Predicting survival and need for surgical intervention in horses with colic using machine learning

KatiKirsch

7 7 2021

1. Introduction

The Horse-Colic Dataset from the UCI Machine Learning Repository consists of a large number of different clinical parameters of horses suffering from colic as well as the information whether, retrospectively, the problem was surgical (needed surgery) or non-surgical (should have been treated conservatively) and the eventual outcome for the horse (lived, died, euthanized).

Colic in horses is defined as abdominal pain that can have numerous causes, some of which can have severe consequences or even be fatal without surgical intervention. Therefore it is crucial to decide, based on the characteristics of the clinical symptoms, whether a horse should undergo surgery or not. On the other hand from an animal welfare perspective, it might not be justifiable to subject a horse to the additional suffering accompanying surgery if it has a very little chance to survive.

In the following, the Horse-Colic data set will be used to create a machine learning algorithm that predicts whether the horse will survive or not based on clinical symptoms.

A second algorithm will be created to predict whether the horse should be treated surgically or not. For this algorithm, the `surgical_lesion` variable will be used as outcome variable. (The variable `surgical_lesion` indicates whether (retrospectively) the lesion was surgical or not)

2. Analysis

2.1 Data exploration and preparation

2.1.1 Downloading, cleaning and exploring the data

The data will be downloaded from the UCI Machine Learning Repository. The variable names and characteristics are specified in the Data Set Information.

```
col_names <- c("surgery", "age", "hospital_number", "rectal_temp", "pulse",
               "respiratory_rate", "temp_of_extremeties", "peripheral_pulse",
               "mucous_membranes", "capillary_refill_time", "pain",
               "peristalsis", "abdominal_distension", "nasogastric_tube",
               "nasogastric_reflux", "nasogastric_reflux_pH",
               "rectal_exam_feces", "abdomen", "packed_cell_volume",
               "total_protein", "abdominocent_appearance",
               "abdominocent_protein", "outcome", "surgical_lesion",
               "lesion_1", "lesion_2", "lesion_3", "cp_data")
colic.train <- read.table(url(
```

```
  "http://archive.ics.uci.edu/ml/machine-learning-databases/horse-colic/horse-colic.data"),
                         header = FALSE, col.names = col_names)
colic.test <- read.table(url(
  "http://archive.ics.uci.edu/ml/machine-learning-databases/horse-colic/horse-colic.test"),
                        header = FALSE, col.names = col_names)
data <- rbind(colic.train, colic.test)
str(data, vec.len = 2)
```

```
## 'data.frame':    368 obs. of  28 variables:
##  $ surgery                : chr  "2" "1" ...
##  $ age                    : int  1 1 1 9 1 ...
##  $ hospital_number        : int  530101 534817 530334 5290409 530255 ...
##  $ rectal_temp            : chr  "38.50" "39.2" ...
##  $ pulse                  : chr  "66" "88" ...
##  $ respiratory_rate       : chr  "28" "20" ...
##  $ temp_of_extremeties    : chr  "3" "?" ...
##  $ peripheral_pulse       : chr  "3" "?" ...
##  $ mucous_membranes       : chr  "?" "4" ...
##  $ capillary_refill_time  : chr  "2" "1" ...
##  $ pain                   : chr  "5" "3" ...
##  $ peristalsis            : chr  "4" "4" ...
##  $ abdominal_distension   : chr  "4" "2" ...
##  $ nasogastric_tube       : chr  "?" "?" ...
##  $ nasogastric_reflux     : chr  "?" "?" ...
##  $ nasogastric_reflux_pH  : chr  "?" "?" ...
##  $ rectal_exam_feces      : chr  "3" "4" ...
##  $ abdomen                : chr  "5" "2" ...
##  $ packed_cell_volume     : chr  "45.00" "50" ...
##  $ total_protein          : chr  "8.40" "85" ...
##  $ abdominocent_appearance: chr  "?" "2" ...
##  $ abdominocent_protein   : chr  "?" "2" ...
##  $ outcome                : chr  "2" "3" ...
##  $ surgical_lesion        : int  2 2 2 1 2 ...
##  $ lesion_1               : int  11300 2208 0 2208 4300 ...
##  $ lesion_2               : int  0 0 0 0 0 ...
##  $ lesion_3               : int  0 0 0 0 0 ...
##  $ cp_data                : int  2 2 1 1 2 ...
```

There are 368 observations and 28 variables, some of them are categorical, others are continuous. Missing values are indicated by `?`, we will convert them to `NA`.

```
if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
library(tidyverse)
data <- data %>% mutate_all(~replace(., .=="?", NA))
```

The numeric variables `rectal_temp`, `pulse`, `respiratory_rate`, `nasogastric_reflux_pH`, `packed_cell_volume`, `total_protein` and `abdominocent_protein` are stored as characters, we will convert them to numeric values.

```
data <- data %>%
  mutate_at(vars(rectal_temp, pulse, respiratory_rate,
                 nasogastric_reflux_pH, packed_cell_volume, total_protein,
                 abdominocent_protein), as.numeric)
```

The remaining categorical variables will be converted to factors based on the Data Set Information.

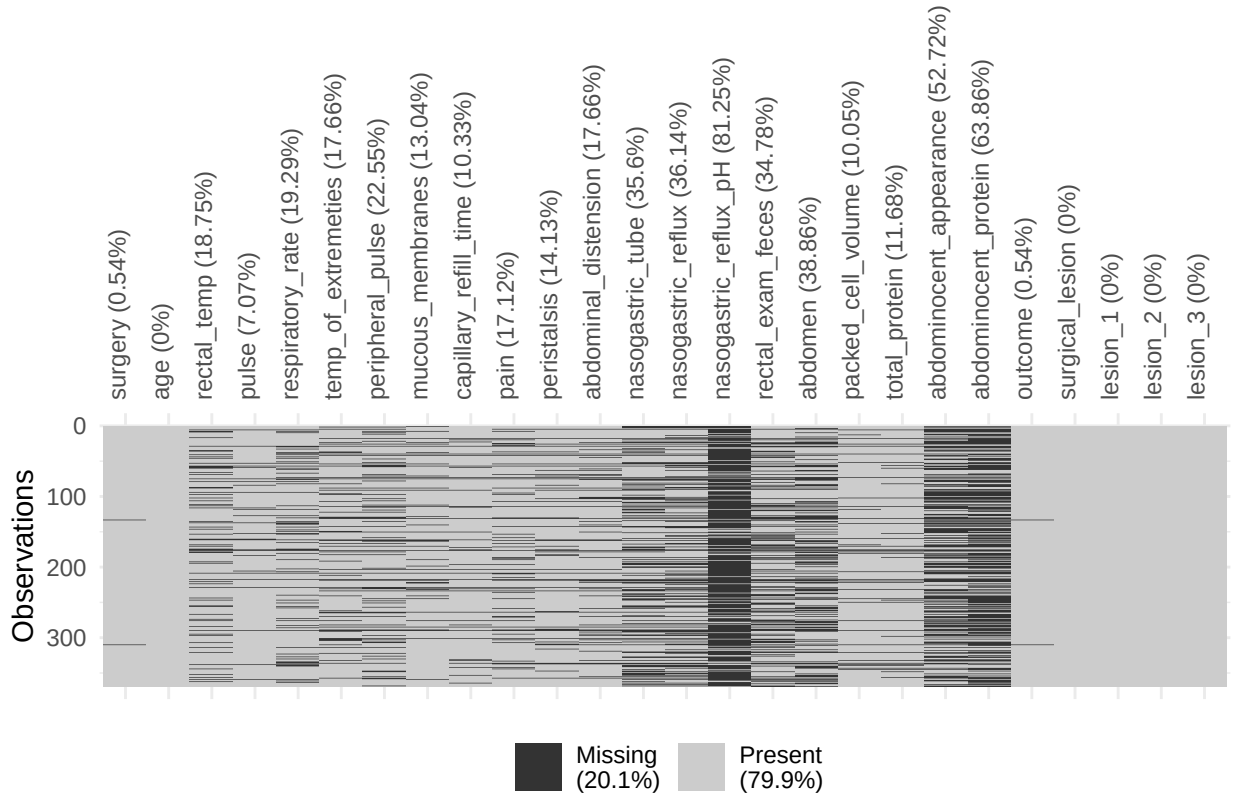
```
data <- data %>%
  mutate(surgery = factor(surgery, labels = c("yes", "no")),
         age = factor(age, labels = c("adult", "young")),
         temp_of_extremeties = factor(temp_of_extremeties,
                                      labels = c("normal", "warm", "cool", "cold")),
         peripheral_pulse = factor(peripheral_pulse,
                                   labels = c("normal", "increased", "reduced", "absent")),
         mucous_membranes = factor(mucous_membranes,
                                   labels = c("normal_pink", "bright_pink", "pale_pink",
                                              "pale_cyanotic", "bright_red", "dark_cyanotic")),
         capillary_refill_time = factor(capillary_refill_time,
                                        labels = c("<3_sec", ">3_sec", "3_sec")),
         pain = factor(pain,
                       labels = c("alert", "depressed", "mild", "severe", "extreme")),
         peristalsis = factor(peristalsis,
                              labels = c("hypermotile", "normal", "hypomotile", "absent")),
         abdominal_distension = factor(abdominal_distension,
                                       labels = c("none", "slight", "moderate", "severe")),
         nasogastric_tube = factor(nasogastric_tube,
                                   labels = c("none", "slight", "significant")),
         nasogastric_reflux = factor(nasogastric_reflux,
                                     labels = c("none", ">1L", "<1L")),
         rectal_exam_feces = factor(rectal_exam_feces,
                                    labels = c("normal", "increased", "decreased", "absent")),
         abdomen = factor(abdomen,
                          labels = c("normal", "other", "firm", "small", "large")),
         abdominocent_appearance = factor(abdominocent_appearance,
                                          labels = c("clear", "cloudy", "serosanguinous")),
         outcome = factor(outcome,
                          labels = c("lived", "died", "euthanized")),
         surgical_lesion = factor(surgical_lesion,
                                  labels = c("yes", "no")))
```

The variable `cp_data` (pathology data present or not) is of no significance since pathology data is not included or collected for these cases, therefore it will be removed. The `hospital_number` will be removed as well since it is just a case identifier and isn't helpful for classification (NOTE: According to the documentation, duplicate hospital numbers indicate that the same horse was treated more than once. In this case, this variable could have predictive value since a horse having recurrent problems with colic may have another chance of survival than a horse having this condition for the first time. However, duplicate hospital numbers are rare in the data set and a unique number does not mean that the horse had this condition for the first time since it may have been treated in another hospital).

