

class RegressionModel(pl.LightningModule):
    def __init__(self, layers, num_neurons, learning_rate, batch_size):
        super().__init__()
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.num_neurons = num_neurons
        hidden = [hidden_layer(num_neurons, num_neurons) for _ in range(layers)]
        self.input = torch.nn.Linear(6, num_neurons)
        self.hidden = torch.nn.Sequential(*hidden)
        self.output = torch.nn.Linear(num_neurons, 1)
        self.relu = torch.nn.ReLU()
        self.save_hyperparameters()

    def forward(self, x):
        x = self.input(x)
        x = self.relu(x)
        x = self.hidden(x)
        x = self.output(x)
        return x

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.forward(x).squeeze(dim=1)
        loss = torch.nn.functional.mse_loss(y_hat, y)
        self.log('train_loss', loss)
        return {'loss': loss}

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.forward(x).squeeze(dim=1)
        loss = torch.nn.functional.mse_loss(y_hat, y)
        self.log('val_loss', loss)
        return {'val_loss', loss}

    def test_step(self, batch, batch_idx):
        x, y = batch
```

```
        y_hat = self(x).squeeze(dim=1)
        loss = F.mse_loss(y_hat, y)
        self.log('test_loss', loss)
        return {'test_loss', loss}

    def validation_end(self, validation_step_outputs):
        avg_loss = torch.stack([x['val_loss'] for x in validation_step_outputs]).mean()
        self.log('avg_val_loss', avg_loss)
        return {'avg_val_loss': avg_loss}

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=self.learning_rate)

    def train_dataloader(self):
        return torch.utils.data.DataLoader(train_dataset, batch_size=self.batch_size, num_wo

    def val_dataloader(self):
        return torch.utils.data.DataLoader(val_dataset, batch_size=self.batch_size, num_work

    def test_dataloader(self):
        return torch.utils.data.DataLoader(test_dataset, batch_size=self.batch_size, num_wor
```

The mathematical notation for the neural network model defined above can be represented as a series of layers, with each layer performing a set of mathematical operations on the input it receives. The output of one layer serves as the input to the next layer. The mathematical notation for each hidden layer would be:

$h_i = f(W_i * h_{i-1} + b_i)$

Where:

- h_i is the output of the i-th hidden layer
- h_{i-1} is the output of the (i-1)-th hidden layer
- W_i is the weight matrix for the i-th hidden layer
- b_i is the bias term for the i-th hidden layer
- f is the activation function (ReLU)

The mathematical notation for the input layer is:

$h_0 = W_0 * x + b_0$

Where:

- h_0 is the output of the input layer
- W_0 is the weight matrix for the input layer

- x is the input data
- b_0 is the bias term for the input layer

The mathematical notation for the output layer is:

$y = W_{n+1} * h_n + b_{n+1}$

Where:

- y is the output variable
- h_n is the output of the last hidden layer
- W_{n+1} is the weight matrix for the output layer
- b_{n+1} is the bias term for the output layer

The optimizer used in the model above is the Adam optimizer. Adam is an optimization algorithm that combines the benefits of the Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). The mathematical notation for the Adam optimizer is:

$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$

Where:

- θ is the set of parameters to optimize
- α is the learning rate
- $\hat{m}_t$ is the biased first moment estimate
- $\hat{v}_t$ is the biased second raw moment estimate
- ϵ is a small positive constant that is added to the denominator to prevent division by zero

It's worth noting that the Adam optimizer also uses the concepts of momentum and adaptive learning rate which are not reflected in the mathematical notation above.

4.3.2.5 Training the neural network

Next, we want to test the model with predefined parameter set. Therefore, we are using the **RegressionModel** class defined earlier to create an instance of the model, and then training the model using the **Trainer** class from the **PyTorch Lightning** library.

The **TensorBoardLogger** is used to log the training progress and the results, it is passed to the **Trainer** class and it will save the logs into predefined folder.

The **RegressionModel** instance is created with the following parameters:

- num_neurons: 64
- layers: 3
- learning_rate: 0.001

- batch_size: 32

The **Trainer** class is initialized with the following parameters:

- max_epochs: 50
- logger: the **TensorBoardLogger** created earlier
- fast_dev_run: False, that means the model will run through the full training process
- auto_lr_find: True, this will use the learning rate finder to find a good initial learning rate before training.

The fit method is called on the trainer object with the model as its argument, this will start the training process. The trainer will take care of the training loop, logging progress, and saving the model.

