Predicting ten year risk of coronary heart disease - Project Log

Dennis Wiersma

2022-11-08

Contents

1 Exploratory Data Analysis **4**
- 1.1 Dataset . . . 4
  - 1.1.1 Loading the data . . . 4
  - 1.1.2 Datatype correction . . . 6
  - 1.1.3 Missing values . . . 7
- 1.2 Visualisations . . . 11
  - 1.2.1 Boxplots split on class variable . . . 11
  - 1.2.2 Boxplots split on class variable - Log transformation . . . 12
  - 1.2.3 Class distributions . . . 13
  - 1.2.4 Correlation . . . 16
  - 1.2.5 PCA - Principal Component Analysis . . . 17
- 1.3 Exporting data . . . 18

2 Machine Learning algorithm **19**
- 2.1 Algorithm metrics . . . 19
  - 2.1.1 Quality & performance metrics . . . 19
  - 2.1.2 Algorithm exploration . . . 21
  - 2.1.3 Algorithm optimisation . . . 23
  - 2.1.4 Meta-learners . . . 24
  - 2.1.5 Final tuning . . . 25
  - 2.1.6 Final model . . . 25
- 2.2 Java wrapper . . . 26

List of Figures

1 Collection of boxplots for each numeric variable, split on class variable. 12
2 Collection of boxplots for each log transformed numeric variable, split on class variable. . . . 13
3 Collection of barplots for every variable, coloured by class variable. 15
4 Matrix of degree of correlation between variables. 16
5 PCA plot charting PC2 against PC1. 18

List of Tables

1 Codebook with variable descriptions. 4
2 First six values in the education column of the dataset. 5
3 List of variables that lack units. 5
4 Number of missing values for each variable that contains missing values. 7
5 All instances of cigsPerDay that are missing values. 8
6 Example confusion matrix. 20
7 Overview of algorithm scoring metrics. 20
8 References to ML algorithms and their (default) settings. 21
9 Comparison of ML algorithms using default settings using ZeroR as a baseline. 21
10 Comparison of ML algorithms using default settings after performing SMOTE on the dataset, using ZeroR as a baseline. 22
11 Comparison of ML algorithms using default settings and cost sensitive classification after performing SMOTE on the dataset, using ZeroR as a baseline. 22
12 Comparison of ZeroR using varying bucket sizes while using cost sensitive classification after performing SMOTE on the dataset, using ZeroR's default settings as a baseline. 23
13 Comparison of RandomForest using a varying number of iterations while using cost sensitive classification after performing SMOTE on the dataset, using RandomForest's default settings as a baseline. 23
14 Comparison of bagging with RandomForests using a varying number of bagging iterations while using cost sensitive classification after performing SMOTE on the dataset, using a RandomForest without bagging as a baseline. 24
15 Comparison of boosting with RandomForests using a varying number of boost iterations while using cost sensitive classification after performing SMOTE on the dataset, using a RandomForest without boost as a baseline. 24
16 Comparison of boosting with RandomForests using a varying number of boost iterations while using cost sensitive classification after performing SMOTE on the dataset, using a RandomForest without boost as a baseline. 25
17 Comparison of boosting with RandomForests using a varying number of boost iterations while using cost sensitive classification after performing SMOTE on the dataset, using a RandomForest without boost as a baseline. 25
18 Final model which utilises boosting with RandomForests while using cost sensitive classification after performing SMOTE on the dataset. 26

1 Exploratory Data Analysis

Cardiovascular disease gets more prominent by the day. As one of the leading causes of death in the developed world it is an important issue to examine. Getting to know more about this matter and what factors impact risk of complications can therefore be rather interesting. This chapter describes the Exploratory Data Analysis which functions as a kick off point for this project.

1.1 Dataset

The dataset used in this research project contains a subset of data gathered from an ongoing cardiovascular study on residents of the town of Framingham, Massachusetts, United States of America. This study is better known as the Framingham Heart Study (FHS).

1.1.1 Loading the data

The subset of the large collection of data used for this research contains over 4000 records and 15 different variables. In this section all necessary files will be loaded for analysis.

1.1.1.1 Loading the codebook The codebook is a document describing variables and listing additional information like units and datatypes.
The codebook will be loaded as follows:

```
# Load codebook
codeBook <- read.table(file = "../CodeBook.txt", header = TRUE, sep = "|")
# Create table w/ codebook contents
knitr::kable(codeBook, "simple", caption = "Codebook with variable descriptions.")
```

Table 1: Codebook with variable descriptions.

name	description	unit	statType	dataType
male	gender, male or female	NA	nominal	bool
age	age	years	continuous	int
education	presumably education level	NA	NA	int
currentSmoker	whether the subject is a smoker	NA	nominal	bool
cigsPerDay	average number of smoked cigarettes per day	cigarettes per day	continuous	int
BPMeds	whether the subject is on blood pressure medication	NA	nominal	bool
prevalentStroke	whether the subject has previously had a stroke	NA	nominal	bool
prevalentHyp	whether the subject is hypertensive	NA	nominal	bool
diabetes	whether the subject is diabetic	NA	nominal	bool
totChol	total cholesterol level	NA	continuous	int
sysBP	systolic blood pressure	NA	continuous	int
diaBP	diastolic blood pressure	NA	continuous	int
BMI	Body Mass Index	kg/m2	continuous	double
heartRate	heart rate	beats per minute	continuous	int
glucose	glucose level	NA	continuous	int
TenYearCHD	ten year risk of coronary heart disease	NA	nominal	bool

1.1.1.2 Loading the dataset To analyse our dataset we'll of course need access to it. It will be loaded with the code below:

```
# Load dataset
dataset <- read.table(file = "../data/framingham.csv", header = TRUE, sep = ",")
# Get dimensions of dataset
dataset.dim <- dim(dataset)
```

```
# Print dimensions of dataset
cat("Instances: ", dataset.dim[1], "\nvariables: ", dataset.dim[2])
```

```
## Instances:  4238
## variables:  16
```

1.1.1.3 Dataset inconsistencies

The dataset mentioned above was found on the Kaggle website. There are a few problems with this data. First off there is the "education" variable. This variable consists of values 1 through 4, presumably to indicate education levels. However, what these numbers represent is not explained anywhere in the dataset's description. Thus far it is still unclear what exactly these values specify.
An example of the education variable:

```
# Get head of education column and print table
knitr::kable(head(dataset["education"]), "simple", caption = "First six values in the education column
```

Table 2: First six values in the education column of the dataset.

education
4
2
1
3
3
2

Also included in this dataset are some numeric variables, described on the dataset's page as "continuous". These values all lack descriptions of their units. Where a variable's unit can sometimes easily be inferred, like with heart rate, other times it means we end up with no unit whatsoever.
The following variables have missing units:

```
# Print codebook entries where unit is NA
knitr::kable(codeBook[which(is.na(codeBook["unit"])), c("name", "unit")], caption = "List of variables
```

Table 3: List of variables that lack units.

	name	unit
1	male	NA
3	education	NA
4	currentSmoker	NA
6	BPMeds	NA
7	prevalentStroke	NA
8	prevalentHyp	NA
9	diabetes	NA
10	totChol	NA
11	sysBP	NA
12	diaBP	NA
15	glucose	NA
16	TenYearCHD	NA

To resolve these missing units one may look for the source of the data. The source mentioned on the Kaggle page links to another page on Kaggle. The problem is that this page no longer exists. We do however know that this data originates from the Framingham Heart Study (FHS). Searching through this website for our

missing units led to this page about available FHS data. From there we find a page on FHS Dataset Inventory for Public Viewing which lists all datasets which can be requested from the FHS team. This page includes documentation on variables. This sounds like progress but there are fifty pages of datasets to choose from. Finding out which of these datasets was used to source the data used in this project would be too time intensive. Therefore, there is no other choice but to work with what we have and use the data without it's corresponding units.