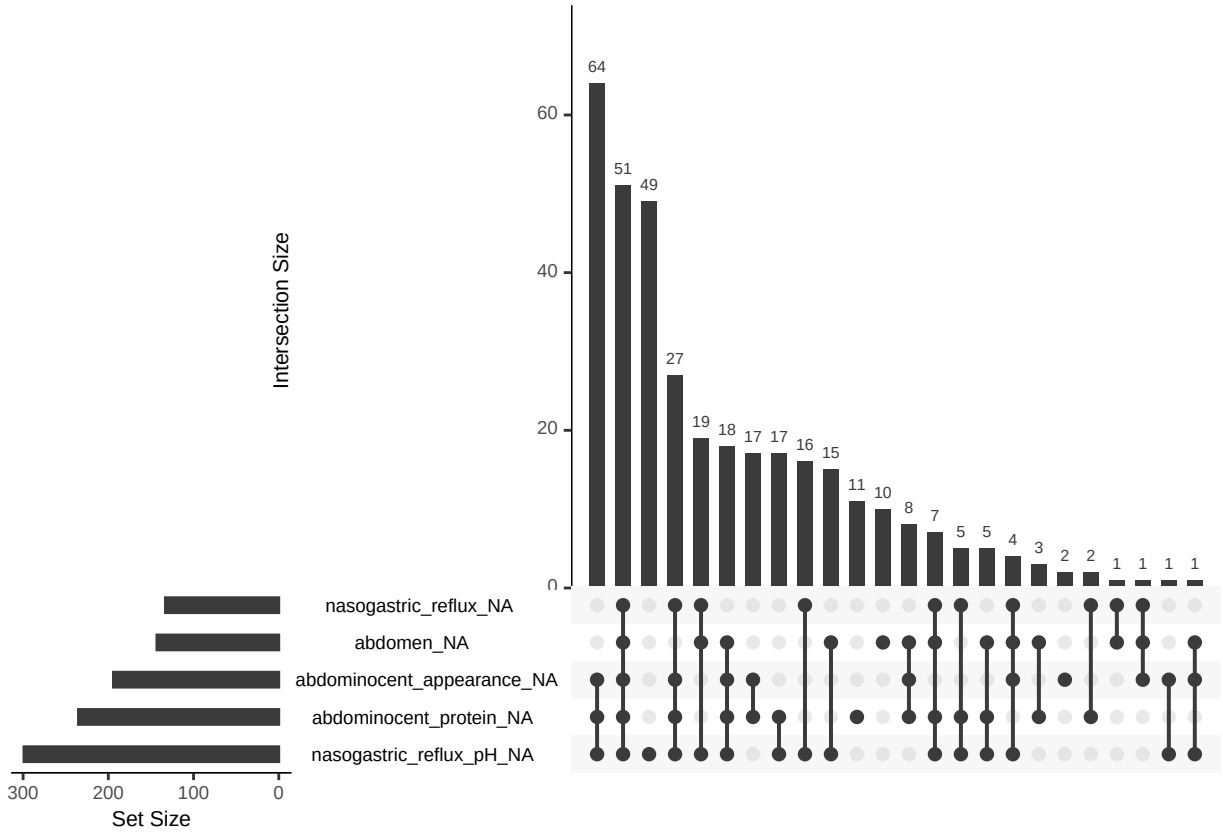
```
data <- data %>% select(-cp_data, -hospital_number)
```

At first glance, we saw that there are missing values in the data. Let's explore that further.

```
if(!require(naniar)) install.packages("naniar", repos = "http://cran.us.r-project.org")
library(naniar)
vis_miss(data) + theme(axis.text.x = element_text(angle = 90))
```




```
if(!require(UpSetR)) install.packages("UpSetR", repos = "http://cran.us.r-project.org")
library(UpSetR)
gg_miss_upset(data)
```



There is a large number of missing values in some features. One of the main tasks before training the machine learning algorithm will therefore be to deal with missing data. Since the data set contains relatively few observations (368), complete case analysis is not a good option. We want to avoid removing observations with missing data in order to retain sufficient observations to train and test the algorithm. Therefore we will rather remove features with too much missing data and impute missing values for the remaining features.

Features that contain large numbers of missing values (above 50%) will be removed from the data set (`nasogastric_reflux_pH`, `abdomo_appearance` and `abdomo_protein`).

```
data <- data %>% select_if(~mean(is.na(.)) < 0.5)
```

Additionally, there are two observations with missing values for the outcome variables, these observations will be removed from the data set as well.

```
data <- data %>% filter(!is.na(outcome) & !is.na(surgical_lesion))
```

After the first cleaning steps and removing features with high proportions of missing data, we will now take a closer look at the `outcome` variable. The `outcome` is classifying whether the horses survived, died or were euthanized. We will now check whether the classes are balanced or not. Ideally, the data set will have an even split of each class.

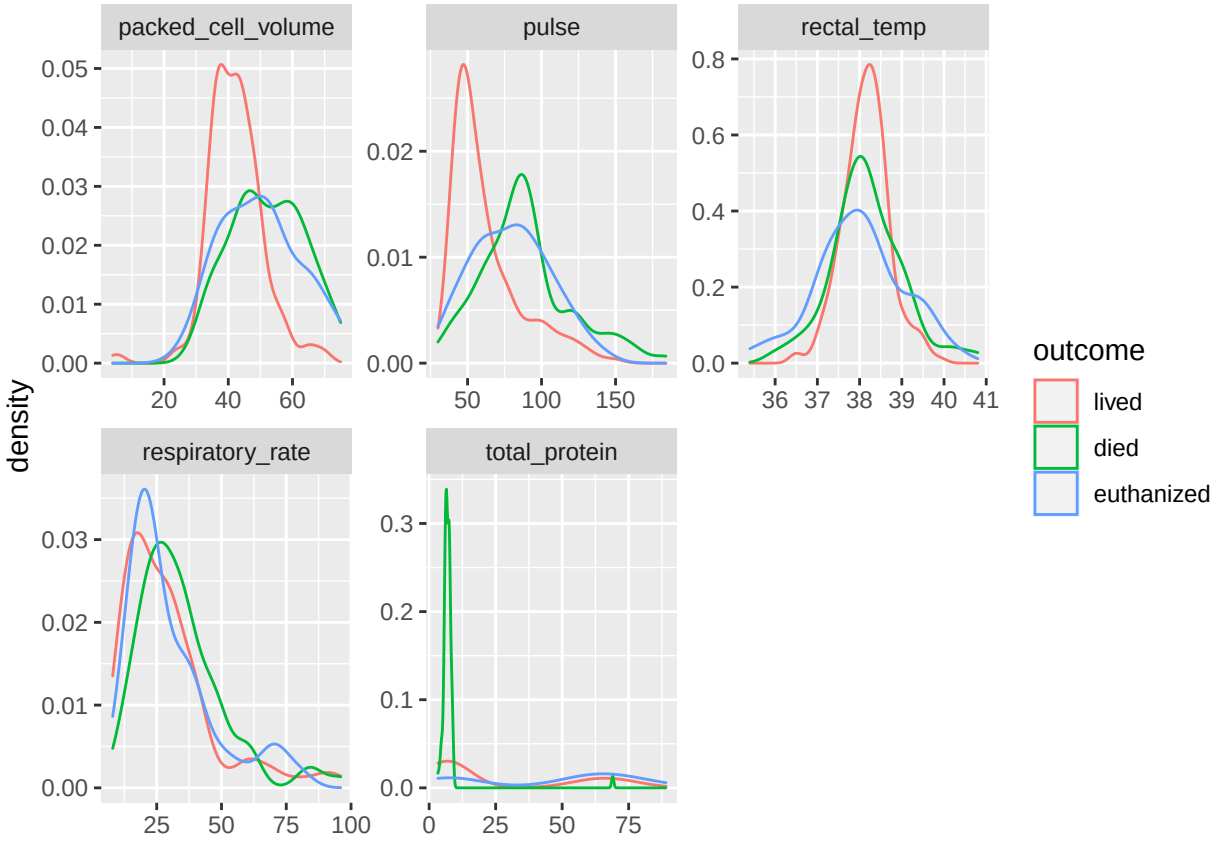
```
data %>% group_by(outcome) %>%
  summarize(n = n(), proportion = n()/nrow(data))
```

```
## # A tibble: 3 x 3
```

```
##   outcome        n proportion
##   <fct>      <int>      <dbl>
## 1 lived        225      0.615
## 2 died          89      0.243
## 3 euthanized    52      0.142
```

About 61% of the horses survived, while about 24% died and 14% were euthanized, so the classes are relatively unbalanced. For the purpose of predicting whether the horse will survive or not, it doesn't matter whether the horse died or was euthanized as the horse must have had a very small or no chance of surviving if it was euthanized and the outcome is the same. We will take a look at how the different features are distributed across horses that survived, died or were euthanized.

```
data %>% select(which(sapply(.,class) == "numeric"),outcome) %>%
  gather(key = "feature", value = "value", -outcome) %>%
  ggplot(aes(value, color = outcome)) +
  geom_density() +
  facet_wrap(~feature, scales = "free") +
  theme(axis.title.x = element_blank())
```



We see, that most features are similarly distributed in horses that died and those that were euthanized, confirming our assumption that based on the given features, it would be very hard to distinguish between horses that died and those that were euthanized. Therefore the `outcome` variable will be converted into a binary variable (survived or died).

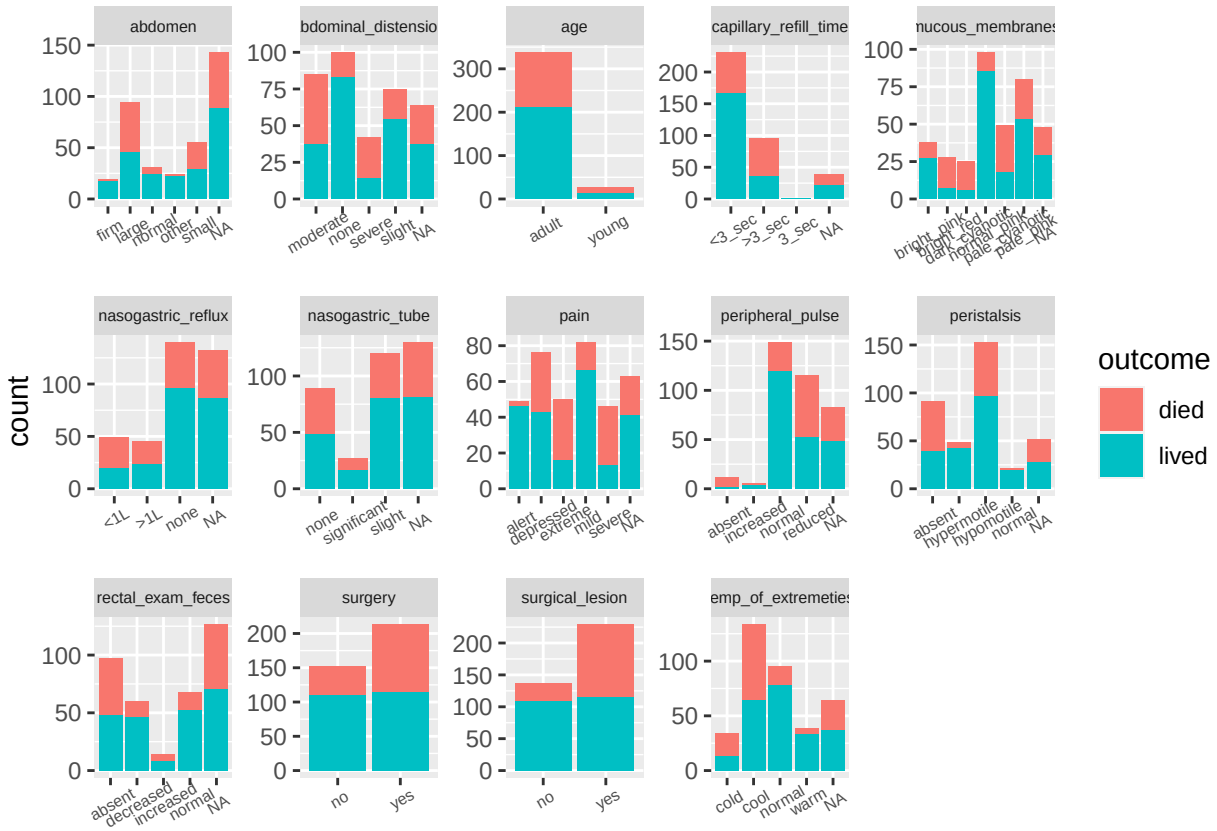
```
data <- data %>%
  mutate(outcome = as.factor(ifelse(outcome == "euthanized",
                                    "died",
                                    levels(outcome)[outcome])))
data %>% group_by(outcome) %>%
  summarize(n = n(), proportion = n()/nrow(data))
```

```
## # A tibble: 2 x 3
##   outcome     n proportion
##   <fct>   <int>      <dbl>
## 1 died      141      0.385
## 2 lived     225      0.615
```

Now, the split between the two outcome categories (survived or died) is more even.

