Heart Failure Prediction

JF

2024-05-18

Heart Failure Prediction - Clinical Records

Introduction

The following is the dataset description extracted directly from Kaggle website (see References 2):

"This dataset contains the medical records of 5000 patients who had heart failure, collected during their follow-up period, where each patient profile has 13 clinical features. The main goal of this project is to predict if the patient will survive or not

Attribute Information

- *age: age of the patient (years)*
- *anaemia: decrease of red blood cells or hemoglobin (boolean)*
- *creatinine phosphokinase (CPK): level of the CPK enzyme in the blood (mcg/L)*
- *diabetes: if the patient has diabetes (boolean)*
- *ejection fraction: percentage of blood leaving the heart at each contraction (percentage)*
- *high blood pressure: if the patient has hypertension (boolean)*
- *platelets: platelets in the blood (kiloplatelets/mL)*
- *sex: woman or man (binary)*
- *serum creatinine: level of serum creatinine in the blood (mg/dL)*
- *serum sodium: level of serum sodium in the blood (mEq/L)*
- *smoking: if the patient smokes or not (boolean)*
- *time: follow-up period (days)*
- *DEATH_EVENT: if the patient died during the follow-up period (boolean)"*

In the python notebooks provided in the same Kaggle platform, some user achieved AUC of 99% or almost 100%, mainly using ensemble learning from gradient boosting methods (such as LightGBM, XGboost or Catboost). These promising metrics didn't reflect 0 mistakes, just very few.

The main metric used was accuracy (class imbalance was ignored as, we will see, death probability in the dataset was around 31%). It also ignores different costs in the prediction (it assumes that false negatives has the same weight as false positives)

The first part of this work will be replicate in R the findings from this researchers, incorporating class imbalance and error cost imbalance. It will also cover simpler classification models such as Logit to see if Gradient Boosting Machines were over complicating the prediction .

The second part will focus in model explainability. In case a black box method is needed to achieve such results, methods as SHAP Values can help to gain insights of how the model is making decisions.

The third an final part will focus in model distillation. With the insights from model explainability, would it be possible to find a simpler model to do the same job?

The highlight in forecasting with an **explainable** model will serve not only to predict, but maybe to avoid some of this correctly predicted deaths. This would be possible if some of the attributes that increases death probabilities beyond the probability threshold are possible to change by the patient behavior (such as smoking). This insight is more actionable that prediction itself and so far is not covered in the published notebooks on Kaggle.

Analysis

For the analysis of the data, the following path will be consider:

1. Exploratory Data Analysis (EDA)
2. Data Preprocessing
3. Model Calibration (hyperparameter search, model selection)
4. Model Explanation

```r
pacman::p_load(readr,janitor,vtable,tidyverse,tidymodels,DataExplorer,
               ggpubr,ggcorrplot,gghighlight,patchwork,doParallel,vip,
               PRROC,SHAPforxgboost,shapviz, explore, CalibratR, DataExplorer)

tidymodels_prefer()

all_cores <- parallel::detectCores(logical = FALSE)
registerDoParallel(cores = all_cores)
```

EDA

Some quick statistics:

```r
df <- read_csv("heart_failure_clinical_records.csv")

df %>%
  data.frame() %>%
  clean_names -> df

st(df, out = "kable")
```

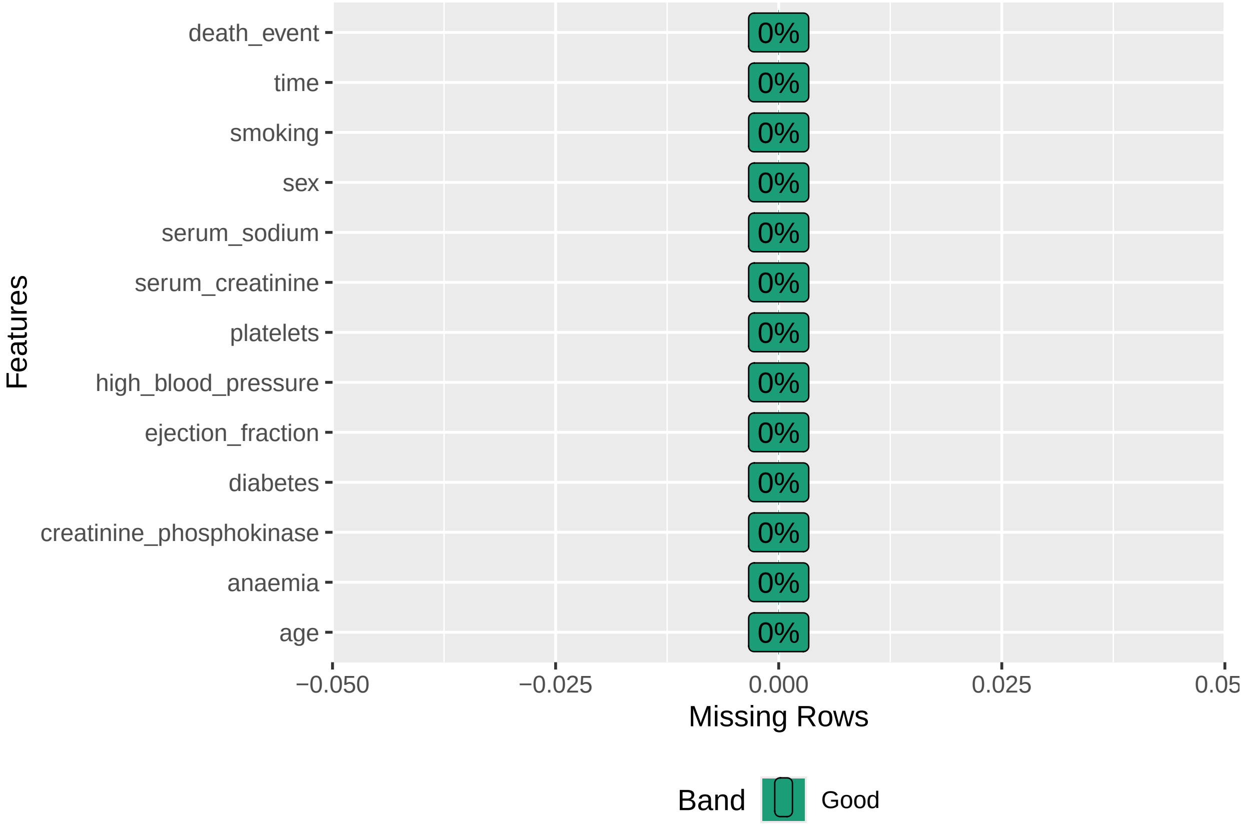
We can see that 31% of people in this data didn't survived.