1.1.2 Datatype correction

```
str(dataset)
```

```
## 'data.frame':    4238 obs. of  16 variables:
##  $ male           : int  1 0 1 0 0 0 0 0 1 1 ...
##  $ age            : int  39 46 48 61 46 43 63 45 52 43 ...
##  $ education      : int  4 2 1 3 3 2 1 2 1 1 ...
##  $ currentSmoker  : int  0 0 1 1 1 0 0 1 0 1 ...
##  $ cigsPerDay     : int  0 0 20 30 23 0 0 20 0 30 ...
##  $ BPMeds         : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ prevalentStroke: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ prevalentHyp   : int  0 0 0 1 0 1 0 0 1 1 ...
##  $ diabetes       : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ totChol        : int  195 250 245 225 285 228 205 313 260 225 ...
##  $ sysBP          : num  106 121 128 150 130 ...
##  $ diaBP          : num  70 81 80 95 84 110 71 71 89 107 ...
##  $ BMI            : num  27 28.7 25.3 28.6 23.1 ...
##  $ heartRate      : int  80 95 75 65 85 77 60 79 76 93 ...
##  $ glucose        : int  77 76 70 103 85 99 85 78 79 88 ...
##  $ TenYearCHD     : int  0 0 0 1 0 0 1 0 0 0 ...
```

When looking at the structure of our dataset above one might notice the fact that some of the data types aren't quite right. For example, a lot of our nominal values are encoded as integers. We shall fix this now:

```
# Change male column from int to female/male factor
dataset$male <- factor(dataset$male, labels = c("female", "male"))
# Change currentSmoker column from int to no/yes factor
dataset$currentSmoker <- factor(dataset$currentSmoker, labels = c("no", "yes"))
# Change BPMeds column from int to no/yes factor
dataset$BPMeds <- factor(dataset$BPMeds, labels = c("no", "yes"))
# Change prevalentStroke column from int to no/yes factor
dataset$prevalentStroke <- factor(dataset$prevalentStroke, labels = c("no", "yes"))
# Change prevalentHyp column from int to no/yes factor
dataset$prevalentHyp <- factor(dataset$prevalentHyp, labels = c("no", "yes"))
# Change diabetes column from int to no/yes factor
dataset$diabetes <- factor(dataset$diabetes, labels = c("no", "yes"))
# Change TenYearCHD column from int to no/yes factor
dataset$TenYearCHD <- factor(dataset$TenYearCHD, labels = c("no", "yes"))
```

If we look at the data's structure again now, we'll see it looks a lot better.

```
str(dataset)
```

```
## 'data.frame':    4238 obs. of  16 variables:
##  $ male           : Factor w/ 2 levels "female","male": 2 1 2 1 1 1 1 1 2 2 ...
##  $ age            : int  39 46 48 61 46 43 63 45 52 43 ...
##  $ education      : int  4 2 1 3 3 2 1 2 1 1 ...
##  $ currentSmoker  : Factor w/ 2 levels "no","yes": 1 1 2 2 2 1 1 2 1 2 ...
```

```
##  $ cigsPerDay     : int  0 0 20 30 23 0 0 20 0 30 ...
##  $ BPMeds         : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prevalentStroke: Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prevalentHyp   : Factor w/ 2 levels "no","yes": 1 1 1 2 1 2 1 1 2 2 ...
##  $ diabetes       : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ totChol        : int  195 250 245 225 285 228 205 313 260 225 ...
##  $ sysBP          : num  106 121 128 150 130 ...
##  $ diaBP          : num  70 81 80 95 84 110 71 71 89 107 ...
##  $ BMI            : num  27 28.7 25.3 28.6 23.1 ...
##  $ heartRate      : int  80 95 75 65 85 77 60 79 76 93 ...
##  $ glucose        : int  77 76 70 103 85 99 85 78 79 88 ...
##  $ TenYearCHD     : Factor w/ 2 levels "no","yes": 1 1 1 2 1 1 2 1 1 1 ...
```

1.1.3 Missing values

Before we can continue to analysing the dataset with fancy distributions and the likes, we have to decide how to deal with missing values in our dataset. Depending on the amount and type of the missing data there are several methods of approach. Let's take a look at how many missing values each of our variables contains:

```
# Get the 7th row of the dataset's summary (the NA's row)
# Then print the rows where number of NA's > 0
knitr::kable(summary(dataset)[7, which(!is.na(summary(dataset)[7,]))], "simple", caption = "Number of mi
```

Table 4: Number of missing values for each variable that contains missing values.

	x
education	NA's :105
cigsPerDay	NA's :29
totChol	NA's :50
BMI	NA's :19
heartRate	NA's :1
glucose	NA's :388

1.1.3.1 Education

First up is education. We still don't know what the education variable really even means, but one could reasonably assume a lower number to correspond to a lower level of education. However, this approach would be unreliable at best and actively harmful to our conclusions at worst. Therefore it would be best to deal with these missing values by simply getting rid of the entire variable, which we shall do now:

```
# Remove the education column from the dataset
dataset <- subset(dataset, select = -c(education))
```

1.1.3.2 cigsPerDay - Cigarettes smoked per day

This variable only has a few dozen missing values, so it wouldn't be the end of the world to simply get rid of them. There is a better way however. Let's first take a look at these missing values and check whether all of the missing cigsPerDay values are actually from smokers:

```
knitr::kable(dataset[which(is.na(dataset["cigsPerDay"])), c("currentSmoker", "cigsPerDay")], caption =
```

Table 5: All instances of cigsPerDay that are missing values.

	currentSmoker	cigsPerDay
132	yes	NA
140	yes	NA
1047	yes	NA
1293	yes	NA
1348	yes	NA
1452	yes	NA
1498	yes	NA
1611	yes	NA
1626	yes	NA
1871	yes	NA
1964	yes	NA
1981	yes	NA
2406	yes	NA
2514	yes	NA
2543	yes	NA
3022	yes	NA
3035	yes	NA
3095	yes	NA
3107	yes	NA
3109	yes	NA
3157	yes	NA
3178	yes	NA
3310	yes	NA
3433	yes	NA
3580	yes	NA
3716	yes	NA
3848	yes	NA
3925	yes	NA
3943	yes	NA

As can be seen in the table above, all of the present NA values are indeed from current smokers. An easy fix for these missing values would be to assign them the median amount of cigarettes smoked per day *by current smokers*, so that is what we'll do next. We'll also check the number of NA values before and after dealing with them as a check.

```
# Check number of NAs
cat("Number of NAs before: ", sum(is.na(dataset["cigsPerDay"])))
```

```
## Number of NAs before:  29
```

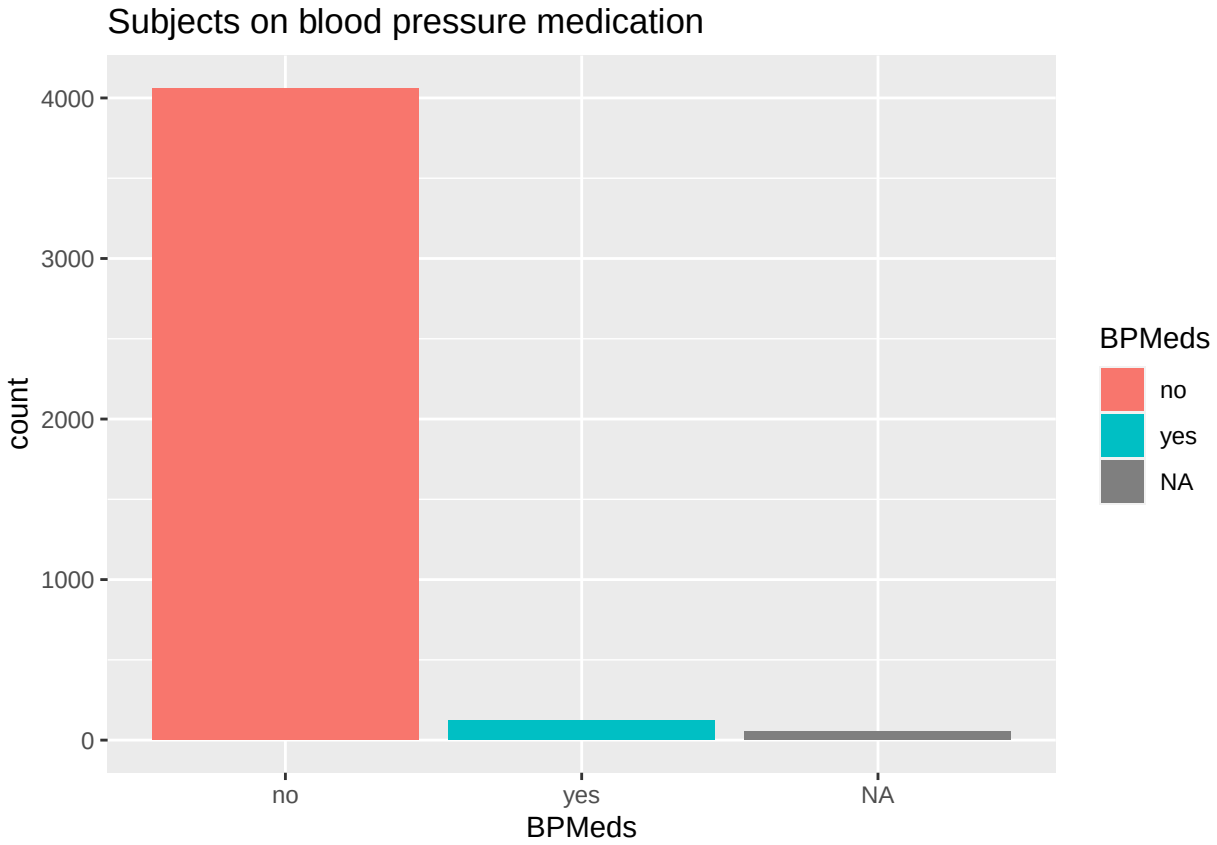
```
# Calculate median cigarettes per day for smokers
medianCigsPerDay <- median(na.omit(dataset["cigsPerDay"][dataset["currentSmoker"] == "yes"]))
# Change number of cigarettes per day to median
dataset[which(is.na(dataset["cigsPerDay"])), "cigsPerDay"] <- medianCigsPerDay
# Check number of NAs
cat("Number of NAs after: ", sum(is.na(dataset["cigsPerDay"])))
```

```
## Number of NAs after:  0
```