We will now look at the distribution of the different factor variables and how they are split across the two outcome classes.

```
data %>% select_if(~!is.numeric(.)) %>%
  gather(key = "feature", value = "value", -outcome) %>%
  ggplot(aes(value, fill = outcome)) +
  geom_bar(stat = "count") +
  facet_wrap(~feature, scales = "free", nrow = 3) +
  theme(axis.text.x = element_text(angle = 30, vjust = 0.8, size = 6),
        axis.title.x = element_blank(),
        strip.text.x = element_text(size = 6))
```



There are some variables that are very unbalanced. Especially `age`, `capillary_refill_time` and `peripheral_pulse`. Since `peripheral_pulse` additionally has a high number of missing values, we will remove this feature from the data set.

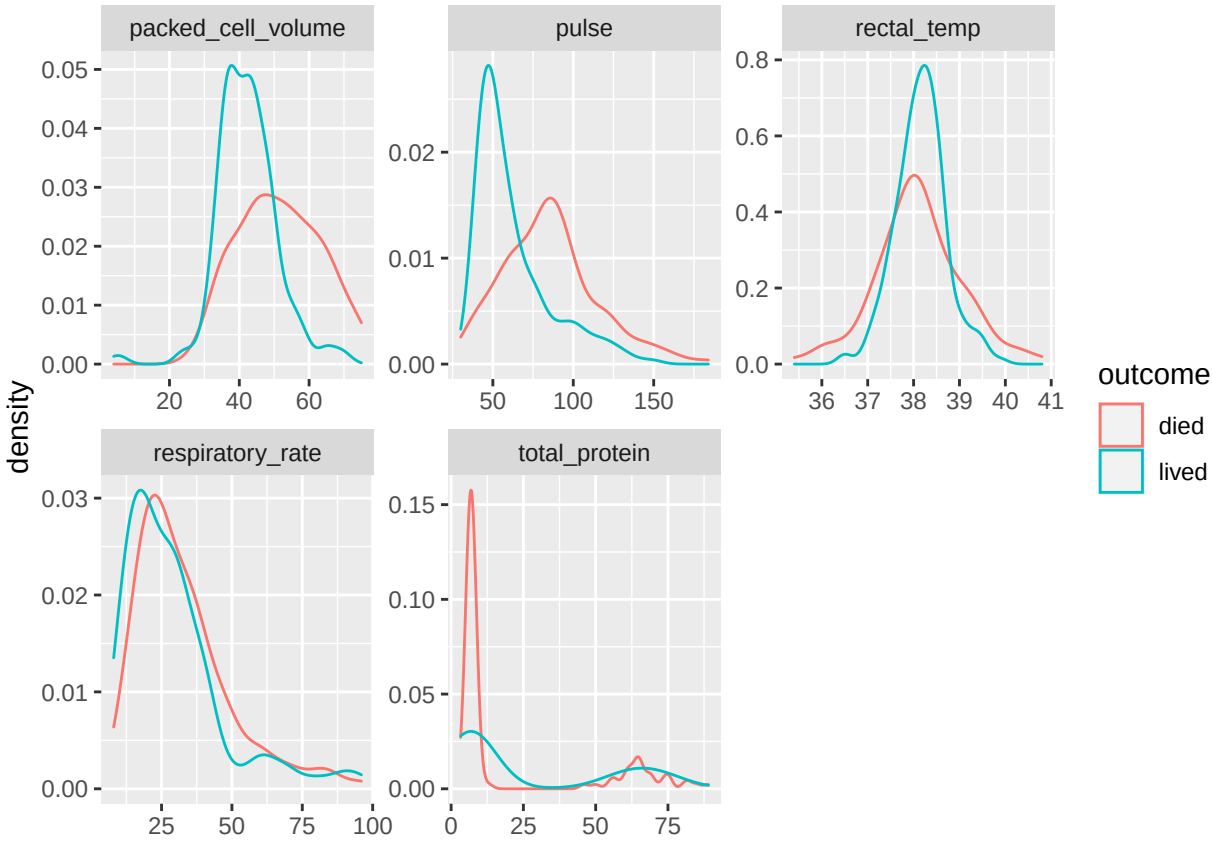
```
data <- data %>% select(-peripheral_pulse)
```

The `capillary_refill_time` variable has following levels: 3 seconds, less than 3 seconds and more than 3 seconds. This is a somewhat unfortunate classification and the level *3 seconds* occurs very rarely. Therefore, all horses observed with a capillary refill time of 3 seconds or more than 3 seconds will be combined into one category (equal or more than 3 seconds).

```
data <- data %>%
  mutate(capillary_refill_time =
           as.factor(ifelse(capillary_refill_time %in% c("3_sec", ">3_sec"),
                            ">=3_sec",
                          levels(capillary_refill_time)[capillary_refill_time])))
```

After taking a closer look at the categorical variables, we will evaluate the continuous variables and there distribution depending on the outcome variable.

```
data %>% select(which(sapply(.,class) == "numeric"), outcome) %>%
  gather(key = "feature", value = "value", -outcome) %>%
  ggplot(aes(value, color = outcome)) +
  geom_density() +
  facet_wrap(~feature, scales = "free") +
  theme(axis.title.x = element_blank())
```



The distribution of total protein values looks strange. Let's take a look at the first entries of the actual data.

```
head(data$total_protein)
```

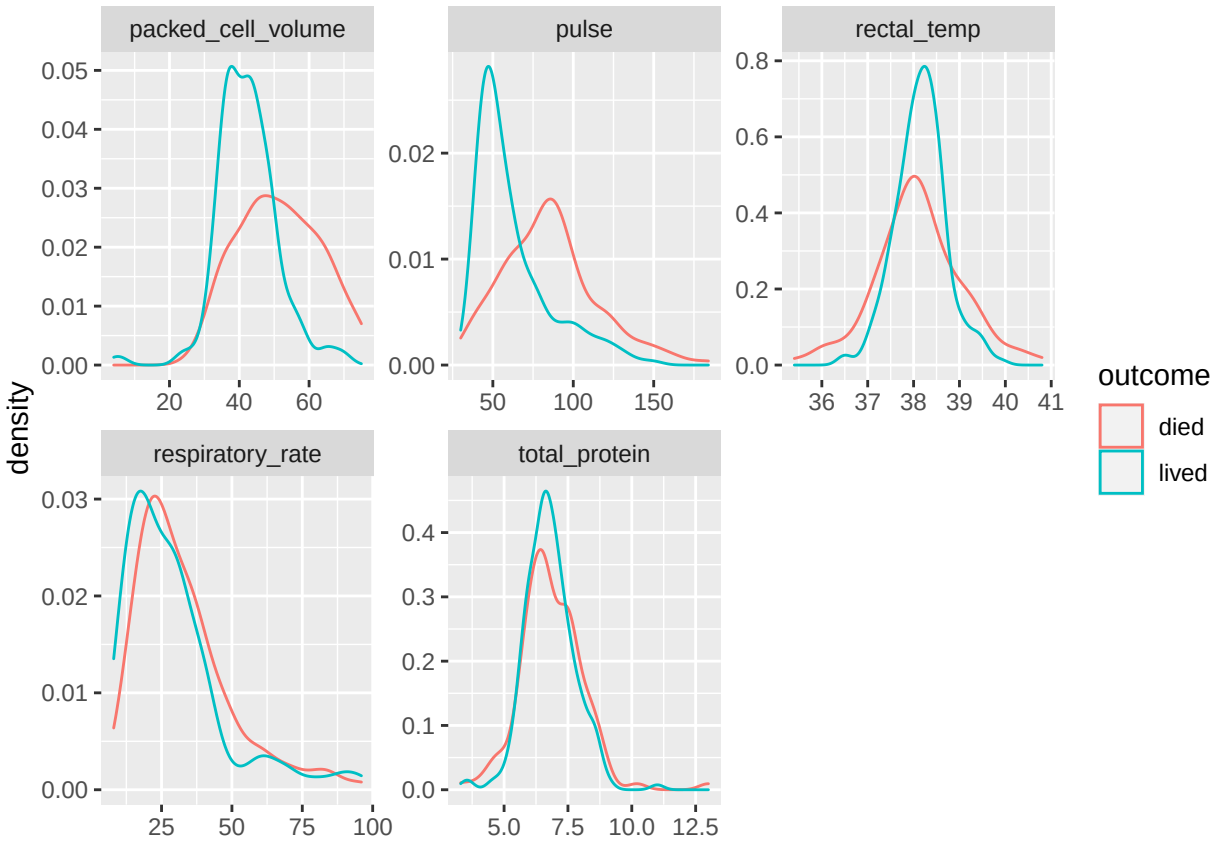
```
## [1]  8.4 85.0  6.7  7.2  7.4   NA
```

It seems that the data has been entered in different units. Some of the values are entered in g/L and the others in g/dL. Those entered in g/L are therefore converted to g/dL.

```
data <- data %>%
  mutate(total_protein = ifelse(total_protein > 25, total_protein/10, total_protein))
```

Let's look at the distributions again.

```
data %>% select(which(sapply(.,class) == "numeric"), outcome) %>%
  gather(key = "feature", value = "value", -outcome) %>%
  ggplot(aes(value, color = outcome)) +
  geom_density() +
  facet_wrap(~feature, scales = "free") +
  theme(axis.title.x = element_blank())
```



That's much better. We can see that especially for `packed_cell_volume` and `pulse`, the distributions seem to differ between horses that survived and those that died. Horses that died had higher packed cell volumes and pulse rates. So these two features seem to be important for predicting whether a horse will survive or

not. For `rectal_temp`, the values seem to be more variable in the group of horses that died, but there is no clear shift in one direction.