```
logger = TensorBoardLogger('lightning_logs', name='insurance')
model = RegressionModel(num_neurons=64, layers=3, learning_rate=0.001, batch_size=32)
trainer = pl.Trainer(max_epochs=50, logger=logger, fast_dev_run=False, auto_lr_find=True)
trainer.fit(model)
```

Before the training, we can check how many weights will be used in the model:

	Name	Type	Params
0	input	Linear	448
1	hidden	Sequential	12.5 K
2	output	Linear	65
3	relu	ReLU	0

Total: 13.0 K Trainable params

After the training of the initial model, we received the following MSE scores:

Model	MSE_{train}	MSE_{valid}
Neural Network (initial)	34806204	35508452

4.3.3 Hyperparameter Optimization

The HPO process on the neural network model was performed using **Optuna** library in Python.

We let Optuna optimize 4 parameters for us:

- 'num_neurons': number of neurons in each layer, between 16 and 256
- 'batch_size': amount of data points used at once, between 32 and 512
- 'layers': amount of layers, between 1 and 5
- 'learning_rate': learning rate of the Adam optimizer, between 1e-5 and 1e-1

```
def optimize_model(trial):

    num_neurons = trial.suggest_int('num_neurons', 16, 256)
    batch_size= trial.suggest_int('batch_size', 32, 512)
    learning_rate= trial.suggest_float('learning_rate', 1e-5, 1e-1)
    layers = trial.suggest_int('layers', 1, 5)

    model = RegressionModel(layers=layers, num_neurons=num_neurons, learning_rate=learning_ra
    #trainer = pl.Trainer(accelerator='mps', max_epochs=50)
    trainer = pl.Trainer(max_epochs=50)

    trainer.fit(model)
    return trainer.callback_metrics['val_loss']

study = optuna.create_study(direction='minimize', study_name='insurance', storage='sqlite:///
study.optimize(optimize_model, n_trials=100)
```

As the result of the HPO, the the following parameters were found to perform the best on the validation set:

- 'num_neurons': 235
- 'batch_size': 314
- 'layers': 5
- 'learning_rate': 0.0748

4.3.4 Performance

Finally, we predict the outcome on the validation set and compare MSE values.

Model	MSE_{valid}
Neural Network (initial)	35508452
Neural Network (optimized)	18894440

We can see that the optimized model could achieve almost 50% improvement in terms of MSE value, which indicates that the performance of the model has been improved a lot.

5 Model Comparison

In this chapter, we want to present a fair comparison of the models. Therefore, we use the test set, which none of the model have seen yet and predict outcome variable using each model's best parameters, which we determined using HPO.

We compare RMSE and R2 scores for the baseline model, for the regression tree model and for the neural network model.

Model	$RMSE_{test}$	$R2_{test}$
Linear Model (Baseline)	6328.782	0.7118
Regression Tree	5152.172	0.8079
Neural Network	4881.731	0.8274

We can conclude that both models outperform the baseline model, but neural network seems to perform the best.

6 Future Work and Discussion

In this study, we evaluated the performance of three different models: a linear model, a regression tree, and a neural network. Our results showed that the regression tree outperformed the linear model on both of the chosen metrics. The neural network, in turn, outperformed both the linear model and the regression tree.

The linear model has the advantage of being the easiest to interpret of all three models, even though its performance is not as good as the other two models. The difference in performance between the regression tree and the linear model is significant, but the increase in performance of the neural network over the tree model is relatively small. This suggests that the regression tree may be a more suitable model for this task because it offers greater explainability in terms of how it can be visualized, easily explained to non-experts, and understood. The neural network, on the other hand, has some of the "black box" problems that some neural networks are prone to; it is difficult to "see inside" the model to understand how it is making its decisions. Note however, that the regression tree is also not a perfect model, as it only predicts 4 values, as we saw in the tree plot. This is not ideal for predicting a continuous outcome.

For future work, we could attempt to improve the interpretability of the neural network by using model-agnostic tools such as SHAP values. Additionally, we could improve the current regression tree by including other parameters into the hyperparameter optimization, testing different splitting criteria such as gini index vs. entropy. If that doesn't improve performance,

we could also build multiple trees and use the ensemble method Random Forest. However, this would reduce interpretability again.

In conclusion, the choice of model depends on the desired balance between performance and interpretability. While the neural network is the model that showed the best performance, the regression tree may be a more suitable choice for this task because of its greater explainability.