1.1.3.3 BPMeds - Blood pressure medication To decide what to do with the missing blood pressure medication values, we shall first look at how balanced this variable is. One might reasonably assume that

subjects that are on blood pressure medication would be in the minority, but it's still worth it to check.

```
ggplot(dataset, aes(x = BPMeds, fill = BPMeds)) +
  geom_bar() +
  ggtitle("Subjects on blood pressure medication")
```



From this plot we may conclude that a large majority of subjects are in fact not on blood pressure medication. We can therefore assume that subjects with missing values for this variable aren't on blood pressure medication either. So let's fix the values:

```
# Check number of NAs
cat("Number of NAs before: ", sum(is.na(dataset["BPMeds"])))
```

```
## Number of NAs before:  53
```

```
# Change missing BPMeds values to "no"
dataset[which(is.na(dataset["BPMeds"])), "BPMeds"] <- "no"
# Check number of NAs
cat("Number of NAs after: ", sum(is.na(dataset["BPMeds"])))
```

```
## Number of NAs after:  0
```

1.1.3.4 totChol - Total cholestorol level Since total cholestorol level is a simple integer variable instead of a nominal variable we can just assign the missing values the median of all the other values.

```
# Check number of NAs
cat("Number of NAs before: ", sum(is.na(dataset["totChol"])))
```

```
## Number of NAs before:  50
```

```
# Calculate median totChol
medianTotChol <- median(na.omit(dataset$totChol))
# Change missing totChol values to median
dataset[which(is.na(dataset["totChol"])), "totChol"] <- medianTotChol
# Check number of NAs
cat("Number of NAs after: ", sum(is.na(dataset["totChol"])))
```

```
## Number of NAs after:  0
```

1.1.3.5 BMI - Body Mass Index

Since BMI is a simple integer variable instead of a nominal variable we can just assign the missing values the median of all the other values.

```
# Check number of NAs
cat("Number of NAs before: ", sum(is.na(dataset["BMI"])))
```

```
## Number of NAs before:  19
```

```
# Calculate median BMI
medianBMI <- median(na.omit(dataset$BMI))
# Change missing BMI values to median
dataset[which(is.na(dataset["BMI"])), "BMI"] <- medianBMI
# Check number of NAs
cat("Number of NAs after: ", sum(is.na(dataset["BMI"])))
```

```
## Number of NAs after:  0
```

1.1.3.6 heartRate - Heart rate

Since heart rate level is a simple integer variable instead of a nominal variable we can just assign the missing values the median of all the other values.

```
# Check number of NAs
cat("Number of NAs before: ", sum(is.na(dataset["heartRate"])))
```

```
## Number of NAs before:  1
```

```
# Calculate median heartRate
medianHeartRate<- median(na.omit(dataset$heartRate))
# Change missing heartRate values to median
dataset[which(is.na(dataset["heartRate"])), "heartRate"] <- medianHeartRate
# Check number of NAs
cat("Number of NAs after: ", sum(is.na(dataset["heartRate"])))
```

```
## Number of NAs after:  0
```

1.1.3.7 glucose - Glucose levels

Since glucose levels is a simple integer variable instead of a nominal variable we can just assign the missing values the median of all the other values.

```
# Check number of NAs
cat("Number of NAs before: ", sum(is.na(dataset["glucose"])))
```

```
## Number of NAs before:  388
```

```
# Calculate median glucose
medianGlucose <- median(na.omit(dataset$glucose))
# Change missing glucose values to median
dataset[which(is.na(dataset["glucose"])), "glucose"] <- medianGlucose
# Check number of NAs
cat("Number of NAs after: ", sum(is.na(dataset["glucose"])))
```

```
## Number of NAs after:  0
```

1.2 Visualisations

Now that we have a complete and workable dataset we can start exploring what our data actually looks like. There's a collection of different plots and figures we'll use to explore our data. However, we are limited in which variables we can use with each analysis. The main differentiating factor here would be whether a variable is numeric or nominal. Certain visualisations cannot be done on nominal data. For example it isn't very useful to compare two factors in a boxplot. We will therefore, on occasion, explore these different types of variables in their own distinct ways.

1.2.1 Boxplots split on class variable

Let us start by looking at the variations in our data. To do this we'll be creating a boxplot for every one of our numeric variables. This will let us examine the variance, as well as some key metrics like the mean, in an easy to grasp visual manner. We split the boxplots up into two groups per variable based on the `TenYearCHD` variable. This let's us get a sense of whether our class variable has an impact on the numeric variable at hand.

```
# function that creates a boxplot on dataset given a column name
boxplots <- function(colName) {
  ggplot(dataset, aes(x = TenYearCHD, y = !!sym(colName), fill = TenYearCHD)) +
  geom_boxplot(show.legend = FALSE) +
  theme(text=element_text(size=8))
}

# vector with all numeric columns
numeric_cols <- c("age", "cigsPerDay", "totChol", "sysBP", "BMI", "heartRate", "glucose")
# execute boxplots function on every numeric column
plot_list <- lapply(numeric_cols, boxplots)
# arrange these boxplots into one figure
ggarrange(plotlist = plot_list, ncol = 4, nrow = 2)
```

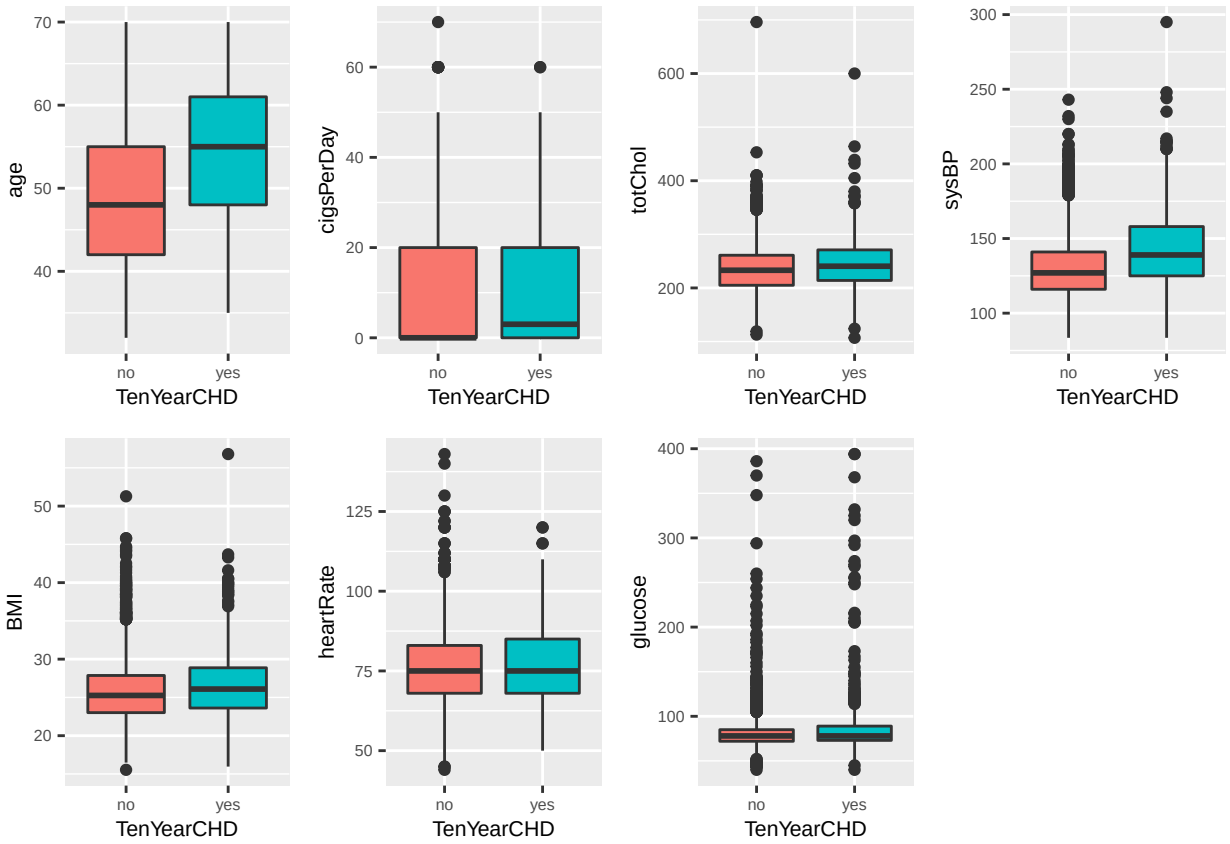


Figure 1: Collection of boxplots for each numeric variable, split on class variable.