For the first algorithm (to predict whether the horse will survive or not), presence and characteristics of surgical lesions will not be included as features since in practice, they would only be available if the horse has undergone surgery. All cases included in this data set were either operated upon or autopsied so that this information was always known. The purpose of the algorithm however, is to predict survival based on clinical symptoms in practice, where these information are not always available. Therefore, they will be removed from the data set.

```
data_survive <- data %>% select(-surgical_lesion, -lesion_1, -lesion_2, -lesion_3)
```

With the second model, we want to predict whether a horse is going to need surgery or not. Therefore, we use the `surgical_lesion` parameter as outcome variable and remove the `outcome` variable instead. We also remove the information whether the horse was treated surgically or not by removing the `surgery` variable as this information should obviously have high predictive value but would be available only in retrospect.

```
data_surgery <- data %>% select(-outcome, -lesion_1, -lesion_2, -lesion_3, -surgery)
```

We will take a look at how many horses in the data set were treated surgically and how many without surgery.

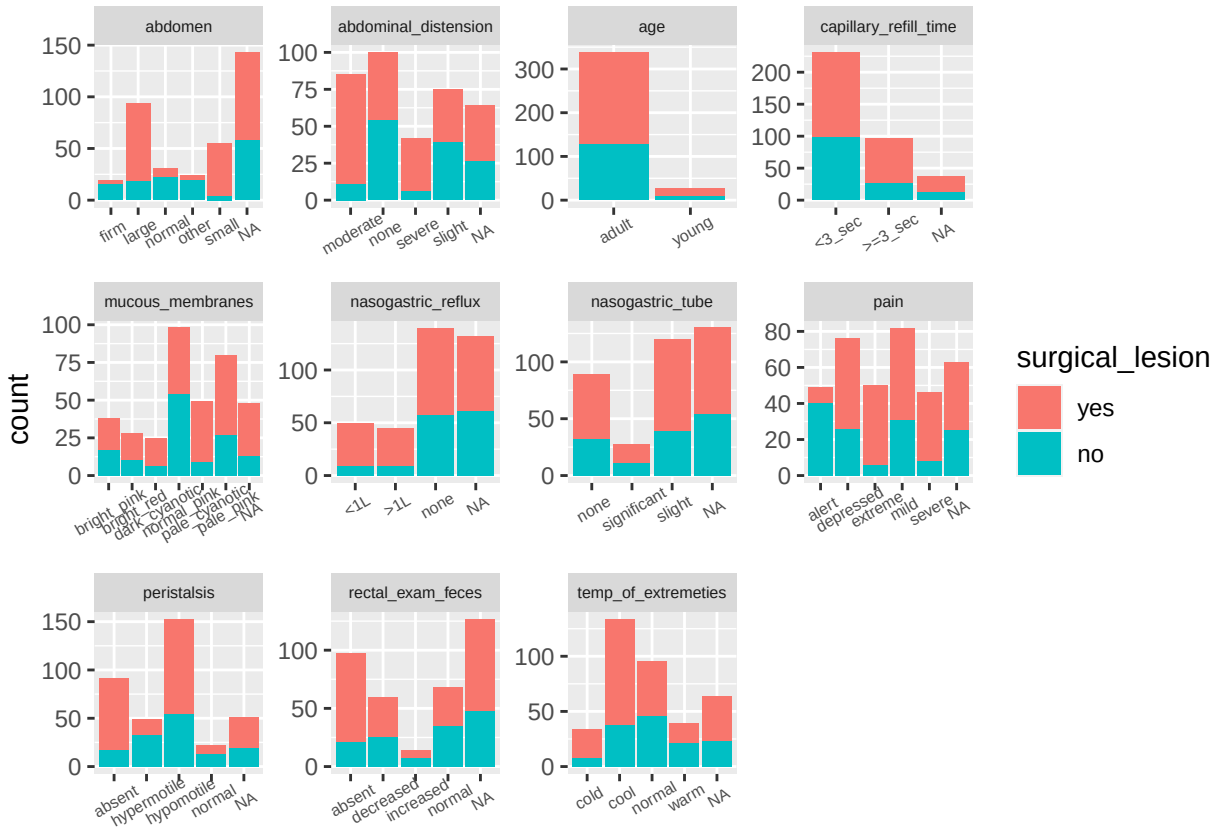
```
data_surgery %>% group_by(surgical_lesion) %>%
  summarize(n = n(), proportion = n()/nrow(data))
```

```
## # A tibble: 2 x 3
##   surgical_lesion     n proportion
##   <fct>           <int>      <dbl>
## 1 yes               230      0.628
## 2 no                136      0.372
```

Approximately 63% of the horses were treated surgically.

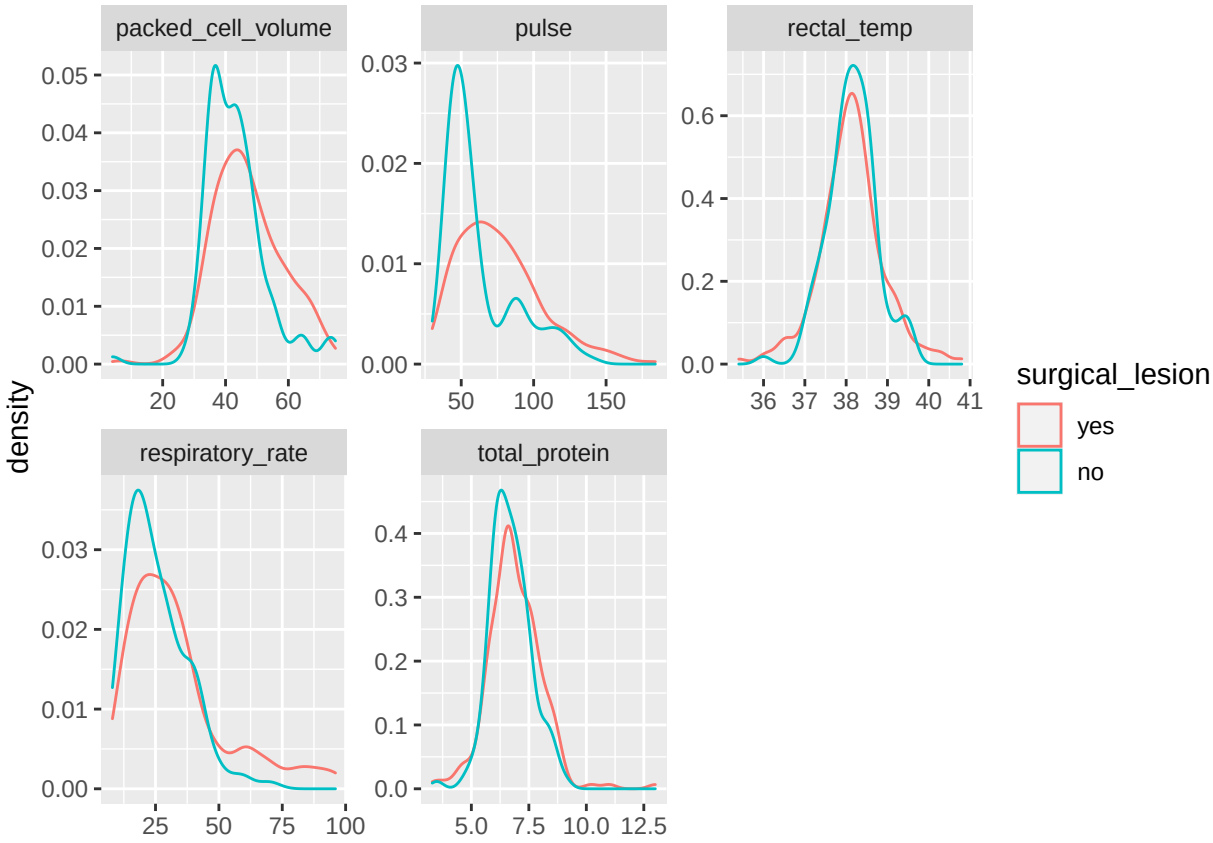
We will again look at how the different levels of the categorical features are distributed across horses that underwent surgery and those that did not.

```
data_surgery %>%
  select_if(~!is.numeric(.)) %>%
  gather(key = "feature", value = "value", -surgical_lesion) %>%
  ggplot(aes(value, fill = surgical_lesion)) +
  geom_bar(stat = "count") +
  facet_wrap(~feature, scales = "free", nrow = 3) +
  theme(axis.text.x = element_text(angle = 30, vjust = 0.8, size = 6),
        axis.title.x = element_blank(),
        strip.text.x = element_text(size = 6))
```



And how the distributions of the continuous variables differ between horses that have been treated surgically and those that were treated conservatively.

```
data_surgery %>% select(which(sapply(.,class) == "numeric"), surgical_lesion) %>%
  gather(key = "feature", value = "value", -surgical_lesion) %>%
  ggplot(aes(value, color = surgical_lesion)) +
  geom_density() +
  facet_wrap(~feature, scales = "free") +
  theme(axis.title.x = element_blank())
```



Again the greatest difference in the distribution can be seen for `packed_cell_volume` and `pulse`, but `respiratory_rate` as well.

2.1.2 Multiple imputation of missing values

Now, we have to deal with the missing values in the remaining features. These are the remaining features that contain missing values:

```
data %>% select_if(~any(is.na(.))) %>%
  summarise(across(everything(), ~sum(is.na(.)))) %>% t() %>%
  as.data.frame() %>%
  rename(missing = V1) %>%
  mutate(missing_percentage = round(missing/nrow(data)*100))
```

```
##                         missing missing_percentage
## rectal_temp                  69                 19
## pulse                        26                  7
## respiratory_rate             71                 19
## temp_of_extremeties          64                 17
## mucous_membranes             48                 13
## capillary_refill_time        38                 10
## pain                         63                 17
## peristalsis                  51                 14
## abdominal_distension         64                 17
## nasogastric_tube            130                 36
```