We will use DataExplorer package to make initial exploration faster. First of all, we check for missing values:

```r
df %>%
  plot_missing()
```

Table 1: Summary Statistics

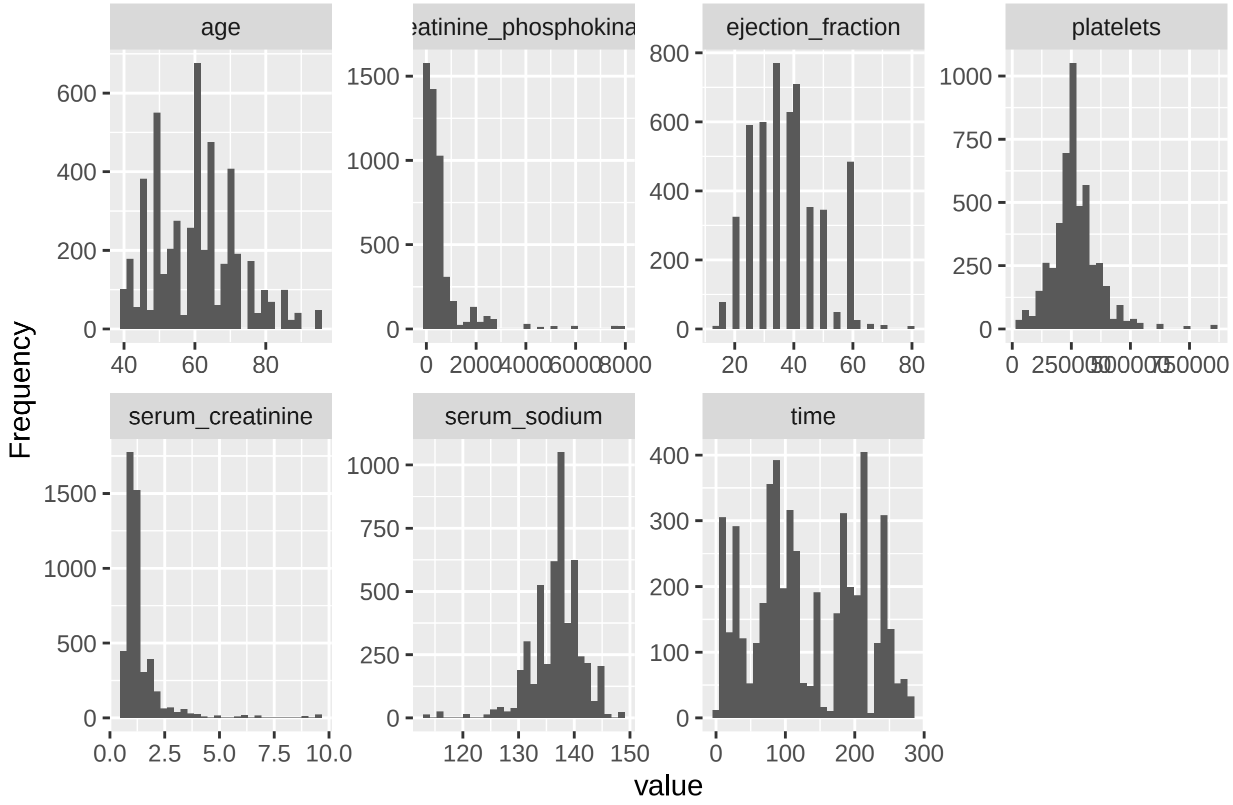
Variable	N	Mean	Std. Dev.	Min	Pctl. 25	Pctl. 75	Max
age	5000	60	12	40	50	68	95
anaemia	5000	0.47	0.5	0	0	1	1
creatinine_phosphokinase	5000	587	977	23	121	582	7861
diabetes	5000	0.44	0.5	0	0	1	1
ejection_fraction	5000	38	12	14	30	45	80
high_blood_pressure	5000	0.36	0.48	0	0	1	1
platelets	5000	265075	98000	25100	215000	310000	850000
serum_creatinine	5000	1.4	1	0.5	0.9	1.4	9.4
serum_sodium	5000	137	4.5	113	134	140	148
sex	5000	0.65	0.48	0	0	1	1
smoking	5000	0.31	0.46	0	0	1	1
time	5000	131	77	4	74	201	285
death_event	5000	0.31	0.46	0	0	1	1



We can see that our data is complete and is no need of missing treatment.

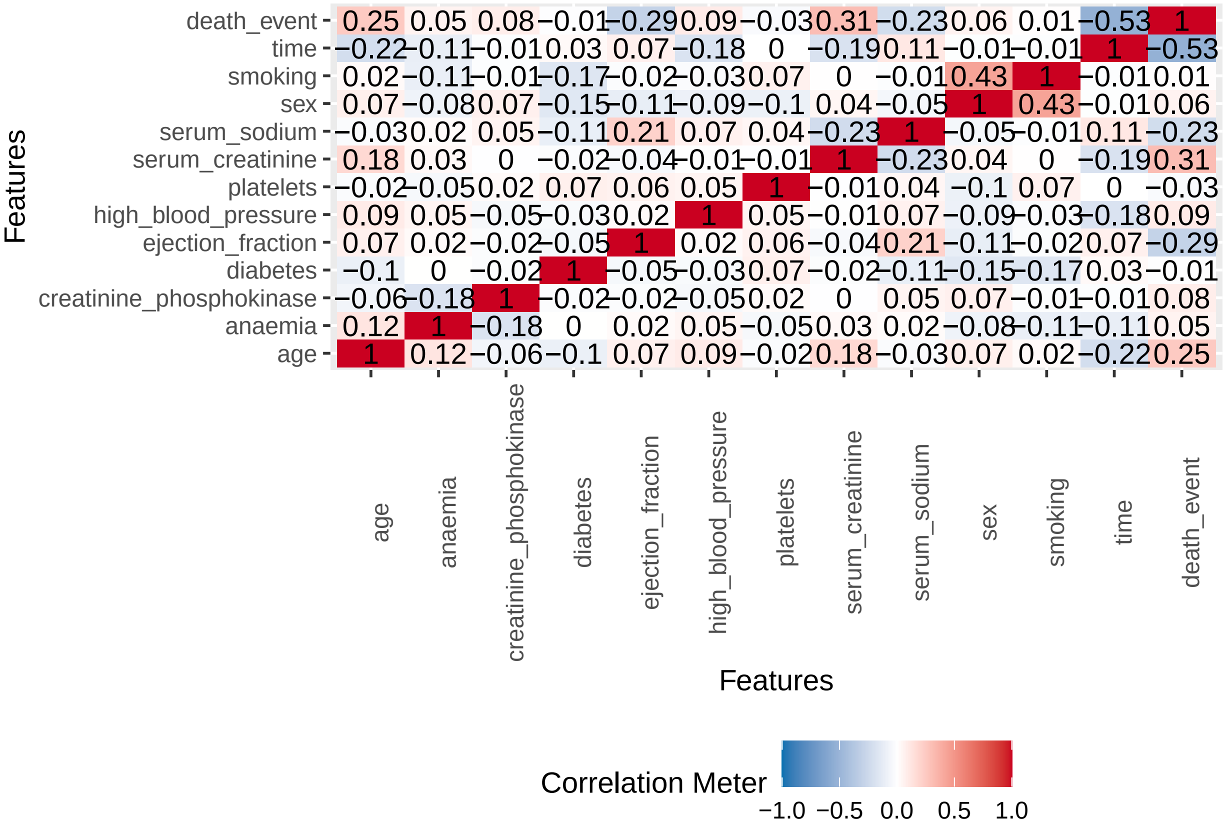
We can also check for distribution in the continuous variables. This will exclude binary variables:

```
df %>%
  plot_histogram()
```



We can check variable correlation (including binary variables). This might help to identify the most promising attributes (correlation with target) or to identify attribute redundance (correlation between attributes).

```
df %>%
  plot_correlation()
```



For example, we can find that "time" attribute has a strong negative correlation with "death_event" (target variable). So, longer follow-up periods are related with less death. It would be important to understand how this data was created to check if some data leakage can be provided indirectly through "time" attribute. Let's say death people are marked as such as soon death happens. If follow up date is influenced by death status, it wouldn't be recommended to use it as predictor. This data leakage might be the reason why a nearly 100% AUC ROC results was obtained by other researchers.