As you may have noticed when exploring these plots, we are presented with a lot of outliers as well as some skewed data. The `glucose` variable would be a great example of the former, while the `cigsPerDay` variable is a wonderful illustration of the latter. We'll try to remedy this in our next step.

1.2.2 Boxplots split on class variable - Log transformation

To try and reduce both the outliers in our data as well as the skewness, we will perform a log transformation which we shall then plot in the same way as in the last step. This should, theoretically, help to reduce both of these problems.
When performing this operation we do not just use the `log()` function, but we add one to all of the values first. To explain why we will once again look at the `cigsPerDay` variable. As you might imagine there are a lot of people that simply don't smoke and will therefore have reported zero cigarettes smoked. If we were to perform a log transformation now, R would return infinity for all the zero values. These infinity values would then automatically be filtered out by R when trying to create a plot, therefore simply removing these data points from the visualisation. If we add one to all of our values the distribution stays the same, but it does eliminate this problem of infinite values.

```
# function that creates a boxplot on log of dataset given a column name
boxplotsLogTransform <- function(colName) {
  ggplot(dataset, aes(x = TenYearCHD, y = log(!!sym(colName) + 1), fill = TenYearCHD)) +
  geom_boxplot(show.legend = FALSE) +
  theme(text=element_text(size=8))
}

# execute boxplots function on every numeric column
```

```
plot_list <- lapply(numeric_cols, boxplotsLogTransform)
# arrange these boxplots into one figure
ggarrange(plotlist = plot_list, ncol = 4, nrow = 2)
```

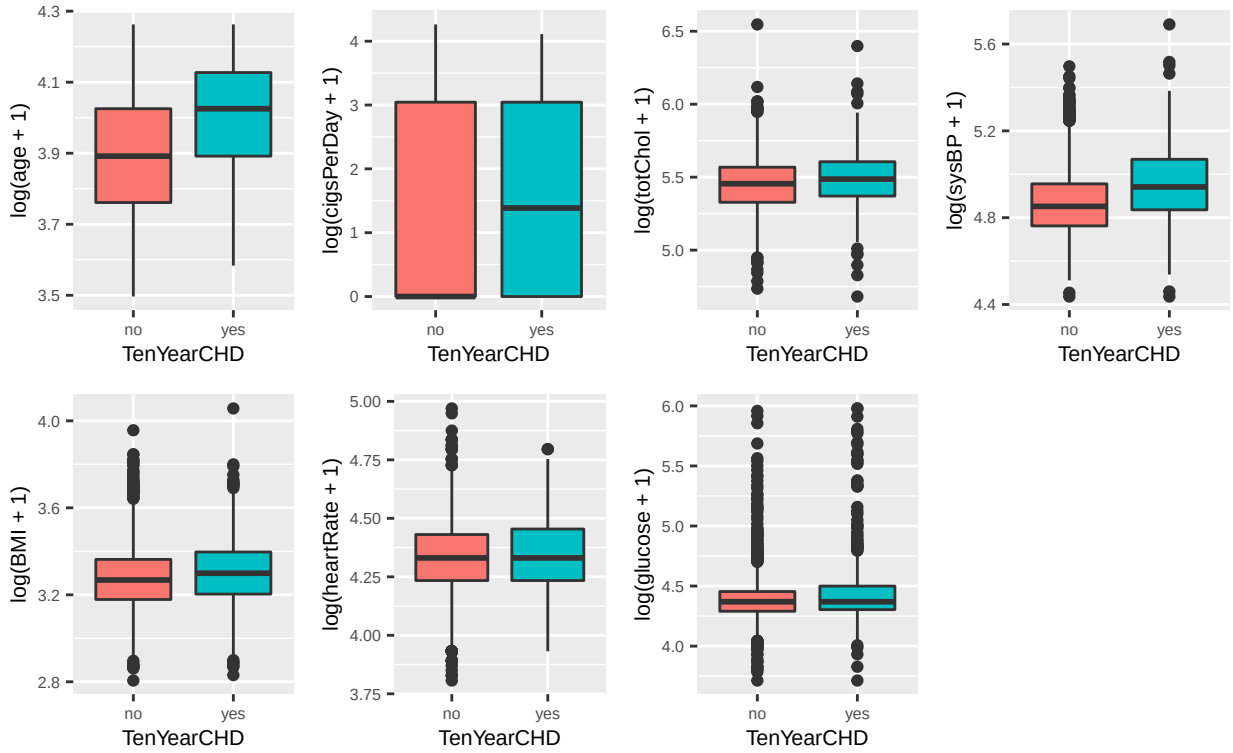


Figure 2: Collection of boxplots for each log transformed numeric variable, split on class variable.

Let's take a look at the same two variables as in our last step, `glucose` and `cigsPerDay`. Examining `glucose`'s boxplot we learn that the boxplot itself has gotten a little larger, indicating fewer outliers. The difference isn't all that noticeable however, and we are still presented with a lot of outliers. Evaluating the `cigsPerDay` variable presents us with a disparity between the two `TenYearCHD` categories. While the `yes` boxplot clearly shows an improvement in its skewness, the `no` boxplot does not.
Overall this transformation doesn't seem to have had too much of an impact. We therefore won't be using it any further, unless it turns out to be preferable when selecting an algorithm later on in this project.

1.2.3 Class distributions

Now that we know about the variations and outliers in our data, it is time to look at it's distribution. At this step we can look at both our numeric as well as our categorical data since were looking at a collection of bar plots with frequencies.
For every one of our variables we'll make one of these bar plots. We can use these to look for two features in particular. First we can quickly glance at the bar plots for our numeric data and speculate whether they seem to be normally distributed. Secondly we may examine all of the bar plots a little more carefully and look for skewness.

```r
# Function that creates a barplot on dataset given a column name
barplots <- function(colName) {
  ggplot(dataset, aes(x = !!sym(colName), fill = !!sym(colName))) +
  geom_bar(show.legend = FALSE) + ylab("") +
  theme(text=element_text(size=8))
}

# Gather column names from entire dataset
cols <- colnames(dataset)
# execute barplots function on every column
plot_list <- lapply(cols, barplots)
# arrange these boxplots into one figure
ggarrange(plotlist = plot_list, ncol = 4, nrow = 4)
```