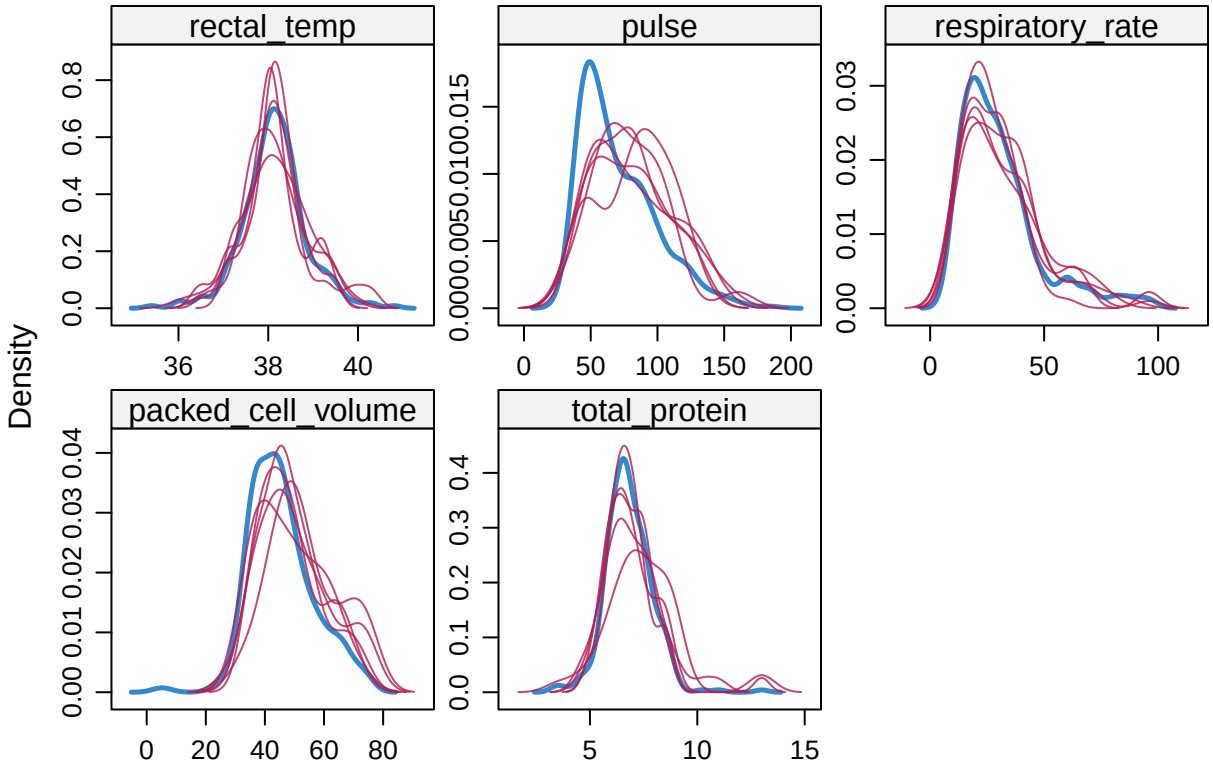
```
## nasogastric_reflux        132                36
## rectal_exam_feces         127                35
## abdomen                   143                39
## packed_cell_volume         36                10
## total_protein              42                11
```

One option would be to impute the mean or median for continuous variables and the mode (most frequent value) for categorical variables, but this method does not account for the variability of the missing data. Instead, we will use the **mice** package to impute the missing values by Multiple Imputation of Chained Equation (MICE). We will use the default methods of **mice** for imputation which are *pmm* (Predictive Mean Matching) for continuous variables, *logreg* (Logistic Regression) for binary variables and *polyreg* (Polytomous Logistic Regression) for nominal variables. We will use 5 iterations. (We do this here using the data prepared for the first model)

```
if(!require(mice)) install.packages("mice", repos = "http://cran.us.r-project.org")
library(mice)
predictormatrix <- quickpred(data_survive, exclude = NULL, mincor = 0.1)
set.seed(42, sample.kind = "Rounding") # for reproducible results
imputed <- mice(data_survive, maxit = 5, predictorMatrix = predictormatrix, printFlag = FALSE)
imp <- complete(imputed) # Get the complete data set with imputed values from first iteration
```

We will inspect the imputed values in the numeric variables by comparing the distributions of observed and imputed values.

```
densityplot(imputed)
```



If we look at the density plots, we can see that the observed and imputed values are similarly distributed and the multiple imputations look similar, however, there is some variation. Especially for the `pulse` variable. However, the proportion of missing values in this variable are relatively low.

2.1.3 Split data set into test and training data

After we have cleaned the data set and imputed missing values we are now going to train and test several machine learning algorithms. Before doing this, we have to split our data set into a train and a test set. For this project, we are going to use 80% of our data for training and the other 20% for testing.

```
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
library(caret)
set.seed(42, sample.kind="Rounding") # For reproducible results
test_index <- createDataPartition(y = imp$outcome, times = 1, p = 0.2, list = FALSE)
```

2.2 Training the machine learning algorithms for predicting the horse's survival

2.2.1 Classification tree

At first, we will fit a classification tree to our training data using the `train` function from the **caret** package and the method *rpart*. We will use 10-fold cross-validation on the train set to tune the model parameters.

As we used multiple imputation to impute the missing values (with 5 iterations), we now actually have 5 data sets instead of only one. In order to account for the variability in the imputed values, we will fit a model to all 5 imputed data sets and then pool the results by majority voting.

```
train.control <- trainControl(method = "cv", number = 10, p = 0.8)
pred_rpart <- data.frame(matrix(0, nrow = nrow(test_index), ncol = 5))
set.seed(42, sample.kind="Rounding") # For reproducible results
for (i in 1:5){
  train <- complete(imputed, i)[-test_index,]
  test <- complete(imputed, i)[test_index,]
  model_rpart <- train(outcome ~ .,
                       method = "rpart",
                       tuneGrid = data.frame(cp = seq(0, 0.05, len = 25)),
                       trControl = train.control,
                       data = train)
  pred_rpart[,i] <- predict(model_rpart, test)
}
pred_rpart_majority <- as.factor(ifelse(rowMeans(pred_rpart == "lived") > 0.5,
                                        "lived", "died"))
```

Let's look at the accuracy of our model comparing the predictions for the outcome in the test set to the actual outcome in the test set.

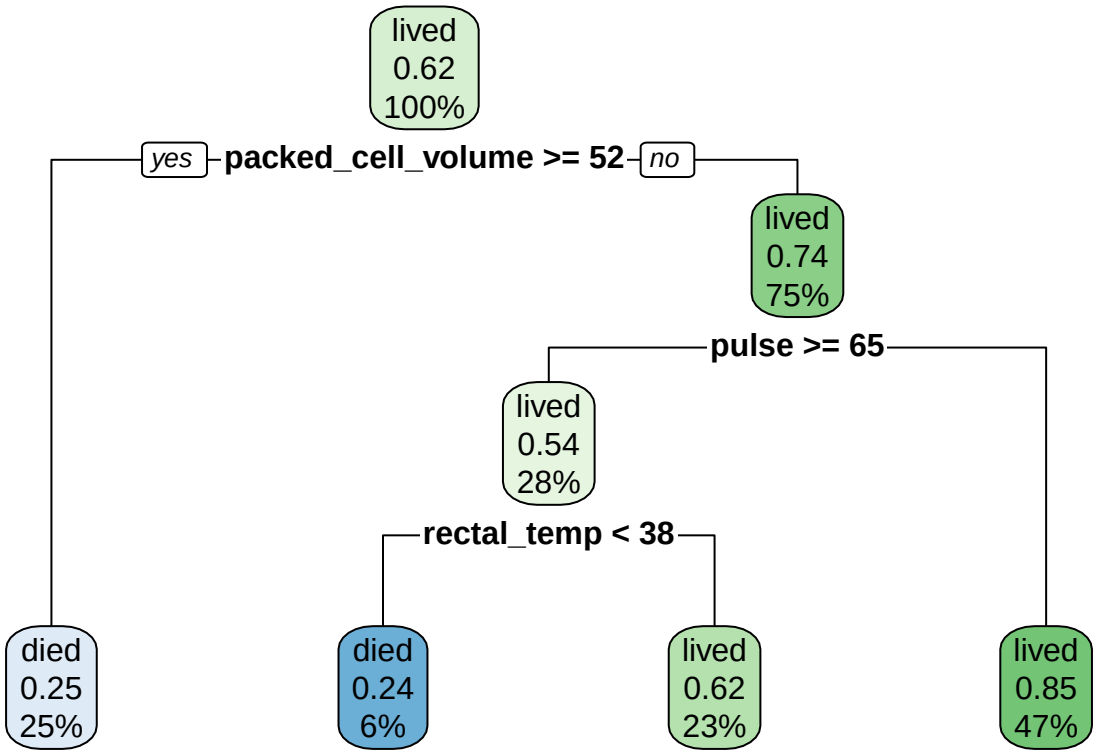
```
cm_rpart <- confusionMatrix(pred_rpart_majority, imp$outcome[test_index])
cm_rpart
```

```
## Confusion Matrix and Statistics
##
##           Reference
```

```
## Prediction died lived
##      died    22     9
##      lived    7    36
##
##                Accuracy : 0.7838
##                  95% CI : (0.6728, 0.8711)
##     No Information Rate : 0.6081
##     P-Value [Acc > NIR] : 0.001027
##
##                   Kappa : 0.5519
##
##  Mcnemar's Test P-Value : 0.802587
##
##             Sensitivity : 0.7586
##             Specificity : 0.8000
##          Pos Pred Value : 0.7097
##          Neg Pred Value : 0.8372
##              Prevalence : 0.3919
##          Detection Rate : 0.2973
##    Detection Prevalence : 0.4189
##       Balanced Accuracy : 0.7793
##
##        'Positive' Class : died
##
```

The accuracy is about 78%. We can plot the final tree (using only the last imputed data set) and look what our model actually looks like.

```
if(!require(rpart.plot)) install.packages("rpart.plot", repos = "http://cran.us.r-project.org")
library(rpart.plot)
rpart.plot(model_rpart$finalModel)
```



We can see that our final tree only uses 3 of the features. Packed cell volume, pulse and rectal temperature. If the packed cell volume is 52% or higher, the probability that the horse is going to survive is only 25%. If the packed cell volume is below 52%, the pulse is below 65 beats/min, the probability that the horse is going to survive is 85%.

The advantage of this simple decision tree is its high interpretability. On the other hand, the accuracy could be higher. In general, decision trees are not very flexible and highly unstable to changes in training data.

2.2.2 Random Forest

Let's see if we can improve the accuracy of our model by fitting a random forest. The goal of fitting a random forest model is to improve prediction performance and reduce instability by averaging multiple decision trees.

```
pred_rf <- data.frame(matrix(0, nrow = nrow(test_index), ncol = 5))
set.seed(42, sample.kind="Rounding") # For reproducible results
for (i in 1:5){
  train <- complete(imputed, i)[-test_index,]
  test <- complete(imputed, i)[test_index,]
  model_rf <- train(outcome ~ .,
                    method = "rf",
                    tuneGrid = data.frame(mtry = seq(1,10,1)),
                    trControl = train.control,
                    data = train)
  pred_rf[,i] <- predict(model_rf, test)
}
pred_rf_majority <- as.factor(ifelse(rowMeans(pred_rf == "lived") > 0.5,
```

```
                                          "lived", "died"))
cm_rf <- confusionMatrix(pred_rf_majority, imp$outcome[test_index])
cm_rf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction died lived
##      died    22     6
##      lived    7    39
##
##                Accuracy : 0.8243
##                  95% CI : (0.7183, 0.903)
##     No Information Rate : 0.6081
##     P-Value [Acc > NIR] : 5.207e-05
##
##                   Kappa : 0.6291
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.7586
##             Specificity : 0.8667
##          Pos Pred Value : 0.7857
##          Neg Pred Value : 0.8478
##              Prevalence : 0.3919
##          Detection Rate : 0.2973
##    Detection Prevalence : 0.3784
##       Balanced Accuracy : 0.8126
##
##        'Positive' Class : died
##
```

Using the random forest model, the accuracy improved by about 4%. The sensitivity is still much lower than the specificity. Let's have a look at the variable importance to examine our random forest model further (In this case we look at the variable importance for the last imputed data set).

```
varImp(model_rf)
```

```
## rf variable importance
##
##   only 20 most important variables shown (out of 37)
##
##                                Overall
## packed_cell_volume             100.000
## pulse                           90.629
## rectal_temp                     59.803
## total_protein                   40.854
## respiratory_rate                35.459
## abdomenlarge                    20.856
## temp_of_extremetiescool         17.271
## abdominal_distensionmoderate    16.678
## painsevere                      15.983
```

```
## painextreme                     13.719
## capillary_refill_time>=3_sec    12.582
## painmild                        10.601
## rectal_exam_fecesdecreased       8.339
## nasogastric_reflux<1L            8.197
## surgeryno                        7.610
## peristalsisabsent                7.277
## nasogastric_tubeslight           5.897
## mucous_membranespale_cyanotic    5.419
## mucous_membranesbright_red       5.300
## mucous_membranespale_pink        5.054
```

As in our first model, `packed_cell_volume`, `pulse` and `rectal_temp` are the most important variables, but in the random forest model, `total_protein` and `respiratory_rate` seem to be important variables to predict whether a horse is going to survive or not as well.