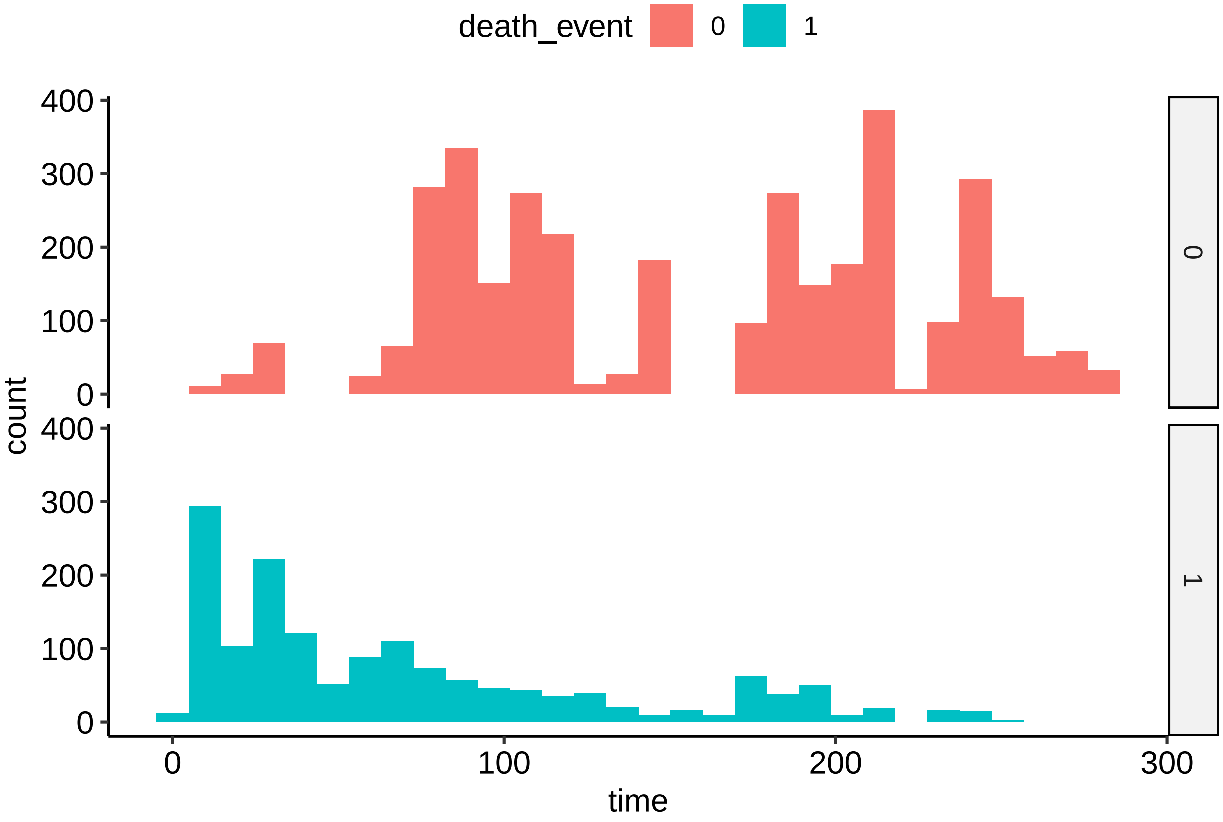
Another way "time" can be affected by date is that the follow-up appointment is fixed by doctors at discharge moment, but doctors might intend to arrange longer follow-up appointments to more promising cases and shorter for the less promising ones. In this way, is not that "time" provide useful information about death probability, is that **doctors criteria** in choosing follow up appointments (hidden in "time" attribute) are related to death probability. If this is the case and "time" is the strongest attribute to achieve a good model, we could never bypass doctors criteria. This will make impossible to automate a decision without doctors inputs and expertise. What will be the value of a predicting algorithm in this case?

In the other hand, if follow up appointment is independent of death result (for example, you fix follow-up dates randomly), shorter follow-up periods contain different information than longer ones. If follow-up period is too short, death might come in the future after appointment and we still won't see it. If follow up appointment is too long, let's say 200 days after the hearth failure event, it might happen that either you show up (you are alive) or not (you don't go to your appointment or you are dead). Are we sure that every person with hearth failure has a follow-up appointment? There is no missing data in the dataset, so either every person with hearth failure has a follow-up meeting, or we are seeing biased data, or missing values were treated somehow.

In my opinion it would be risky to trust predictions with "time" attribute. Let's build a model with it to see if we can replicate results in Kaggle notebooks (and see if the model strongly uses "time" for prediction), but let's build also a model without "time" attributes to continue with the rest of the analysis.

Original paper, cited in References #1, claims that serum creatinine and ejection fraction alone can successfully predict survival. Ignoring "time" should not extremely compromise prediction if true.

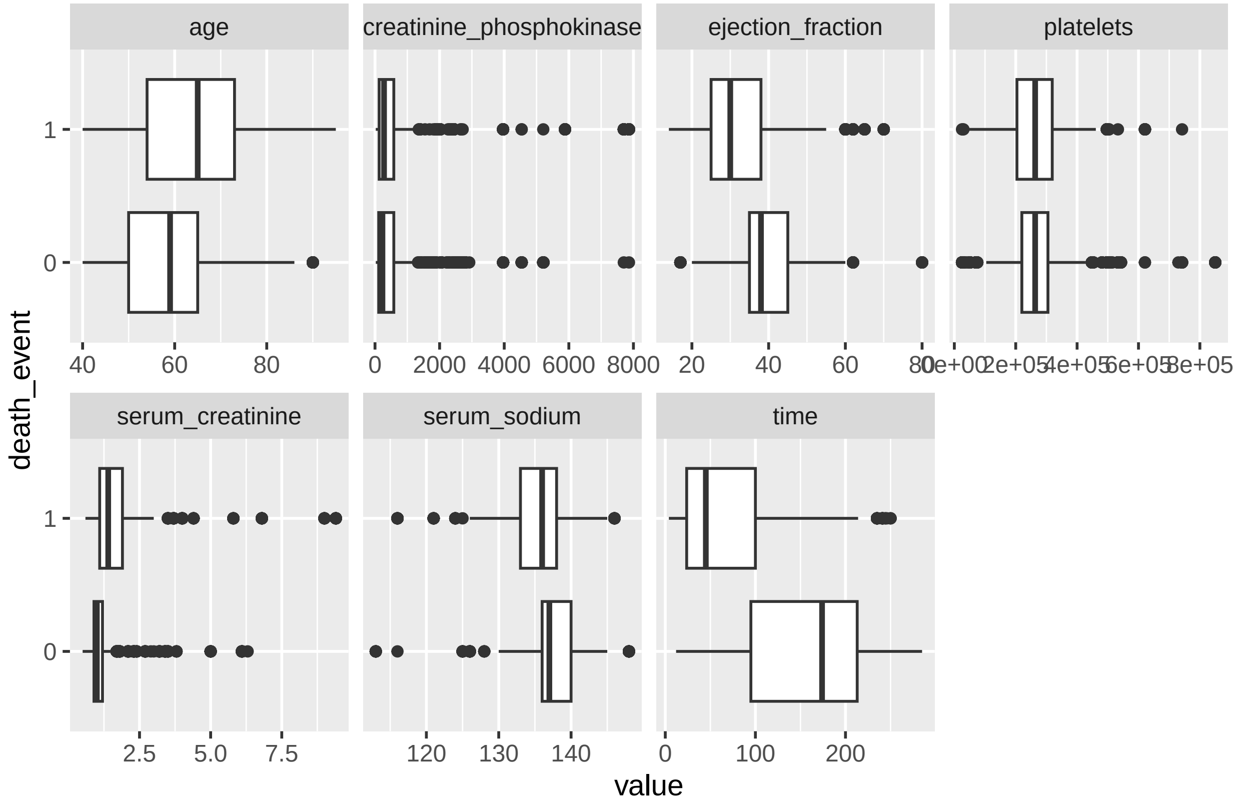
```r
df %>%
  mutate(death_event = factor(death_event)) %>%
  ggplot(aes(x = time, fill = death_event)) +
  geom_histogram() +
  theme_pubr() +
  facet_grid(death_event~.)
```



Previous histogram show the relation in the distribution of "time" between death and alive final status.