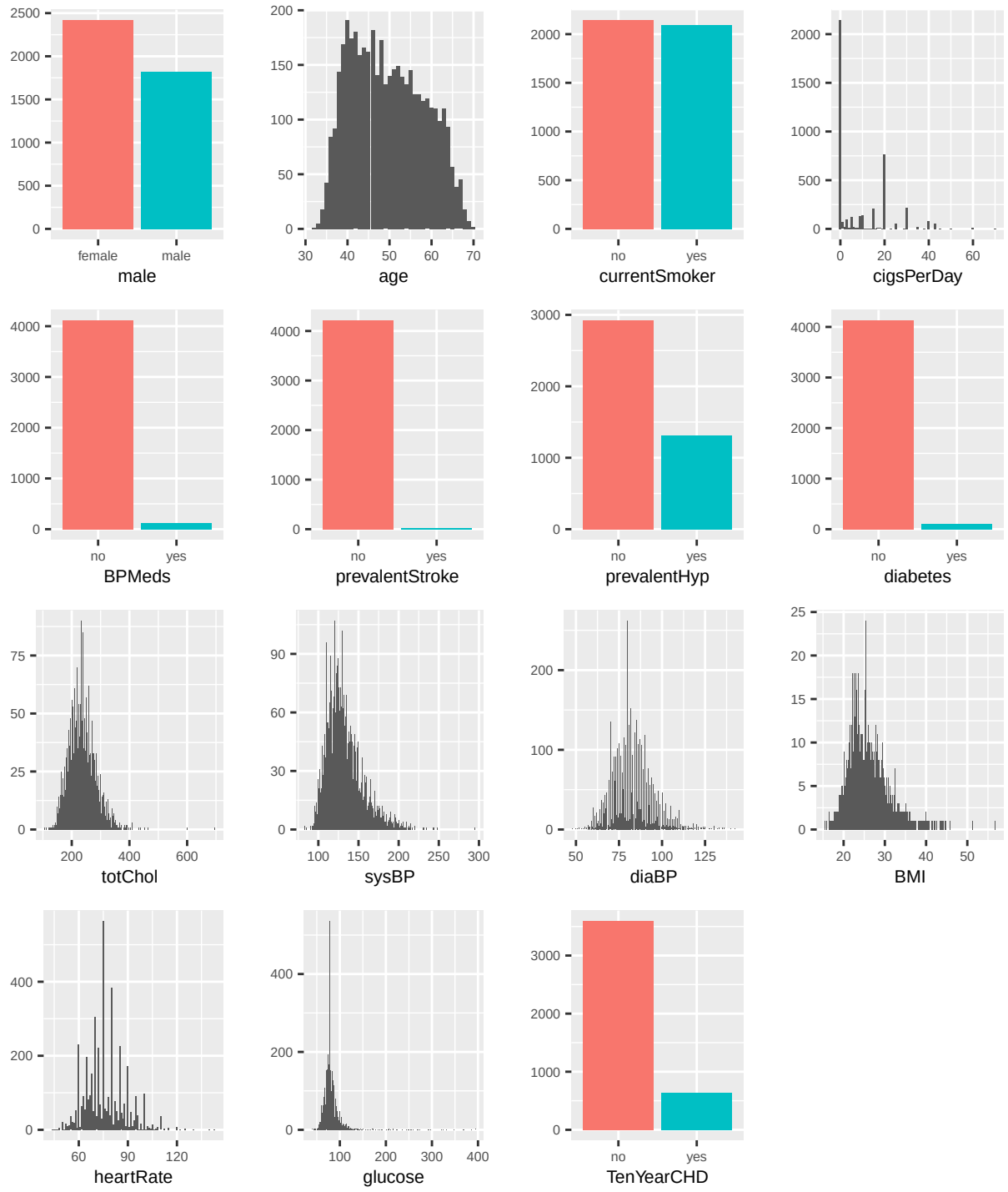


Figure 3: Collection of barplots for every variable, coloured by class variable.

The first thing we notice is that a lot of the categorical variables are very skewed towards one particular value. In fact, all but two of these variables seem to have a significant level of skewness present. Removing most of our variables because of skewness would be foolish, so this is something we'll have to take into account when selecting an algorithm later on. One possible set of techniques to look into would be Synthetic Minority Oversampling Technique or SMOTE for short. What this entails and whether it is a good fit for our needs is something to be looked into later.

Something worth noting about our numeric variables is that all of them look to be normally distributed when examined with the naked eye. This is not of immediate concern but certainly good to know.
In a similar vain, one might notice a small detail that wasn't mentioned in our dataset's source. The age range for this dataset looks to be thirty to seventy year olds. Again, not something that is of immediate concern, but certainly good to know.

1.2.4 Correlation

Many machine learning algorithms assume all attributes are independent. It is therefore important to check whether the variables in our dataset are actually independent, or if they turn out to be correlated. We'll do this by creating a correlation matrix. In this matrix the size and shade of the circle indicate the amount of correlation. A larger circle indicates a stronger correlation, so a smaller circle in turn indicates the opposite. A blue shade demonstrates a positive correlation while a red shade illustrates a negative correlation. Therefore the lighter shades in between indicate the lack of a correlation.

```
# Convert all columns in the dataset to numeric data
dataset.numeric <- sapply(dataset, as.integer)
# Get correlation metrics on the dataset
correlation <- cor(dataset.numeric)
# Create a correlation matrix using the correlation metrics
corrplot(correlation, type = "lower", tl.col = "black", tl.cex = 0.85, tl.offset = 1,
         title = "Correlation matrix", outline = TRUE, mar = c(0, 0, 1, 0), cl.cex = 0.75)
```

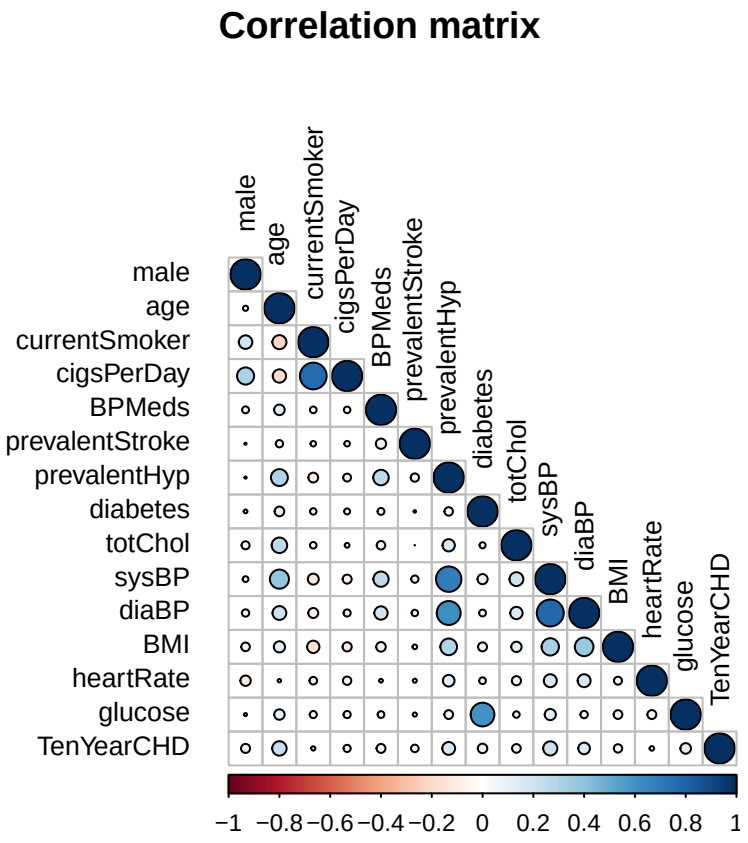


Figure 4: Matrix of degree of correlation between variables.

A good control point indicating the validity of this correlation matrix can be found between diabetes and glucose. This correlation has been studied extensively to the point where it is essentially common knowledge

that high glucose levels correlate with diabetes. Seeing this correlation in our own data is a nice confirmation that this plot is correct. Apart from that, the correlation between these two variables indicates it might be preferable to choose one of them over the other going forward.
Let us now dive into the other correlations. One fairly obvious correlation is the one found between current smokers and the amount of cigarettes smoked per day. Since these variables are so similar and definitely not independent of each other, it might be worth dropping one of them depending on our choice of algorithm later.
The same can be said for systolic blood pressure and diastolic blood pressure. Since the former measures the amount of force put on the arteries as the heart beats, and the latter measures the amount of force on the arteries when the heart is in a state of rest, they might as well be combined into a single blood pressure variable if the chosen algorithm requires it.
Apart from correlating with each other the two variables named above correlate with hypertension as well. This too is a fairly obvious correlation since hypertension denotes the subject has high blood pressure. Another possible solution would therefore be to combine all three of these variables into a single feature if necessary.

1.2.5 PCA - Principal Component Analysis

When trying to determine whether our data clusters together one might be inclined to create a scatter plot. This is a valid approach when trying to compare two different variables. Comparing three different variables would already be slightly more challenging since this would require a three dimensional plot. This would become physically impossible however when trying to compare four or more variables, since our mortal souls are limited to a mere three dimensional space. This is where Principle Component Analysis, or PCA for short, comes in. PCA is a technique where the dimensions (variables) are reduced by converting them into Principle Components. These PCs are ordered by their impact on the variation of the data, where PC1 has the highest impact. The first two PCs will be plotted against each other, where every variable has an arrow indicating it's impact on these first two PCs. Since the plotted PCs have the highest impact on the data's variance, and the biggest arrows have the largest impact on the PCs, larger arrows indicate more significant variables. Furthermore arrows pointing the same way indicate positive correlation, while opposing arrows indicate negative correlation.

```
# Get PCA calculation on numeric dataset
pca <- prcomp(dataset.numeric, center = TRUE, scale. = TRUE)
# Plot PCA calculation
ggbiplot(pca, obs.scale = 1, var.scale = 1,
         groups = dataset$TenYearCHD, circle = TRUE, alpha = 0.35) +
  scale_color_discrete(name = "Risk of CHD") +
  ggtitle("Principal Component Analysis")
```