2.2.3 XGBoost

After fitting a decision tree and a random forest model, we will try to improve the accuracy further by fitting a XGBoost model. XGBoost stands for extreme gradient boosting and is an implementation of gradient boosted decision trees. For training an algorithm with XGBoost, we use the **xgboost** package. The first thing we have to do is to prepare the data so that it can be used to train the model with XGBoost. At first, we remove the information about the target variable we want to predict, in this case, the `outcome` variable from the training data and convert the column with the outcome variable to a boolean vector. We call this vector `survived`. `TRUE` indicates that the horses survived, `FALSE` indicates that the horses died.

```
data_xgb <- imp %>% select(-outcome)
survived <- data$outcome == "lived"
```

The XGBoost function only accepts numeric variables. So we have to convert the categorical variables into numeric variables. We can do this by one-hot encoding. One-hot encoding is the process of converting a categorical variable with multiple categories into multiple variables, each with a value of 1 if this observation belongs to that column or 0 if it does not. To do this, we use the `dummyVars` function which expands factors to a set of dummy variables.

```
dmy <- dummyVars(" ~ .", data = data_xgb)
data_xgb <- data.frame(predict(dmy, newdata = data_xgb))
```

The core `xgboost` function requires data to be a matrix. Therefore, we convert the data frame into a matrix.

```
data_xgb_matrix <- data.matrix(data_xgb)
```

After converting the data to a format that is suitable for XGBoost, we have to split the data into a training and a test set. For reproducibility, we use the same split as before.

```
train_data <- data_xgb_matrix[-test_index,]
train_labels <- survived[-test_index]
test_data <- data_xgb_matrix[test_index,]
test_labels <- survived[test_index]
```

XGBoost has a built-in datatype, `DMatrix`, that is particularly good at storing and accessing sparse matrices efficiently. So, the very final step is to convert our matrices into `DMatrix` objects.

```r
if(!require(xgboost)) install.packages("xgboost", repos = "http://cran.us.r-project.org")
library(xgboost)
dtrain <- xgb.DMatrix(data = train_data, label= train_labels)
dtest <- xgb.DMatrix(data = test_data, label= test_labels)
```

We are going to start by training a basic model with all the hyperparameters at default settings. Because we want to predict something that's binary (either horses survive or they don't), we're going to use `binary:logistic`, which is logistic regression for binary (two-class) classification.

```r
set.seed(42, sample.kind="Rounding") # For reproducible results
model_xgb_default <- xgboost(data = dtrain,
                             nrounds = 100,
                             objective = "binary:logistic",
                             eval_metric = "logloss",
                             verbose = F)

pred_train <- predict(model_xgb_default, dtrain)
err_train <- mean(as.numeric(pred_train > 0.5) != train_labels)
pred_test <- predict(model_xgb_default, dtest)
err_test <- mean(as.numeric(pred_test > 0.5) != test_labels)
print(paste("train-error=", err_train, "test-error=", err_test))
```

```
## [1] "train-error= 0 test-error= 0.243243243243243"
```

We see that the error on the training data is much lower (in fact there is no error), than the error on the test set. This implicates that our model is overfitted. We can tune the hyperparameters in the xgboost model in order to prevent overfitting.

In order to control the complexity of our model, we can reduce the `max_depth` parameter. The `max_depth` is the maximum number of nodes allowed from the root to the farthest leaf of a tree. Deeper trees can model more complex relationships by adding more nodes, but as we go deeper, splits become less relevant and are sometimes only due to noise, causing the model to overfit. Additionally, we can increase the `min_child_weight` parameter. The `min_child_weight` is the minimum weight required to create a new node in the tree. A smaller `min_child_weight` allows the algorithm to create children that correspond to fewer samples, thus allowing for more complex trees.

To control the sampling of the data set that is done at each boosting round we can adjust the `subsample` parameter. Instead of using the whole training set every time, we can build our model on slightly different data at each step, which makes it less likely to overfit to a single sample or feature. Furthermore, by adjusting the 'colsample_bytree' parameter, we can tell the model to use a subsample of features to consider for constructing each tree. By using only a random subsample of variables for each tree, the resulting model is more robust to overfitting.

The `eta` parameter controls the learning rate. It defines the rate of "correction" after each boosting round. A lower `eta` makes the model more robust to overfitting, however, with a lower learning rate we usually need more boosting rounds to train the model. We will set the `early_stopping_rounds` parameter to 3 in order to stop when there is no further improvement of the model. We will tune the selected hyperparameters in our model by 10-fold cross-validation using the `xgb.cv`function. In order to find the best hyperparameters, we run 100 iterations of cross-validation for models with different hyperparameters randomly selected from a set of different values.

```r
best_param <- list() # Create an empty list for tuned hyperparameters
best_seednumber <- 1234 # Seed from best iteration
best_logloss <- Inf # Logloss from best iteration
```

```
best_logloss_index <- 0 # Index (number of boosting rounds) from best iteration
set.seed(42, sample.kind="Rounding")
for (iter in 1:100) {
  param <- list(objective = "binary:logistic",
                eval_metric = "logloss",
                max_depth = sample(3:9, 1),
                eta = runif(1, 0.01, 0.3),
                subsample = runif(1, 0.6, 1),
                colsample_bytree = runif(1, 0.6, 1),
                min_child_weight = sample(1:40, 1)
  )
  cv.nround <-  1000
  cv.nfold <-  10 # 10-fold cross-validation
  seed.number  <-  sample.int(10000, 1) # set seed for cross-validation
  set.seed(seed.number, sample.kind="Rounding")
  negative_cases <- sum(train_labels == FALSE)
  postive_cases <- sum(train_labels == TRUE)
  mdcv <- xgb.cv(data = dtrain, params = param,
                 nfold = cv.nfold, nrounds = cv.nround,
                 verbose = F, early_stopping_rounds = 3,
                 scale_pos_weight = negative_cases/postive_cases,
                 maximize = FALSE)

  min_logloss <- min(mdcv$evaluation_log$test_logloss_mean)
  min_logloss_index <- which.min(mdcv$evaluation_log$test_logloss_mean)

  if (min_logloss < best_logloss) {
    best_logloss <- min_logloss
    best_logloss_index <- min_logloss_index
    best_seednumber <- seed.number
    best_param <- param
  }
}

# The best index (min_logloss_index) is the best "nround" in the model
nround <- best_logloss_index
set.seed(best_seednumber, sample.kind="Rounding")
model_xgb <- xgboost(data = dtrain, params = best_param, nround = nround, verbose = F)

# Check error in test data
pred_test <- predict(model_xgb, dtest)
err_test <- mean(as.numeric(pred_test > 0.5) != test_labels)
print(err_test)
```

```
## [1] 0.2297297
```

We will use the hyperparameters that we tuned using only the first imputed dataset

As we did before, we will fit a model to each of the 5 imputed data sets and then pool the results by majority voting.

```
pred_xgb <- data.frame(matrix(0, nrow = nrow(test_index), ncol = 5))
set.seed(42, sample.kind="Rounding")
```

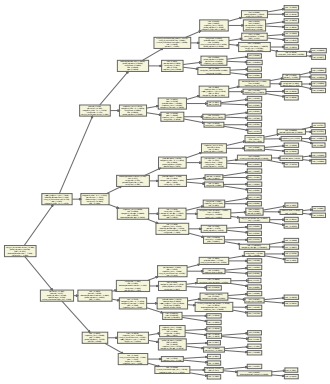
```r
for (i in 1:5){
  data_xgb <- complete(imputed, i) %>% select(-outcome)
  survived <- complete(imputed, i)$outcome == "lived"
  dmy <- dummyVars(" ~ .", data = data_xgb)
  data_xgb <- data.frame(predict(dmy, newdata = data_xgb))
  data_xgb_matrix <- data.matrix(data_xgb)
  train_data <- data_xgb_matrix[-test_index,]
  train_labels <- survived[-test_index]
  test_data <- data_xgb_matrix[test_index,]
  test_labels <- survived[test_index]
  dtrain <- xgb.DMatrix(data = train_data, label= train_labels)
  dtest <- xgb.DMatrix(data = test_data, label= test_labels)
  negative_cases <- sum(train_labels == FALSE)
  postive_cases <- sum(train_labels == TRUE)
  model_xgb <- xgboost(data = dtrain,
                       nrounds = nround,
                       params = best_param,
                       scale_pos_weight = negative_cases/postive_cases,
                       verbose = F)
  pred_xgb[,i] <- predict(model_xgb, dtest)
}
pred_xgb_majority <- as.factor(ifelse(rowMeans(pred_xgb) > 0.5, "lived", "died"))
cm_xgb <- confusionMatrix(pred_xgb_majority, data$outcome[test_index])
cm_xgb
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction died lived
##      died    25     6
##      lived    4    39
##
##                Accuracy : 0.8649
##                  95% CI : (0.7655, 0.9332)
##     No Information Rate : 0.6081
##     P-Value [Acc > NIR] : 1.194e-06
##
##                   Kappa : 0.7199
##
##  Mcnemar's Test P-Value : 0.7518
##
##             Sensitivity : 0.8621
##             Specificity : 0.8667
##          Pos Pred Value : 0.8065
##          Neg Pred Value : 0.9070
##              Prevalence : 0.3919
##          Detection Rate : 0.3378
##    Detection Prevalence : 0.4189
##       Balanced Accuracy : 0.8644
##
##        'Positive' Class : died
##
```

The accuracy of the xgboost model is approximately 86%.