We can summarize that comparation with boxplots for every continuous attribute:

```r
df %>%
  mutate(death_event = factor(death_event)) %>%
  plot_boxplot(by = "death_event")
```



Data Preprocessing

For further analysis, we will use Tidymodels framework.

We will split data in training (70%) and test (30%) stratify by death_event.

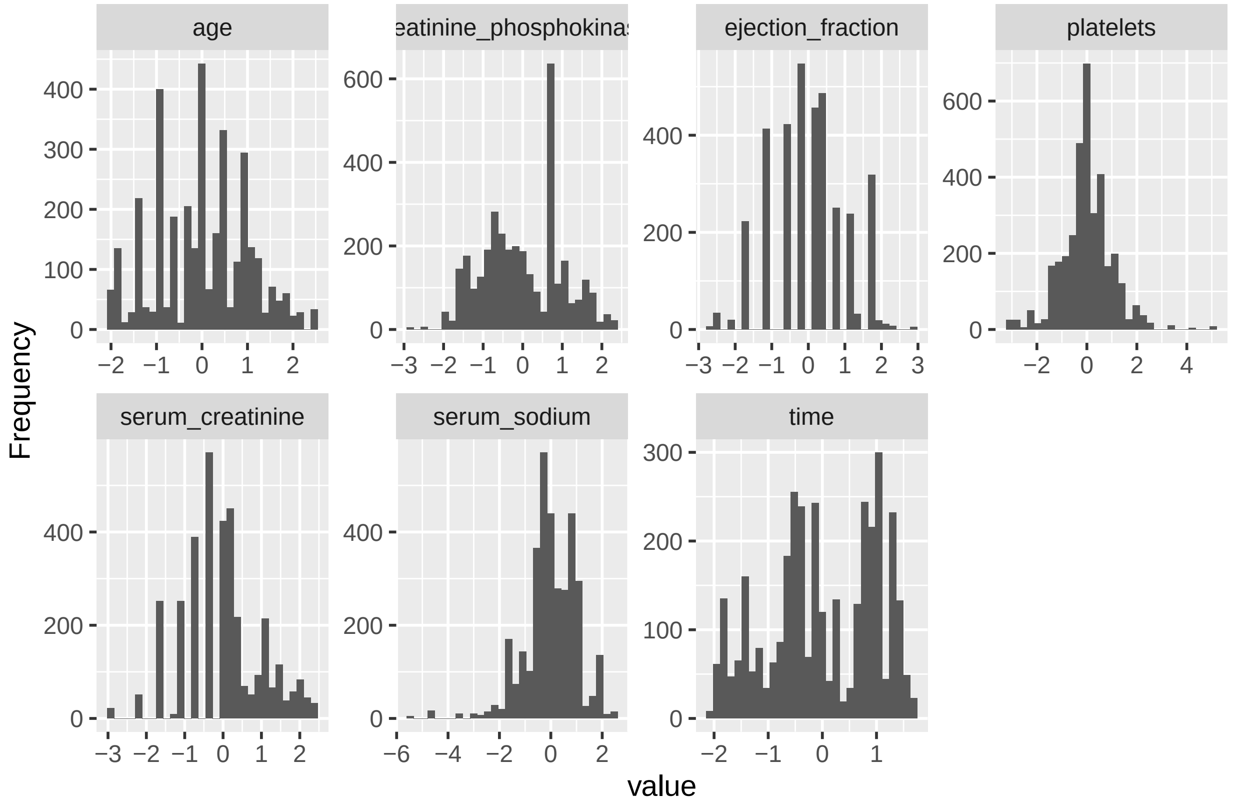
```r
df %>%
  mutate(death_event = factor(death_event)) -> df

set.seed(99)
df_split <- rsample::initial_split(
  df,
  prop = 0.7,
  strata = death_event
)
```

We will use a recipe to preprocess data.

```r
preprocessing_recipe <-
  recipes::recipe(death_event ~ ., data = training(df_split)) %>%
  recipes::step_nzv(all_nominal_predictors()) %>%     # Remove near zero variance
  step_YeoJohnson(all_numeric_predictors()) %>%       # Transform with Yeo-Johnson
  step_normalize(all_numeric_predictors()) %>%        # Normalize variables
  prep()
```

These are the new distributions in the train set:

```r
train <- recipes::bake(
    preprocessing_recipe,
    new_data = training(df_split)
  )

plot_histogram(train)
```



We can see that very skewed distributions like creatinine_phosphokinase now have a more "normal" looking distribution.

Model Calibration

We will use cross-validation to calibrate hyperparameters. We will use 3 folds to avoid splitting too much the data we have for training.

As this will be modeled as a binary classification problem, we will aim to improve ROC AUC metric.

```r
df_cv_folds <-
  recipes::bake(
    preprocessing_recipe,
    new_data = training(df_split)
  ) %>%
  rsample::vfold_cv(v = 3)
```

Model 1: XGBoost with time To replicate results in Kaggle's notebooks, we will use a XGBoost to see if we can get closer to those results.

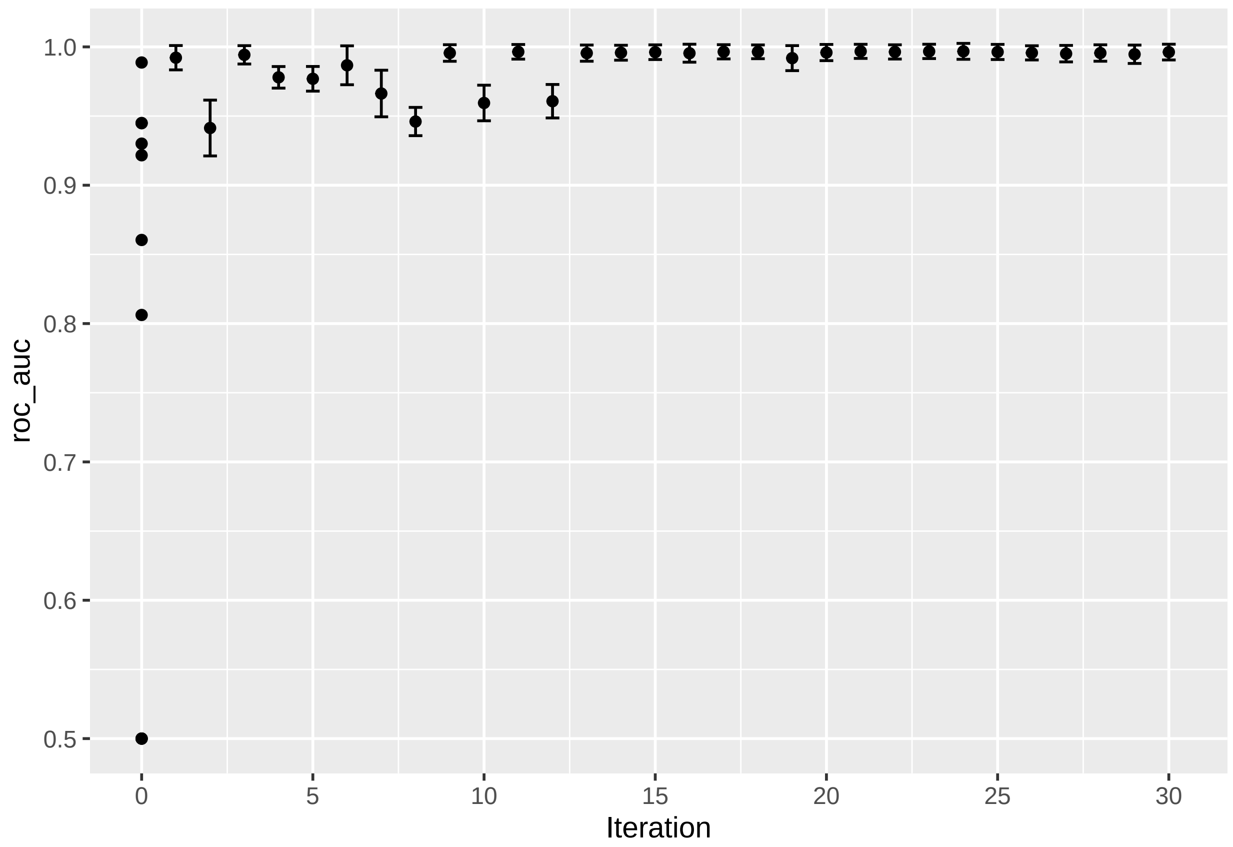
```r
## XGBoost model specifications -------

xgboost_model <-
  parsnip::boost_tree(
    trees = 1000,
    tree_depth = tune(), min_n = tune(),
    loss_reduction = 0,
    sample_size = tune(), mtry = 100,
    learn_rate = tune(),
    stop_iter = 10
  ) %>%
  set_engine("xgboost", nthread = 10) %>%
  set_mode("classification")

## Grid specification ---------

xgboost_params <-
  dials::parameters(
    tree_depth(),
    min_n(),
    #loss_reduction(),
    sample_size = sample_prop(),
    #finalize(mtry(), training(df_split)),
    learn_rate()
  )

xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 10
  )

## Define the workflow ----------

xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
  add_formula(death_event ~ .)

## Tune starting grid -----------

xgboost_ini <- tune::tune_grid(
  object = xgboost_wf,
  resamples = df_cv_folds,
  grid = xgboost_grid,
  control = tune::control_grid(verbose = T)
)

## Tune the model -----------

xgb_bo <-
```

```
  xgboost_wf %>%
  tune_bayes(
    resamples = df_cv_folds,
    metrics = metric_set(roc_auc),
    initial = xgboost_ini,
    param_info = xgboost_params,
    iter = 30,
    control = control_bayes(no_improve = 10,verbose = T)
  )

# xgb_bo %>% collect_metrics()

xgb_bo %>%
  tune::show_best(metric = "roc_auc") %>%
  knitr::kable()
```