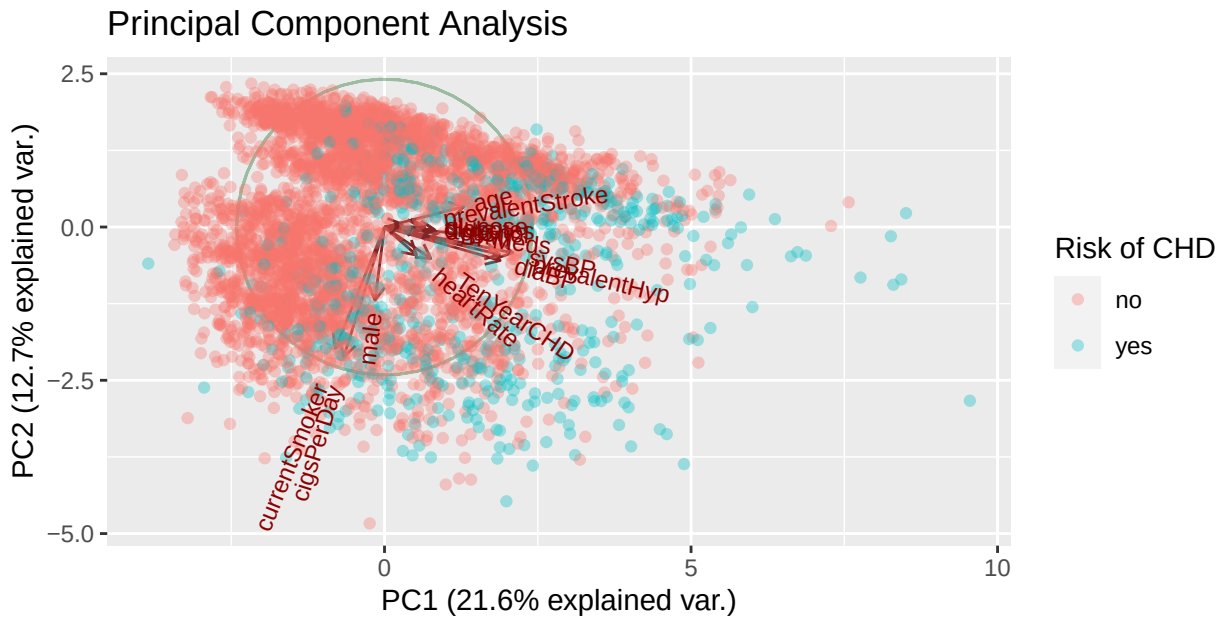


Figure 5: PCA plot charting PC2 against PC1.

In the plot above we can clearly see a contrast between the location of subjects with and without ten year risk of CHD. The subjects with risk of CHD cluster further along the x-axis than subjects lacking this risk. Since PC1 (x-axis) accounts for the larger amount of variation in our data (21.6%) differences along the x-axis are to be considered more significant than differences along the y-axis.

1.3 Exporting data

Now that we have analysed our data and cleaned it accordingly we can go ahead and export it. We'll export into two formats: Comma Separated Values (csv) for easy usage with reporting, and Attribute Relation File Format (ARFF) for usage in the Weka software package. We'll place both of these new data files in the data folder of this project's directory.

```
# To CSV
write.csv(dataset, file = "../data/processedData.csv")
# To ARFF
RWeka::write.arff(dataset, file = "../data/processedData.arff")
```

2 Machine Learning algorithm

Now that we have processed and exported our dataset we can take a big step forward in this project. It is finally time to look at how we shall apply methods of Machine Learning to our data. For easy, visual interaction with our data we will open the previously exported ARFF file in Weka. Once loaded we can start to do all kinds of fun things to our dataset. Before we just start clicking random buttons though, we should discuss our options first.

2.1 Algorithm metrics

Instead of just haphazardly trying out different algorithms, it'd be a better idea to go about this process a bit more systematically. We'll start by listing some important metrics to consider when comparing algorithms and briefly discuss their impact regarding this particular dataset. After this we'll inventorise for which algorithms we'll collect these metrics.

2.1.1 Quality & performance metrics

Weka provides us with loads of metrics which we can combine into performance graphs in whichever way we'd like. Taking in every single one of these measurements would be an overwhelming amount of information, so we'll pick the most important ones and discuss them below.

2.1.1.1 Speed Speed is far from the most important metric when considering the context of our data. After all, this isn't some big social media website needing to fulfill millions of requests at any time of day. It is however still an interesting metric to remember since the difference between testing an instance in a few seconds versus half an hour is still fairly significant.

2.1.1.2 Accuracy Quick? Yes. Reliable? No. Accuracy is simply a measure for which instances an algorithm classified correctly. Seems sensible right? It is right there, front and centre, when running any algorithm and it is easy to grasp. Yet solutions that are the first thing you think of, seem sensible, and are easy to implement are often terrible, ineffective solutions, once implemented will drag on civilisation forever. And so it is for accuracy in the machine learning sphere. Accuracy is often still used as the be all and end all solution for grading an algorithms performance, especially by folk new to the field. It will be included here as a quick and dirty indication, but we will combine it with other, more informative solutions.

2.1.1.3 The confusion matrix This matrix is important to understand since it's the key to understanding the metrics yet to be discussed. The confusion matrix does exactly as it's name suggests. It provides us with an overview of how "confused" our algorithm is, and communicates this in an easy to understand matrix.
In this environment "confused" refers to the instances the algorithm in question labelled improperly. This can be in the form of instances incorrectly labelled as positive, known as the "false positives" (FP). Counter to that, we can have positive instances labelled as negative instead. We call these "false negatives" (FN). Correctly labelled records are intuitively referred to as "true positives" (TP) and "true negatives" (TN). The number of real positive and negative cases in the data are known as "condition positive" (P) and "condition negative" (N) respectively.

An example of a confusion matrix:

Table 6: Example confusion matrix.

		Predicted condition	
		Positive	**Negative**
Actual condition	**Positive**	52 (TP)	8 (FN)
	Negative	4 (FP)	32 (TN)

2.1.1.4 F-measure This metric is a score based on two other metrics, both of which are based on the confusion matrix discussed above. The two parts that make up the F-measure are the precision (also known as positive predictive value (PPV)) and the recall (also known as sensitivity, hit rate, or true positive rate (TPR)). The former represents the fraction of correctly labelled instances out of all instances labelled as positive. In formulaic form it would be described by $PPV = \frac{TP}{TP+FP}$. The latter of our two parts describes the fraction of positive instances which were labelled as so. It's formula turns out as follows: $TPR = \frac{TP}{P} = \frac{TP}{TP+FN}$.
Both of these values can then be combined into the F-measure with the following formula:

$$F = 2 \times \frac{PPV \times TPR}{PPV + TPR} = \frac{2TP}{2TP + FP + FN}$$

It represents the harmonic mean of precision and recall, where an outcome of 1 would indicate both of these values are perfect and an outcome of 0 would indicate both are zero as well.

2.1.1.5 ROC Area Last but not least is the Receiver Operating Characteristic Area. This name may seem a little odd, but that's because electrical engineers and radar engineers came up with it. Like with the F-measure, the ROC Area is another metric that's made up of two other measures, those being the true positive rate (TPR) (also referred to as benefit) and the false positive rate (FPR) (also referred to as cost). These respectively describe the instances correctly and incorrectly labelled as positive. When we plot the cumulative distribution function of these two metrics against each other, with TRP on the y-axis and FPR on the y-axis, we may calculate the area under the curve. An area of 1 corresponds to a most optimal model, an area of exactly 0.5 corresponds to as good as a random guess, and an area below 0.5 corresponds to worse than a random guess.
The latter of these three options seems like an abominable outcome, but it may not be so. If we can *reliably* predict the *wrong* answer, we can simply flip it around to reliably predict the *right* answer. This method will work either way, as long as we're sure enough about the answer's correctness.

2.1.1.6 Metrics overview Now that we know what to look for in an algorithm we can summarise this in a short overview.

Table 7: Overview of algorithm scoring metrics.

Metric	**Description**	**Desired outcome**
Speed	How many seconds it takes to test the model	Lower is better
Accuracy	Fraction of correctly labelled instances	Higher is better
False Negative Rate	Fraction of incorrectly labelled positive instances	Lower is better
F-measure	Harmonic mean of precision and recall	Higher is better
ROC Area	Area under ROC curve where 0.5 equals a random guess	Deviation from 0.5 is better

2.1.2 Algorithm exploration

We know what to test for, but what do we actually want to test? Well, machine learning algorithms of course, but which ones? We'll pick a few algorithms from each of the main categories:
As a start we'll pick ZeroR as a baseline, and then add OneR as a slightly more complex addition in the rules division.
In the Trees section we'll pick J48 since it can handle both numeric and nominal data and random forest will be generated as well.
In the lazy learning department we'll go with IBk, otherwise known as k-nearest neighbours.
From Bayes we shall add Naive Bayes to our list of algorithms.
Last but not least from the Rules category we'll pick Simple Logistic and Sequential Minimal Optimisation (SMO).
All tables containing test results were generated by the Weka experimenter after which they were (partially) combined and edited stylistically. A white dot represents a statistically significant higher result than ZeroR, while a black dot represents a statistically significant lower result than ZeroR.