Let's take a closer look at the model (fitted to the last imputed data set) by plotting it.

```
xgb.plot.multi.trees(feature_names = names(data_xgb_matrix), model = model_xgb)
```



2.2.4 Model Evaluation

```
result_firstmodel <- data.frame("Model" = c("Classification Tree",
                                            "Random Forest",
                                            "XGBoost"),
                                "Accuracy" = c(cm_rpart$overall["Accuracy"],
                                               cm_rf$overall["Accuracy"],
                                               cm_xgb$overall["Accuracy"]),
                                "Sensitivity" = c(cm_rpart$byClass["Sensitivity"],
                                                  cm_rf$byClass["Sensitivity"],
                                                  cm_xgb$byClass["Sensitivity"]),
                                "Specificity" = c(cm_rpart$byClass["Specificity"],
                                                  cm_rf$byClass["Specificity"],
                                                  cm_xgb$byClass["Specificity"]))
result_firstmodel
```

```
##                 Model  Accuracy Sensitivity Specificity
## 1 Classification Tree 0.7837838   0.7586207   0.8000000
## 2       Random Forest 0.8243243   0.7586207   0.8666667
## 3             XGBoost 0.8648649   0.8620690   0.8666667
```

The accuracy of the xgboost model is considerably higher than the accuracy of the random forest model and classification tree. Moreover, the sensitivity of the xgboost model is considerably higher than for the other

two models, which means that the classification tree and random forest models are more likely to falsely predict that a horse will survive (false negative) than to predict that a horse dies when it actually survives (false positives). Intuitively, this may be somewhat expected considering the fact that a horse with very severe symptoms may still survive if appropriately treated but a horse with rather mild symptoms is less likely to unexpectedly die despite treatment. In the case of predicting survivability in horses with colic, from a clinical standpoint, sensitivity (correctly predicting that a horse will survive) and specificity (correctly predicting that a horse will die) seem to be equally important. In contrast to for example diagnostic tests, where sensitivity may be more important than specificity, in this case we would want our models sensitivity and specificity to be both as high as possible. For the xgboost model, sensitivity and specificity are both evenly high. From a practical standpoint we would therefore assess the xgboost model as best performing model, although the overall accuracy of the random forest model is slightly higher.

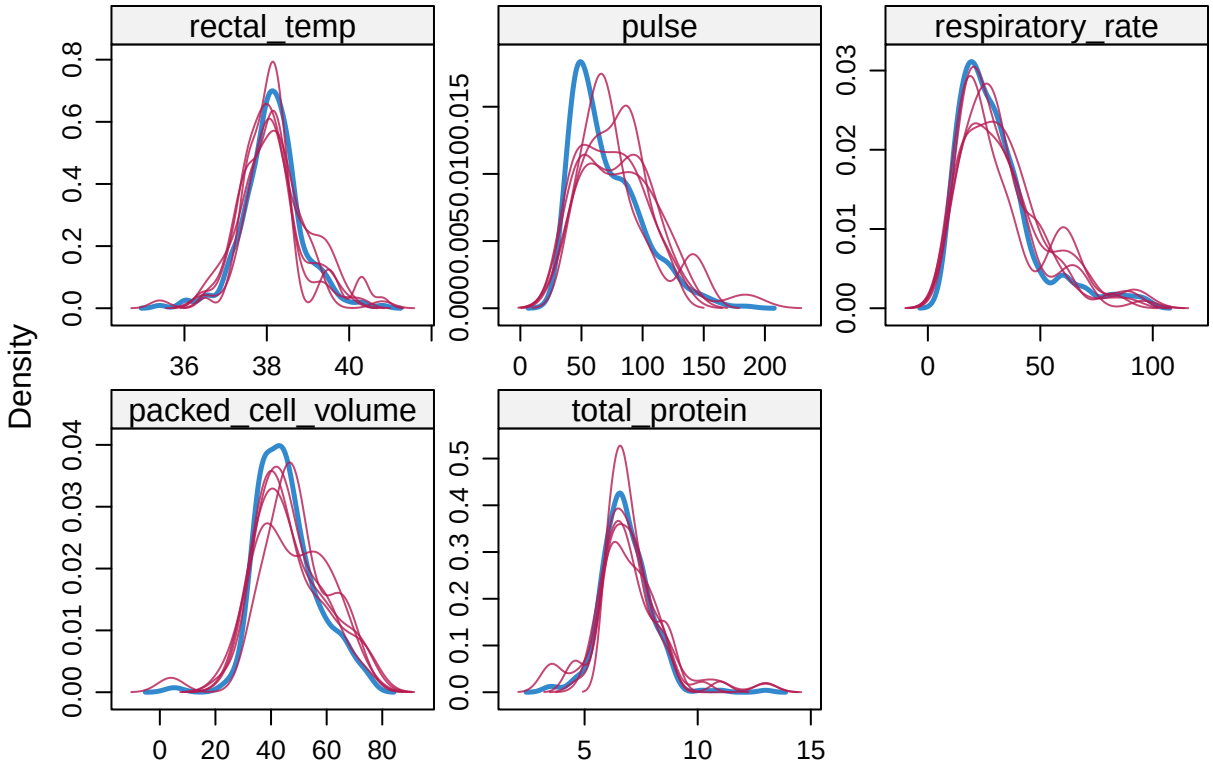
2.2 Training the machine learning algorithms for predicting the horse's need for surgery

With a second model, we will now try to predict whether a horse needs surgery or not.

2.2.1 Multiple Imputation of missing values

As we did for our first model, we will impute the missing values using **mice**.

```
predictormatrix <- quickpred(data_surgery, exclude = NULL, mincor = 0.1)
set.seed(42, sample.kind = "Rounding") # for reproducible results
imputed <- mice(data_surgery, maxit = 5, predictorMatrix = predictormatrix, printFlag = FALSE)
imp <- complete(imputed)
densityplot(imputed)
```



After multiple imputation of the missing values, we split the data into training and test data sets. We will use 80% of the observations for training and 20% for testing again.

```
set.seed(42, sample.kind="Rounding") # For reproducible results
test_index <- createDataPartition(y = imp$surgical_lesion, times = 1, p = 0.2, list = FALSE)
```

2.2.2 Classification tree

Our first model will again be the classification tree. We again fit a model to each of the 5 imputed data sets and then pool the results by majority voting.

```
train.control <- trainControl(method = "cv", number = 10, p = 0.8)
pred_rpart <- data.frame(matrix(0, nrow = nrow(test_index), ncol = 5))
set.seed(42, sample.kind="Rounding") # For reproducible results
for (i in 1:5){
  train <- complete(imputed, i)[-test_index,]
  test <- complete(imputed, i)[test_index,]
  model_rpart <- train(surgical_lesion ~ .,
                       method = "rpart",
                       tuneGrid = data.frame(cp = seq(0, 0.05, len = 25)),
                       trControl = train.control,
                       data = train)
  pred_rpart[,i] <- predict(model_rpart, test)
}
pred_rpart_majority <- as.factor(ifelse(rowMeans(pred_rpart == "yes") > 0.5,
```

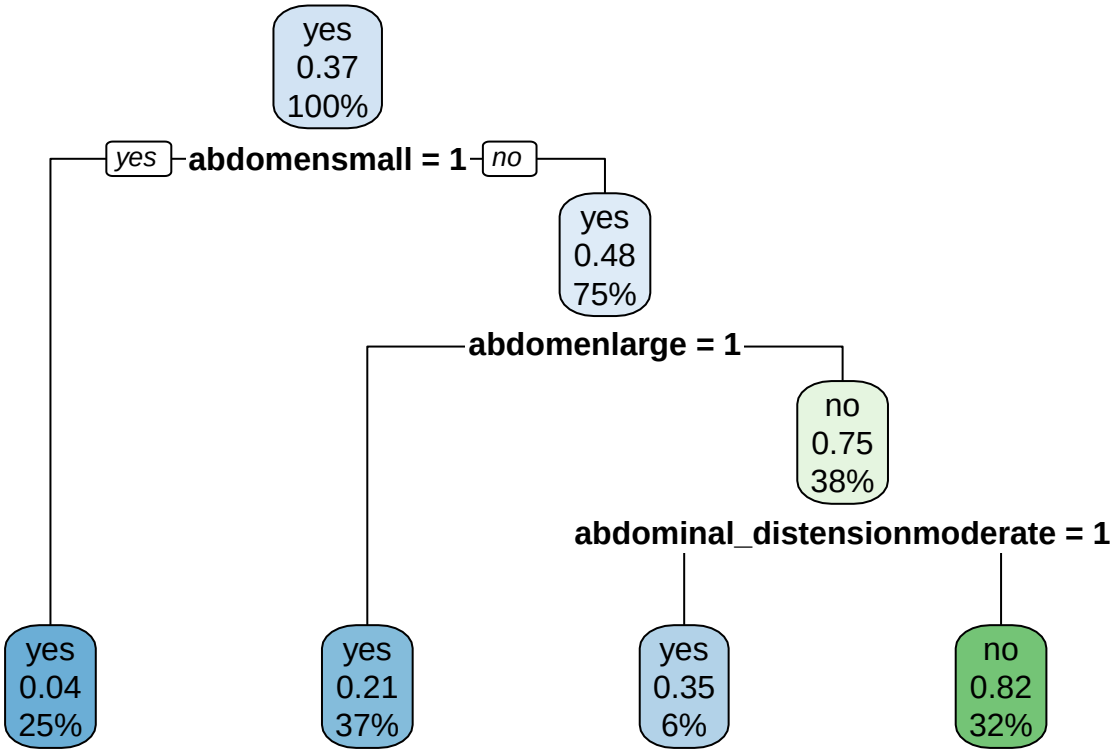
```
                                          "yes", "no"))
cm_rpart <- confusionMatrix(pred_rpart_majority, test$surgical_lesion)
cm_rpart
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction yes no
##        yes  40  5
##        no    6 23
##
##                Accuracy : 0.8514
##                  95% CI : (0.7496, 0.9234)
##     No Information Rate : 0.6216
##     P-Value [Acc > NIR] : 1.283e-05
##
##                   Kappa : 0.6862
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.8696
##             Specificity : 0.8214
##          Pos Pred Value : 0.8889
##          Neg Pred Value : 0.7931
##              Prevalence : 0.6216
##          Detection Rate : 0.5405
##    Detection Prevalence : 0.6081
##       Balanced Accuracy : 0.8455
##
##        'Positive' Class : yes
##
```

The overall accuracy is approximately 85% with a slightly higher sensitivity than specificity.

Let's have a look at the resulting decision tree (for the model fitted to the last imputed data set).

```
rpart.plot(model_rpart$finalModel)
```



The final classification tree only uses 2 of the features, abdomen and abdominal distension. If the abdomen is small, the probability that the horse is going to need surgery is only 4%. If the abdomen is not small but also not large and the abdominal distension is moderate, the probability that the horse is not going to need surgery and can be treated conservatively is 82%.