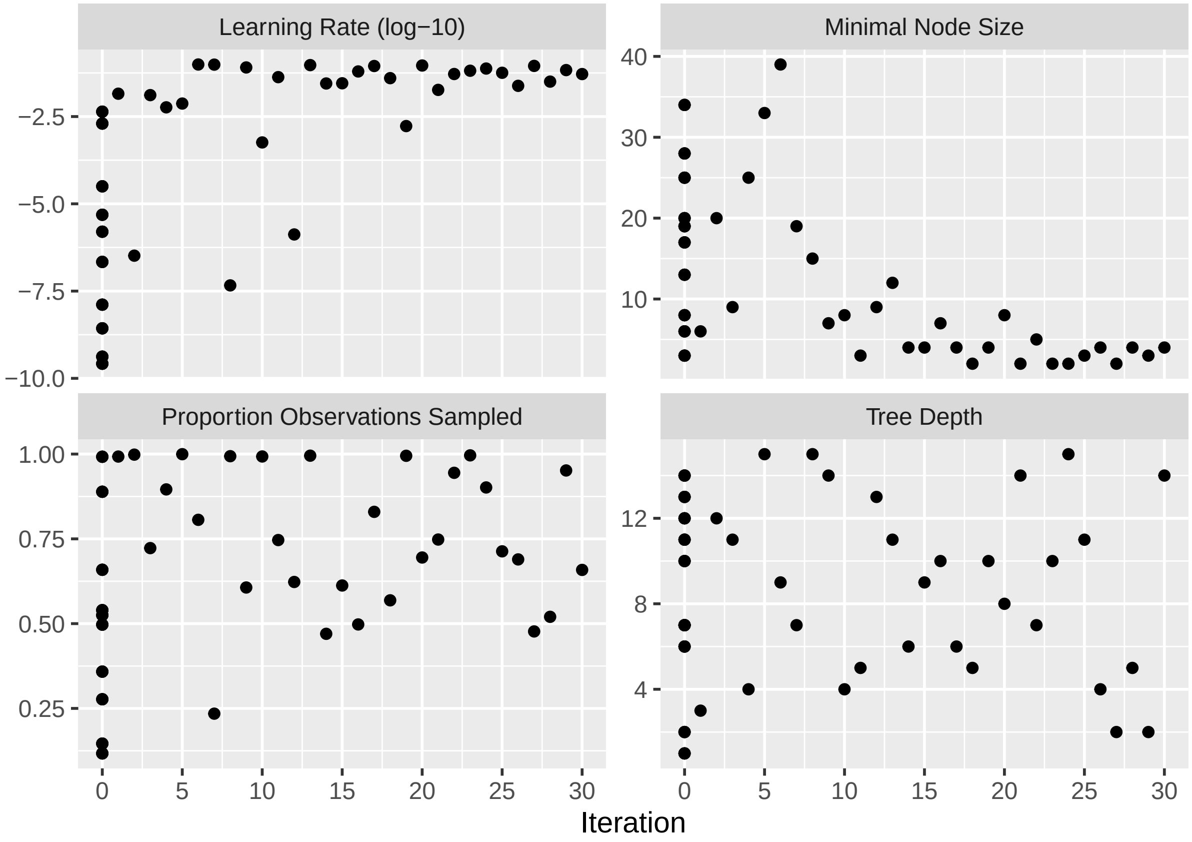
tree_depth	min_n	sample_size	learn_rate	.metric	.estimator	mean	n	std_err	.config	.iter
15	2	0.9015248	0.0750713	roc_auc	binary	0.9967794	3	0.0018091	Iter24	24
14	2	0.7479719	0.0183618	roc_auc	binary	0.9967671	3	0.0016010	Iter21	21
10	2	0.9962009	0.0649591	roc_auc	binary	0.9967282	3	0.0016253	Iter23	23
6	4	0.8296386	0.0887519	roc_auc	binary	0.9964523	3	0.0016160	Iter17	17
5	2	0.5687312	0.0398773	roc_auc	binary	0.9964411	3	0.0015645	Iter18	18

Checking hyperparameter searching paths:

```
xgb_bo %>%
  autoplot(type = "performance")
```