2.1.2.1 Default settings testing To start we'll run all of these algorithms with their default settings, using ZeroR as the baseline, and performing 10-fold cross validation on the dataset.

Table 8: References to ML algorithms and their (default) settings.

(1)	rules.ZeroR ''
(2)	rules.OneR '-B 6'
(3)	trees.J48 '-C 0.25 -M 2'
(4)	trees.RandomForest '-P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1'
(5)	lazy.IBk '-K 1 -W 0 -A \"weka.core.neighboursearch.LinearNNSearch -A \\\"weka.core.EuclideanDistance -R first-last\\\"
(6)	bayes.NaiveBayes ''
(7)	functions.SimpleLogistic '-I 0 -M 500 -H 50 -W 0.0'
(8)	functions.SMO '-C 1.0 -L 0.001 -P 1.0E-12 -N 0 -V -1 -W 1 -K \"functions.supportVector.PolyKernel -E 1.0 -C 250007\" -calibrator \"functions.Lo

Table 9: Comparison of ML algorithms using default settings using ZeroR as a baseline.

Algorithm	*ZeroR*	*OneR*	*J48*	*RandomForest*	*IBk*	*NaiveBayes*	*SimpleLogistic*	*SMO*
Elapsed time testing	0.00	0.00	0.00	0.01 ○	0.04 ○	0.00 ○	0.00	0.00
Accuracy	84.80	84.41 ●	83.98 ●	84.92	78.52 ●	81.79 ●	85.43 ○	84.79
False Negative Rate	0.00	0.01 ○	0.03 ○	0.01 ○	0.12 ○	0.08 ○	0.01 ○	0.00
F-measure	0.92	0.92 ●	0.91 ●	0.92	0.87 ●	0.90 ●	0.92 ○	0.92
ROC Area	0.50	0.51 ○	0.64 ○	0.69 ○	0.56 ○	0.71 ○	0.73 ○	0.50

○, ● statistically significant increase or decrease

The thing to notice in these results is that with both the Accuracy as well as the F-measure metric, we get performances which are statistically significantly worse than ZeroR, our baseline algorithm. Even though the ROC Area looks fine, this result isn't any good. This occurance can be explained by our skewed data from the chapter on Class distributions, in particular our skewed class variable. Since 84.8% of our TenYearCHD variable is classified as `no` even ZeroR will get an accuracy of 84.8%, even though this algorithm isn't any good. We will fix this issue with the method we mentioned before; SMOTE.

2.1.2.2 SMOTE With this technique we will artificially create extra instances of the `yes` kind of our class variable based on other values in the dataset. Luckily we can simply do this in Weka. First we open our existing dataset in the Weka explorer after which we will select the `SMOTE` filter under `filters > supervised > instance`. We then tune the parameters as follows:
`classValue` = 0
`nearestNeighbors` = 5
`percentage` = 450
This means we select the right instances to increase, base new instances on it's 5 nearest neighbors, and increase the number of instances by 450% therefore balancing the `yes` and `no` labels.
We have now added enough synthetic instances to balance our classes, but SMOTE introduces a new problem.

When creating new instances it will produce values with a tremendous amount of decimal places. If we were to just use these new values as is, we'd get a huge bias towards these synthetic instances, in the process probably turning OneR into the most ideal algorithm. To remedy this we will round the values in our new post-SMOTE dataset to the same number of decimals as the pre-SMOTE dataset. To do this we will apply another filter: `filters > unsupervised > attribute > NumericCleaner`, which we will apply using default settings except for the `decimals` parameter. We will set this to whichever number of decimals the attribute initially had. Once we have done this we can then save this dataset and return to the Weka experimenter. After completing all these steps we get the following results:

Table 10: Comparison of ML algorithms using default settings after performing SMOTE on the dataset, using ZeroR as a baseline.

Algorithm	*ZeroR*	*OneR*	*J48*	*RandomForest*	*IBk*	*NaiveBayes*	*SimpleLogistic*	*SMO*
Elapsed time testing	0.00	0.00	0.00	0.03 ∘	0.12 ∘	0.00 ∘	0.00	0.00
Accuracy	50.36	80.28 ∘	74.83 ∘	84.23 ∘	77.06 ∘	62.33 ∘	67.36 ∘	67.17 ∘
False Negative Rate	0.00	0.09 ∘	0.26 ∘	0.17 ∘	0.31 ∘	0.22 ∘	0.33 ∘	0.36 ∘
F-measure	0.67	0.82 ∘	0.75 ∘	0.84 ∘	0.75 ∘	0.67	0.67	0.66
ROC Area	0.50	0.80 ∘	0.77 ∘	0.92 ∘	0.77 ∘	0.70 ∘	0.73 ∘	0.67 ∘

∘, • statistically significant increase or decrease

We now see that ZeroR doesn't perform nearly as well as it did before, and most of the other algorithms are already significantly better. This is a much better result and more accurately reflects what initial results should look like.
Now that we have some decent results we can take a look at which algorithms actually perform well. When taking each metric into account we see that both the J48 tree as well as RandomForest perform quite well, but they are not quite perfect. They have a false negative rate of 26% and 17% respectively. Although this is quite a bit lower than most of the other algorithms, it is still quite dubious to use an algorithm which classifies $\frac{1}{10}$ of positive instances as negative.

2.1.2.3 Cost Sensitive Classifier To remedy the issue of false negatives we can apply a technique called cost sensitive classification. This classifier allows one to apply a custom amount of cost to values in the confusion matrix. In our case false negatives are a great sin, so we'll try doubling its cost two as compared to false positive values' cost of one. We can do this in Weka by wrapping all our algorithm choices in a cost sensitive classifier: Instead of immediately choosing an algorithm we'll go `meta > CostSensitiveClassifier` and then enter an algorithm of choice at the `classifier` option. We can then proceed to enter our 2x2 `costMatrix` and set `minimizeExpectedCost` to `True`. After doing this for each algorithm we get the following results.

Table 11: Comparison of ML algorithms using default settings and cost sensitive classification after performing SMOTE on the dataset, using ZeroR as a baseline.

Algorithm	*ZeroR*	*OneR*	*J48*	*RandomForest*	*IBk*	*NaiveBayes*	*SimpleLogistic*	*SMO*
Elapsed time testing	0.00	0.00	0.00	0.03 ∘	0.12 ∘	0.00 ∘	0.00	0.00
Accuracy	50.36	80.28 ∘	74.65 ∘	80.29 ∘	77.06 ∘	61.80 ∘	61.95 ∘	67.17 ∘
False Negative Rate	0.00	0.09 ∘	0.23 ∘	0.04 ∘	0.31 ∘	0.15 ∘	0.12 ∘	0.36 ∘
F-measure	0.67	0.82 ∘	0.75 ∘	0.83 ∘	0.75 ∘	0.69 ∘	0.70 ∘	0.66
ROC Area	0.50	0.80 ∘	0.75 ∘	0.80 ∘	0.77 ∘	0.62 ∘	0.62 ∘	0.67 ∘

∘, • statistically significant increase or decrease

In these results we can see that some of our best performing algorithms, including our best: the RandomForest, take a hit in accuracy and ROC Area. However, we do get a significantly lower false negative rate in return for that hit. The two best performing algorithms that we are left with look to be OneR and RandomForest. We'll continue our analysis with just these two, since optimising every algorithms would be too time intensive for this project.

2.1.3 Algorithm optimisation

Now that we have found our best performing algorithms, we can continue on to optimising their settings as much as possible. To do this we'll once again use the Weka experimenter. In contrast to the last chapter we will stop using ZeroR as a baseline at this point. Instead we'll be using the default settings setup of each algorithm as a baseline for optimising that particular algorithm.

2.1.3.1 OneR For our convenience OneR only has a single setting that really interests us, which is its minimum bucket size.

2.1.3.1.1 Minimal bucket size In Weka the default value for this setting is six which we will therefore use as a baseline in this analysis. Since we have a fairly large dataset we can afford to increase this value dramatically as we do not want to end up with a nonsensical model. When OneR selects its preferred attribute it will start to classify instances based on that attribute. When the bucket size is too low it might create some weird models. For example:
Let's say OneR picked, for instance, age as the attribute it wants to use. When the bucket size is too low it might classify an instance with an age of 38 as `yes`, an instance with an age of 39 as `no`, and an instance with an age of 40 as `yes` again. This is a bit odd, and to prevent this we will increase the bucket size.