2.2.3 Random Forest

As our second model we will fit a random forest again.

```
pred_rf <- data.frame(matrix(0, nrow = nrow(test), ncol = 5))
set.seed(42, sample.kind="Rounding") # For reproducible results
for (i in 1:5){
  train <- complete(imputed, i)[-test_index,]
  test <- complete(imputed, i)[test_index,]
  model_rf <- train(surgical_lesion ~ .,
                    method = "rf",
                    tuneGrid = data.frame(mtry = seq(1,10,1)),
                    trControl = train.control,
                    data = train)
  pred_rf[,i] <- predict(model_rf, test)
}
pred_rf_majority <- as.factor(ifelse(rowMeans(pred_rf == "yes") > 0.5,
                                     "yes", "no"))
cm_rf <- confusionMatrix(pred_rf_majority, test$surgical_lesion)
cm_rf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction yes no
##        yes  41  7
##        no    5 21
##
##                Accuracy : 0.8378
##                  95% CI : (0.7339, 0.9133)
##     No Information Rate : 0.6216
##     P-Value [Acc > NIR] : 4.269e-05
##
##                   Kappa : 0.6504
##
##  Mcnemar's Test P-Value : 0.7728
##
##             Sensitivity : 0.8913
##             Specificity : 0.7500
##          Pos Pred Value : 0.8542
##          Neg Pred Value : 0.8077
##              Prevalence : 0.6216
##          Detection Rate : 0.5541
##    Detection Prevalence : 0.6486
##       Balanced Accuracy : 0.8207
##
##        'Positive' Class : yes
##
```

The accuracy of the random forest model in this case is slightly lower than the accuracy of the classification tree. The sensitivity is extremely high, but the specificity is low.

2.2.4 XGBoost

As our third model, we will fit a XGBoost model again. At first, we are going to fit a default model.

```
pred_xgb <- data.frame(matrix(0, nrow = nrow(test), ncol = 5))
set.seed(42, sample.kind="Rounding") # For reproducible results
for (i in 1:5){
  data_xgb <- complete(imputed, i) %>% select(-surgical_lesion)
  surgical_lesion <- complete(imputed, i)$surgical_lesion == "yes"
  dmy <- dummyVars(" ~ .", data = data_xgb)
  data_xgb <- data.frame(predict(dmy, newdata = data_xgb))
  data_xgb_matrix <- data.matrix(data_xgb)
  train_data <- data_xgb_matrix[-test_index,]
  train_labels <- surgical_lesion[-test_index]
  test_data <- data_xgb_matrix[test_index,]
  test_labels <- surgical_lesion[test_index]
  dtrain <- xgb.DMatrix(data = train_data, label= train_labels)
  dtest <- xgb.DMatrix(data = test_data, label= test_labels)
  model_xgb <- xgboost(data = dtrain,
                       nrounds = 100,
                       objective = "binary:logistic",
                       eval_metric = "logloss",
```

```
                      verbose = F)
  pred_xgb[,i] <- predict(model_xgb, dtest)
}
pred_xgb_majority <- as.factor(ifelse(rowMeans(pred_xgb) > 0.5, "yes", "no"))
confusionMatrix(pred_xgb_majority, test$surgical_lesion)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction yes no
##        yes  41  5
##        no    5 23
##
##                Accuracy : 0.8649
##                  95% CI : (0.7655, 0.9332)
##     No Information Rate : 0.6216
##     P-Value [Acc > NIR] : 3.489e-06
##
##                   Kappa : 0.7127
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.8913
##             Specificity : 0.8214
##          Pos Pred Value : 0.8913
##          Neg Pred Value : 0.8214
##              Prevalence : 0.6216
##          Detection Rate : 0.5541
##    Detection Prevalence : 0.6216
##       Balanced Accuracy : 0.8564
##
##        'Positive' Class : yes
##
```

The accuracy of the default XGBoost model is already slightly higher than the accuracy of the other two models. Let's see if we can improve that further by tuning the hyperparameters of the model.

```
best_param <- list() # Create an empty list for tuned hyperparameters
best_seednumber <- 1234 # Seed from best iteration
best_logloss <- Inf # Logloss from best iteration
best_logloss_index <- 0 # Index (number of boosting rounds) from best iteration
set.seed(42, sample.kind = "Rounding")
for (iter in 1:100) {
  param <- list(objective = "binary:logistic",
                eval_metric = "logloss",
                max_depth = sample(3:9, 1),
                eta = runif(1, 0.01, 0.3),
                subsample = runif(1, 0.6, 1),
                colsample_bytree = runif(1, 0.6, 1),
                min_child_weight = sample(1:40, 1)
  )
  cv.nround <-  1000
  cv.nfold <-  10 # 10-fold cross-validation
```

```
  seed.number  <-  sample.int(10000, 1) # set seed for cross-validation
  set.seed(seed.number, sample.kind="Rounding")
  negative_cases <- sum(train_labels == FALSE)
  postive_cases <- sum(train_labels == TRUE)
  mdcv <- xgb.cv(data = dtrain, params = param,
                 nfold = cv.nfold, nrounds = cv.nround,
                 verbose = F, early_stopping_rounds = 3,
                 scale_pos_weight = negative_cases/postive_cases,
                 maximize = FALSE)

  min_logloss <- min(mdcv$evaluation_log$test_logloss_mean)
  min_logloss_index <- which.min(mdcv$evaluation_log$test_logloss_mean)

  if (min_logloss < best_logloss) {
    best_logloss <- min_logloss
    best_logloss_index <- min_logloss_index
    best_seednumber <- seed.number
    best_param <- param
  }
}

# The best index (min_logloss_index) is the best "nround" in the model
nround <- best_logloss_index
set.seed(best_seednumber, sample.kind="Rounding")
model_xgb <- xgboost(data = dtrain, params = best_param, nround = nround, verbose = F)

# Check error in test data
pred_test <- predict(model_xgb, dtest)
err_test <- mean(as.numeric(pred_test > 0.5) != test_labels)
print(err_test)
```

```
## [1] 0.1891892
```

We will use the hyperparameters that we tuned (in this case using only the last imputed dataset).

As we did before, we will fit a model to each of the 5 imputed data sets and then pool the results by majority voting.

```
pred_xgb <- data.frame(matrix(0, nrow = nrow(test_index), ncol = 5))
set.seed(42, sample.kind = "Rounding")
for (i in 1:5){
  data_xgb <- complete(imputed, i) %>% select(-surgical_lesion)
  surgical_lesion <- complete(imputed, i)$surgical_lesion == "yes"
  dmy <- dummyVars(" ~ .", data = data_xgb)
  data_xgb <- data.frame(predict(dmy, newdata = data_xgb))
  data_xgb_matrix <- data.matrix(data_xgb)
  train_data <- data_xgb_matrix[-test_index,]
  train_labels <- surgical_lesion[-test_index]
  test_data <- data_xgb_matrix[test_index,]
  test_labels <- surgical_lesion[test_index]
  dtrain <- xgb.DMatrix(data = train_data, label= train_labels)
  dtest <- xgb.DMatrix(data = test_data, label= test_labels)
  negative_cases <- sum(train_labels == FALSE)
```

```
  postive_cases <- sum(train_labels == TRUE)
  model_xgb <- xgboost(data = dtrain,
                       nrounds = nround,
                       params = best_param,
                       scale_pos_weight = negative_cases/postive_cases,
                       verbose = F)
  pred_xgb[,i] <- predict(model_xgb, dtest)
}
pred_xgb_majority <- as.factor(ifelse(rowMeans(pred_xgb) > 0.5, "yes", "no"))
cm_xgb <- confusionMatrix(pred_xgb_majority, test$surgical_lesion)
cm_xgb
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction yes no
##        yes  41  4
##        no    5 24
##
##                Accuracy : 0.8784
##                  95% CI : (0.7816, 0.9429)
##     No Information Rate : 0.6216
##     P-Value [Acc > NIR] : 8.512e-07
##
##                   Kappa : 0.7433
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.8913
##             Specificity : 0.8571
##          Pos Pred Value : 0.9111
##          Neg Pred Value : 0.8276
##              Prevalence : 0.6216
##          Detection Rate : 0.5541
##    Detection Prevalence : 0.6081
##       Balanced Accuracy : 0.8742
##
##        'Positive' Class : yes
##
```

In comparison to the default model, the accuracy of the model with tuned hyperparameters further increase to approximately 88%

2.2.5 Results

```
result_secondmodel <- data.frame("Model" = c("Classification Tree",
                                             "Random Forest",
                                             "XGBoost"),
                                 "Accuracy" = c(cm_rpart$overall["Accuracy"],
                                                cm_rf$overall["Accuracy"],
                                                cm_xgb$overall["Accuracy"]),
```

```
                                "Sensitivity" = c(cm_rpart$byClass["Sensitivity"],
                                                  cm_rf$byClass["Sensitivity"],
                                                  cm_xgb$byClass["Sensitivity"]),
                                "Specificity" = c(cm_rpart$byClass["Specificity"],
                                                  cm_rf$byClass["Specificity"],
                                                  cm_xgb$byClass["Specificity"]))
result_secondmodel
```

```
##                 Model  Accuracy Sensitivity Specificity
## 1 Classification Tree 0.8513514   0.8695652   0.8214286
## 2       Random Forest 0.8378378   0.8913043   0.7500000
## 3             XGBoost 0.8783784   0.8913043   0.8571429
```

In the case of the second algorithm (predicting need for surgery), the xgboost model yields the highest overall accuracy. The sensitivity of all three models is similar, but the specificity of the xgboost model is considerably higher.

2.3. Discussion and limitations

One of the main difficulties in fitting a machine learning algorithm to predict survivability likelihood and need for surgery in horses with colic based on the UCI Horse Colic data set is that the data set contains relatively few observations. When using small data sets to train machine learning algorithms, the risk of overfitting is high. When overfitting occurs, the model excessively adjusts to the training data and performs poorly in predicting new data. Tuning hyperparameters using k-fold cross validation may prevent overfitting to a certain degree, however it still seems advisable to consider results from small data sets with caution.

Another limitation of the data set is the high number of missing observations. Since the strict use of only complete cases would decrease the number of observations even further, missing values were imputed using mice. Usually, the idea behind multiple imputation is to consider the variability of imputed values by fitting separate models to all iterations of imputed datasets and than pool the results into a single multiple imputation result. In this case, models were fitted to 5 iterations of the imputed data set and the final result was calculated by majority voting of all 5 models on the test data set.

Due to the small size of the data set used to train the machine learning algorithm, the random allocation of observations to training and test sets may have a great influence on the performance of the different machine learning algorithms. Therefore, it may be advisable to repeat the process described in this report several times and then calculate mean values and standard deviations for performance measures in order to assess performance of the different models. For the sake of simplicity, this wasn't done in this report. Usually, machine learning algorithms that have been trained and tested using a data set should be finally validated using a final hold-out validation set (which usually is a subset of the original data set) that has not been used throughout the whole model training and selection process (as if the algorithm would be used for completely unknown data). However, due to the very small number of observations in the UCI Horse Colic data set this has not been done in the present report.

2.4. Conclusions

The xgboost model yielded a reasonable high accuracy for both, predicting survivability likelihood (88%) and need for surgery (88%) in horses with colic. The use of this machine learning algorithm in a clinical setting would have great potential to assist clinical decision making in veterinary medicine. Especially if a large number of different clinical symptoms have to be assessed as a whole also considering their mutual interdependence. However in order to be able to reliably predict survivability likelihood and need for surgery based on clinical symptoms, the size of the data set for training the machine learning algorithm should be considerably increased and the algorithm should be finally validated using new data that have not been used throughout the model training and selection process.