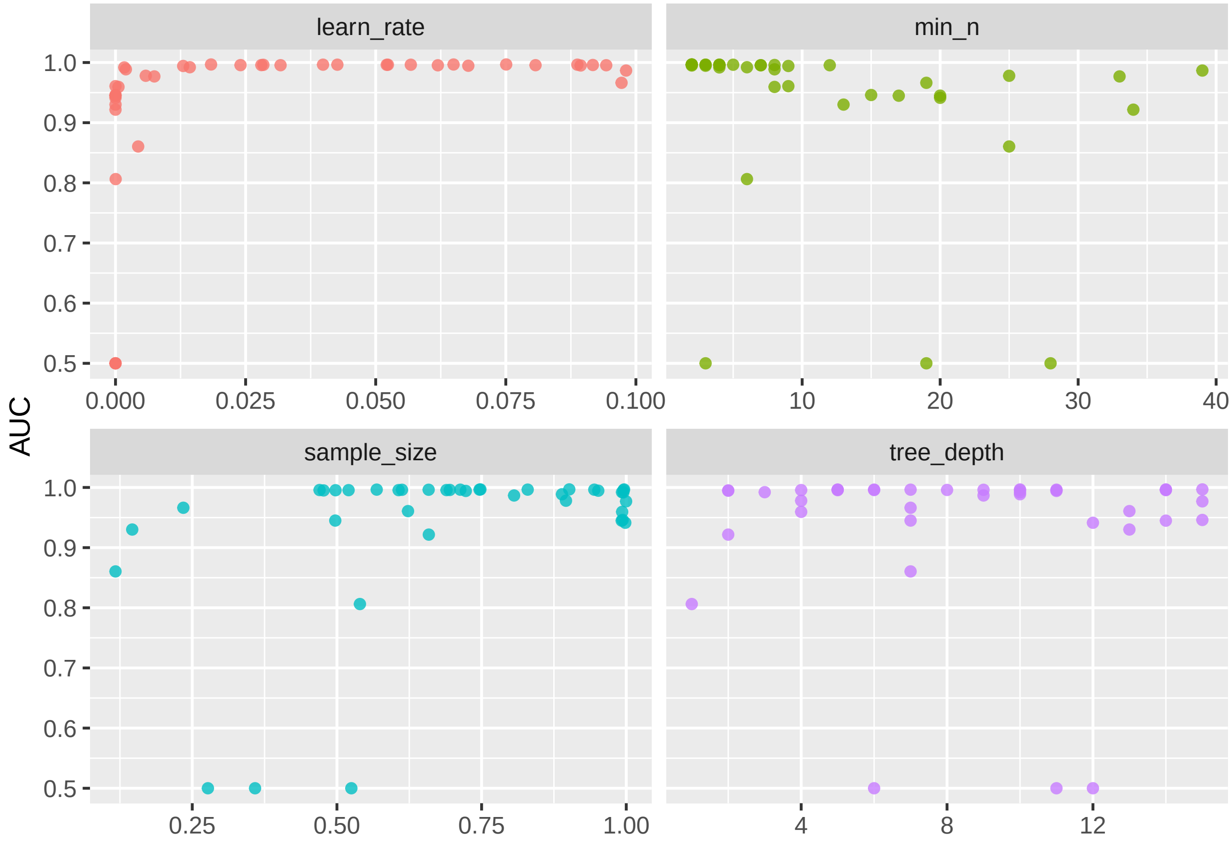


```r
xgb_bo %>%
  autoplot(type = "parameters")
```



And checking the hyperparameters values evaluated in terms of out-of-sample AUC ROC:

```
xgb_bo %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc") %>%
  dplyr::select(mean, tree_depth:learn_rate) %>%
  pivot_longer(tree_depth:learn_rate,
               values_to = "value",
               names_to = "parameter"
  ) %>%
  ggplot(aes(value, mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE) +
  facet_wrap(~parameter, scales = "free_x") +
  labs(x = NULL, y = "AUC")
```



The best hyperparameters found:

```
xgboost_best_params <- xgb_bo %>%
  tune::select_best()

xgboost_best_params
```

```
## # A tibble: 1 x 5
##   tree_depth min_n sample_size learn_rate .config
##        <int> <int>       <dbl>      <dbl> <chr>
## 1         15     2       0.902     0.0751 Iter24
```

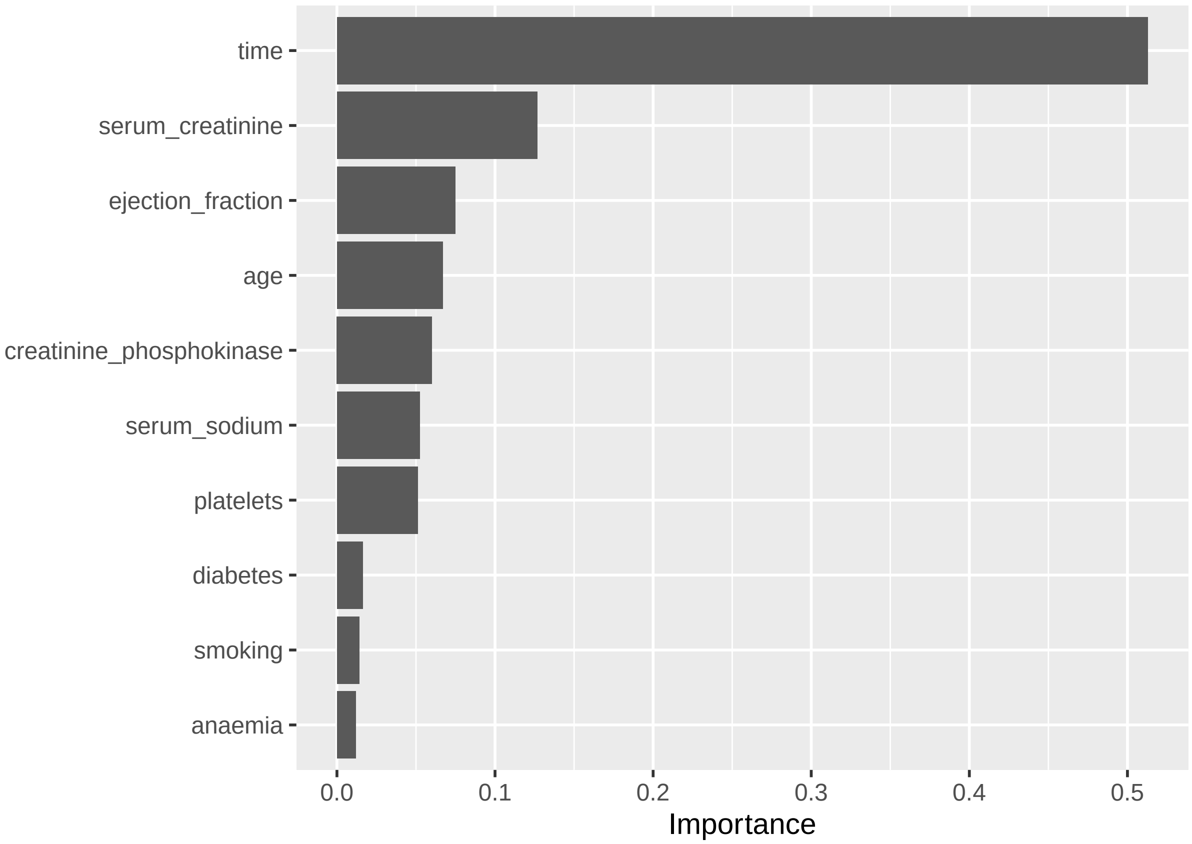
Final model:

```
xgboost_model %>%
  finalize_model(xgboost_best_params) -> xgboost_model_final

final_xgb <- finalize_workflow(
  xgboost_wf,
  xgboost_best_params
)

final_xgb
```

```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## death_event ~ .
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = 100
##   trees = 1000
##   min_n = 2
##   tree_depth = 15
##   learn_rate = 0.0750712716112979
##   loss_reduction = 0
##   sample_size = 0.901524762203745
##   stop_iter = 10
##
## Engine-Specific Arguments:
##   nthread = 10
##
## Computational engine: xgboost
```

Using vip package for variable importance:

```
final_xgb %>%
  fit(data = training(df_split)) %>%
  extract_fit_parsnip() %>%
  vip(geom = "col")
```