Table 12: Comparison of ZeroR using varying bucket sizes while using cost sensitive classification after performing SMOTE on the dataset, using ZeroR's default settings as a baseline.

OneR setting	*-B 6*	*-B 25*	*-B 50*	*-B 100*	*-B 200*
Elapsed time testing	0.00	0.00	0.00	0.00	0.00
Accuracy	80.28	74.19 •	64.39 •	64.89 •	64.90 •
False Negative Rate	0.09	0.23 ○	0.43 ○	0.47 ○	0.46 ○
F-measure	0.82	0.75 •	0.62 •	0.61 •	0.61 •
ROC Area	0.80	0.74 •	0.64 •	0.65 •	0.65 •

○, • statistically significant increase or decrease

Unsurprisingly, as we make our model more sane by increasing the bucket size, the model's performance drops significantly in every single measure, except of course for elapsed time. This is a very clear indication that OneR is probably not the algorithm we'd like to continue on with. Luckily we spread our eggs over two baskets, so let us now take a look at the Random Forest.

2.1.3.2 RandomForest This algorithm has a bunch more settings than OneR, as it is a bit more complicated. There is however only one setting that we are really interested in here, which is `numIterations`.

2.1.3.2.1 Number of iterations The number of iterations we use when producing our forest represents the number of trees in our forest. Theoretically we should get a better result with more iterations since producing more trees increases the chance we'll find a better one.

Table 13: Comparison of RandomForest using a varying number of iterations while using cost sensitive classification after performing SMOTE on the dataset, using RandomForest's default settings as a baseline.

RandomForest setting	*-I 100*	*-I 250*	*-I 500*	*-I 750*	*-I 1000*
Elapsed time testing	0.03	0.14 ○	0.36 ○	8.17	0.89 ○
Accuracy	80.29	80.51	80.53	80.51	80.55
False Negative Rate	0.04	0.04	0.04	0.04	0.04
F-measure	0.83	0.83	0.83	0.83	0.83
ROC Area	0.80	0.80	0.80	0.80	0.80

○, • statistically significant increase or decrease

Although we do see some changes in our accuracy values, none of them are statistically significant. We do however see significant changes in our elapsed time testing, but these increases are actually a negative consequence. All this means that we can stick with a Random Forest doing 100 iterations.

2.1.4 Meta-learners

We now have the most optimal options for an algorithm. We can however still try too improve this algorithm by using so called meta-learners. These meta algorithms take an existing base algorithm and add something extra on top. In this chapter we will explore both bagging as well as boosting.

2.1.4.1 Bagging Bagging is a meta-learner, or ensemble learning, is a technique where we sample the original dataset to create sets of the same size. We then build a model (in this case based on our RandomForest) for each of these sets. The outcomes for each prediction are then aggregated into a single prediction by, in this case, voting. This method helps to decrease variance, while increasing accuracy. We will perform this technique with four different numbers of iterations and compare it to a baseline without bagging.

Table 14: Comparison of bagging with RandomForests using a varying number of bagging iterations while using cost sensitive classification after performing SMOTE on the dataset, using a RandomForest without bagging as a baseline.

Bagging setting	*no bagging*	*-I 10*	*-I 25*	*-I 40*	*-I 55*
Elapsed time testing	0.03	0.43 ○	11.06	2.42 ○	4.91
Accuracy	80.29	78.37 ●	79.28 ●	79.19 ●	79.37 ●
False Negative Rate	0.04	0.04 ●	0.04	0.04	0.04
F-measure	0.83	0.82 ●	0.82 ●	0.82 ●	0.82 ●
ROC Area	0.80	0.88 ○	0.89 ○	0.90 ○	0.90 ○

○, ● statistically significant increase or decrease

This analysis has led us to some very interesting results. While the accuracy has taken a small, but statistically significant, hit across the board, ROC Area shows some major improvements. Elapsed time testing has seen some significant increases, with the value for twenty-five bagging iterations being particularly odd. While some other measures show statistically significant differences, they do stay fairly similar. In the interest of keeping model training times at a reasonable level, while also squeezing out the maximum amount of performance, the model created with twenty-five iterations seems to be the best this analysis has to offer.

2.1.4.2 Boosting Boosting is another ensemble learner which builds multiple models. This is an iterative technique which builds new models based on the performance of the previously built models. It does this by assigning extra weight to instances which were misclassified before. This encourages the new model to become a so called expert at instances that were misclassified by earlier models. This algorithm is known in Weka as `AdaBoostM1`. We will once again perform this technique with four different numbers of iterations and compare it to a baseline without boosting.

Table 15: Comparison of boosting with RandomForests using a varying number of boost iterations while using cost sensitive classification after performing SMOTE on the dataset, using a RandomForest without boost as a baseline.

Boost setting	*no boost*	*-I 10*	*-I 25*	*-I 40*	*-I 55*
Elapsed time testing	0.03	0.42 ○	1.25 ○	2.49 ○	3.65 ○
Accuracy	80.29	85.25 ○	88.02 ○	88.84 ○	89.28 ○
False Negative Rate	0.04	0.05 ○	0.06 ○	0.06 ○	0.06 ○
F-measure	0.83	0.87 ○	0.89 ○	0.89 ○	0.90 ○
ROC Area	0.80	0.94 ○	0.94 ○	0.93 ○	0.93 ○

○, ● statistically significant increase or decrease

Boosting seems to have led us to a treasure trove! We see statistically significant increases across the board. This is unfortunately at the cost of a false negative rate which is two percentage points higher. One could however argue that this is a sacrifice worth making since we do get significantly higher values for accuracy, ROC area, and even F-measure. Although training the algorithms with 55 boosting iterations takes some time, it seems to be well worth the cost.

2.1.5 Final tuning

At this point we have a pretty good model which is great! There's always more we can do though, so let us experiment with some final touches to see if we can make our pretty good model just that much better.

2.1.5.1 Reevaluating cost matrix Since our accuracy score and ROC Area are looking pretty strong we can examine whether it's worth it to increase the cost for false negatives in our `CostSensitiveClassifier` just a bit more. We'll use our best model thus far but we will up the false negative cost from 2 to 2.5. This leaves us with the following results:

Table 16: Comparison of boosting with RandomForests using a varying number of boost iterations while using cost sensitive classification after performing SMOTE on the dataset, using a RandomForest without boost as a baseline.

FN Cost	*2.0*	*2.5*	
Elapsed time testing	6.94	22.86	
Accuracy	89.28	88.88	
False Negative Rate	0.06	0.05	•
F-measure	0.90	0.90	
ROC Area	0.93	0.93	

○, • statistically significant increase or decrease

We see two important changes: a slight decrease in accuracy which is not statistically significant, and a one percentage point decrease in our false negative rate which is statistically significant. We will therefore want to keep this small, but useful, change around.

2.1.5.2 Attribute selection As a last attempt to improve our model we will try an attribute selector. This will only keep around the attributes it thinks are actually useful when building and testing our model. We will make use of Weka's `AttributeSelectedClassifier` for this where the evaluator is `CfsSubsetEval` and search is `ExhaustiveSearch`. This presents us with the foloowing results:

Table 17: Comparison of boosting with RandomForests using a varying number of boost iterations while using cost sensitive classification after performing SMOTE on the dataset, using a RandomForest without boost as a baseline.

Attribute selector	*No selector*	*Selector*
Elapsed time testing	4.49	4.76
Accuracy	88.88	77.73 •
False Negative Rate	0.05	0.13 ○
F-measure	0.90	0.80 •
ROC Area	0.93	0.82 •

○, • statistically significant increase or decrease

This approach has worsened our results significantly. Metrics dropped across the board, making this method not worthwhile to pursue.

2.1.6 Final model

We have finally reached the last stop in our machine learning model building adventures. Let's take a brief look at what it is we ended up with. When entering our model settings into Weka we start off with `AdaBoostM1` for which we set `numIterations` to 55. We'll then want to choose a classifier and for that we will use the `CostSensitiveClassifier`. For this will resize the `costMatrix` to 2x2, set the cost of false negatives to 2.5, and set `minimizeExpectedCost` to True. Last but not least we pass a classifier to this algorithm as well, and choose the RandomForest which we will use with just the `numExecutionSlots` parameter set to 8 for full CPU utilisation. This leaves us with a final model which performs as follows:

Table 18: Final model which utilises boosting with RandomForests while using cost sensitive classification after performing SMOTE on the dataset.

Final Model	
Accuracy	89.209
False Negative Rate	0.05
F-measure	0.892
ROC Area	0.942

This model can be found at the following location in the project directory:
`/data/BoostedCostSensitiveRandomForest.model`

2.2 Java wrapper