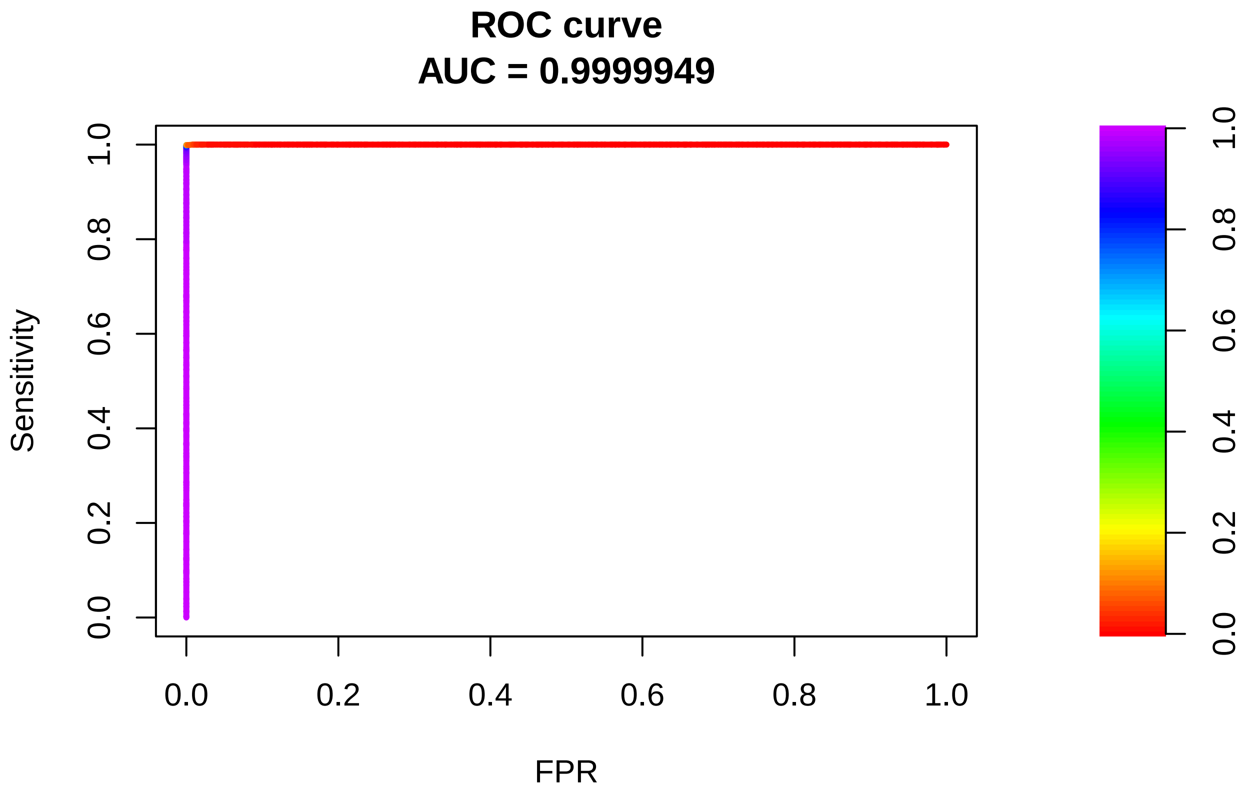


This will not account for variable importance as SHAP Values would do, only express variable importance in terms of "gain". As expected, "time" is strongly used to make a prediction.

Let's see how well training set entirely is predicted:

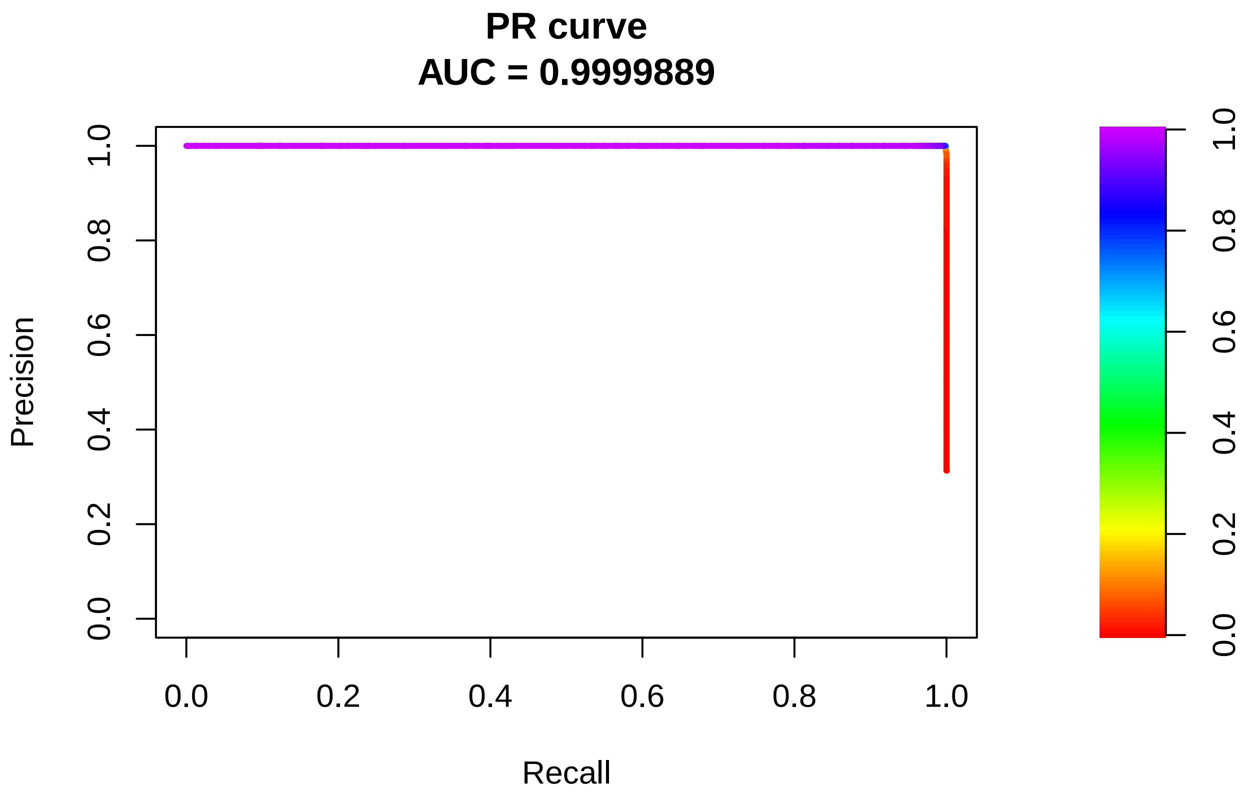
```
train_processed <- bake(preprocessing_recipe, new_data = training(df_split))

xgboost_model_fit <- xgboost_model_final %>%
  fit(
    formula = death_event ~ .,
    data    = train_processed
  )

train_prediction <- xgboost_model_fit %>%
  predict_classprob.model_fit(new_data = train_processed) %>%
  bind_cols(training(df_split))

train_prediction %>%
  dplyr::select(death_event,'1') %>%
  dplyr::rename(death = death_event, pred = '1') -> train_prediction2

xgb.roc <- roc.curve(weights.class0 = as.numeric(as.character(train_prediction2$death)), scores.class0 =

xgb.pr  <- pr.curve(weights.class0 = as.numeric(as.character(train_prediction2$death)), scores.class0 =

xgb.realiability <- reliability_diagramm(as.numeric(as.character(train_prediction2$death)),train_predic
```

```r
plot(xgb.roc)
```



We achieved AUC ROC of 0.999, which is nearly replicating the results in Kaggle's notebooks.

```r
plot(xgb.pr)
```

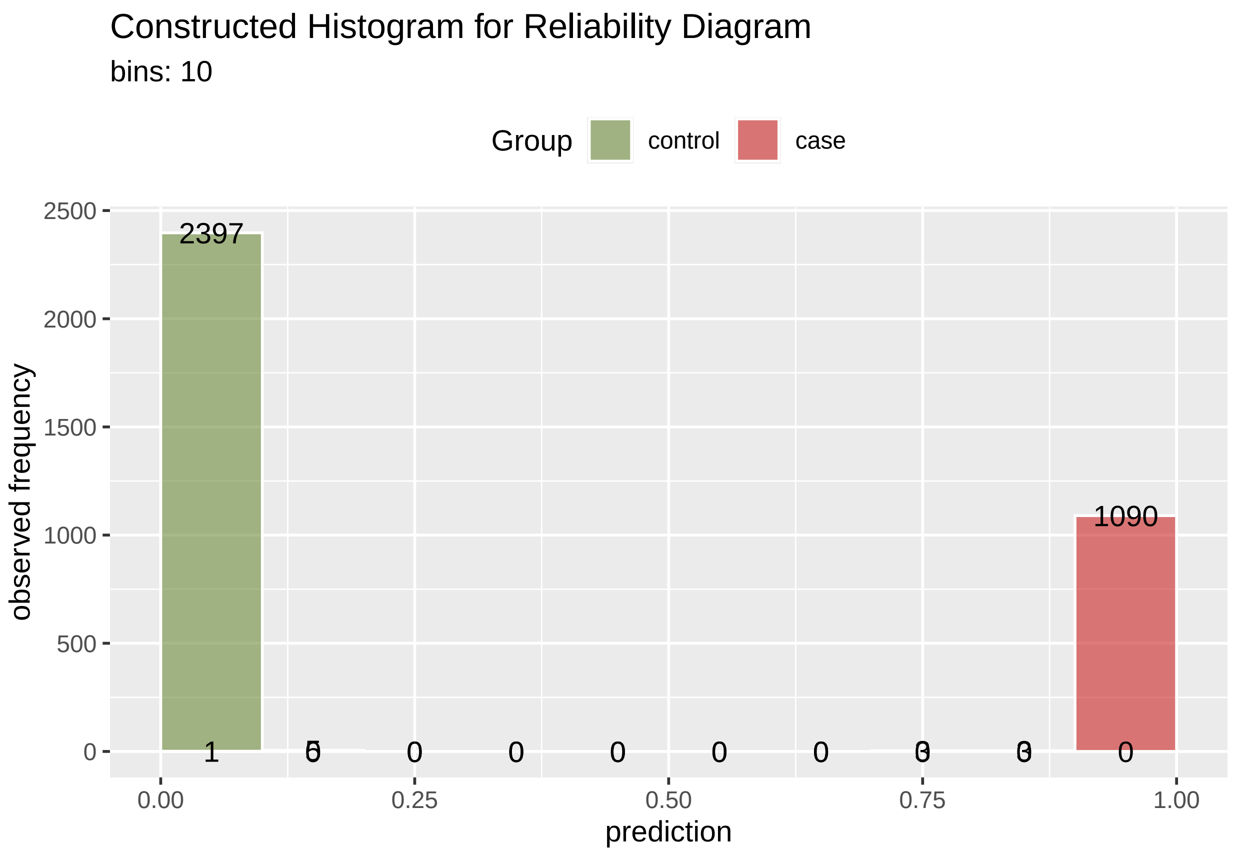


PR Curve is also suspiciously good

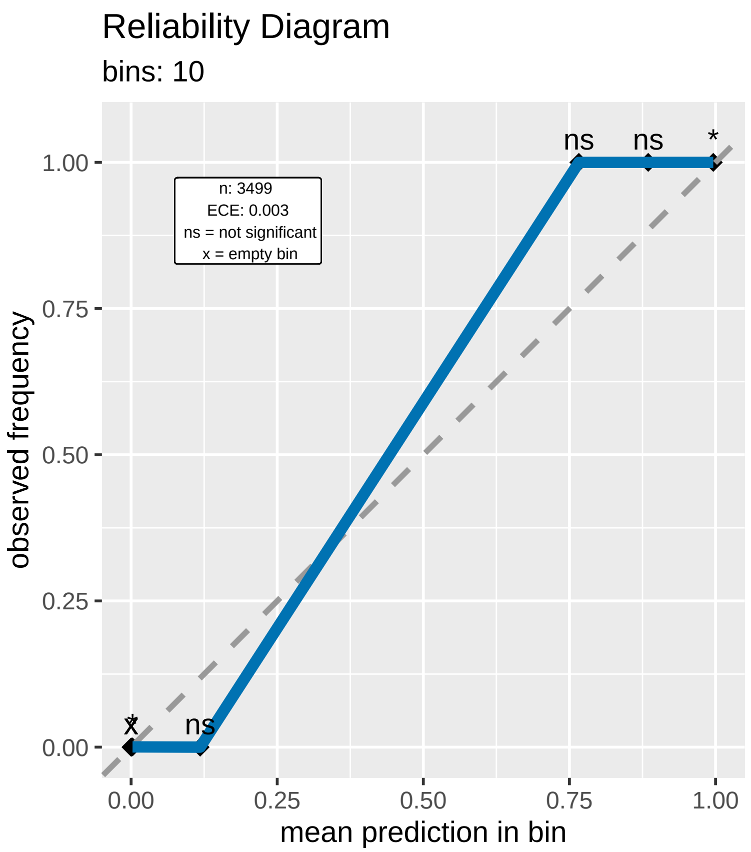
```
xgb.realiability
```

```
## $calibration_error
## $calibration_error$ECE_equal_width
## [1] 0.00304
##
## $calibration_error$MCE_equal_width
## [1] 0.00136
##
## $calibration_error$ECE_equal_freq
## [1] 0.00153
##
## $calibration_error$MCE_equal_freq
## [1] 0.00059
##
## $calibration_error$RMSE
## [1] 0.01985
##
## $calibration_error$CLE_class_1
## [1] 0.00566
##
## $calibration_error$CLE_class_0
## [1] 0.00262
##
```

```
## $calibration_error$brier
## [1] 0.00039
##
## $calibration_error$brier_class_1
## [1] 0.00033
##
## $calibration_error$brier_class_0
## [1] 6e-05
##
##
## $discrimination_error
## $discrimination_error$sens
## [1] 0.999
##
## $discrimination_error$spec
## [1] 1
##
## $discrimination_error$acc
## [1] 1
##
## $discrimination_error$ppv
## [1] 1
##
## $discrimination_error$npv
## [1] 1
##
## $discrimination_error$cutoff
## [1] 0.441
##
## $discrimination_error$auc
## [1] 1
##
##
## $rd_breaks
## [1] 10
##
## $histogram_plot
```



```
##
## $diagram_plot
```



```
##
## $mean_pred_per_bin
##  [1] 0.002407179 0.118146813 0.000000000 0.000000000 0.000000000 0.000000000
##  [7] 0.000000000 0.766122912 0.884784023 0.996116517
##
## $accuracy_per_bin
##  [1] 0.0004170142 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##  [6] 0.0000000000 0.0000000000 1.0000000000 1.0000000000 1.0000000000
##
## $freq_per_bin
##  [1] 0.6853386682 0.0014289797 0.0000000000 0.0000000000 0.0000000000
##  [6] 0.0000000000 0.0000000000 0.0008573878 0.0008573878 0.3115175765
##
## $sign
##  [1] "*"  "ns" "x"  "x"  "x"  "x"  "x"  "ns" "ns" "*"
```

We can see that probability seems well calibrated (small ECE and blue line near diagonal).

In terms of confusion matrix (with a probability cut-off in 0.5):

```
caret::confusionMatrix(train_prediction2$death
                       , factor(as.numeric(train_prediction2$pred > 0.5))
                       , positive = "1", mode = "everything")
```

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

```
##
##           Reference
## Prediction    0    1
##          0 2402    0
##          1    1 1096
##
##                Accuracy : 0.9997
##                  95% CI : (0.9984, 1)
##     No Information Rate : 0.6868
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9993
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 1.0000
##             Specificity : 0.9996
##          Pos Pred Value : 0.9991
##          Neg Pred Value : 1.0000
##               Precision : 0.9991
##                  Recall : 1.0000
##                      F1 : 0.9995
##              Prevalence : 0.3132
##          Detection Rate : 0.3132
##    Detection Prevalence : 0.3135
##       Balanced Accuracy : 0.9998
##
##        'Positive' Class : 1
##
```

We can see that only 1 case from training is miss-classified.

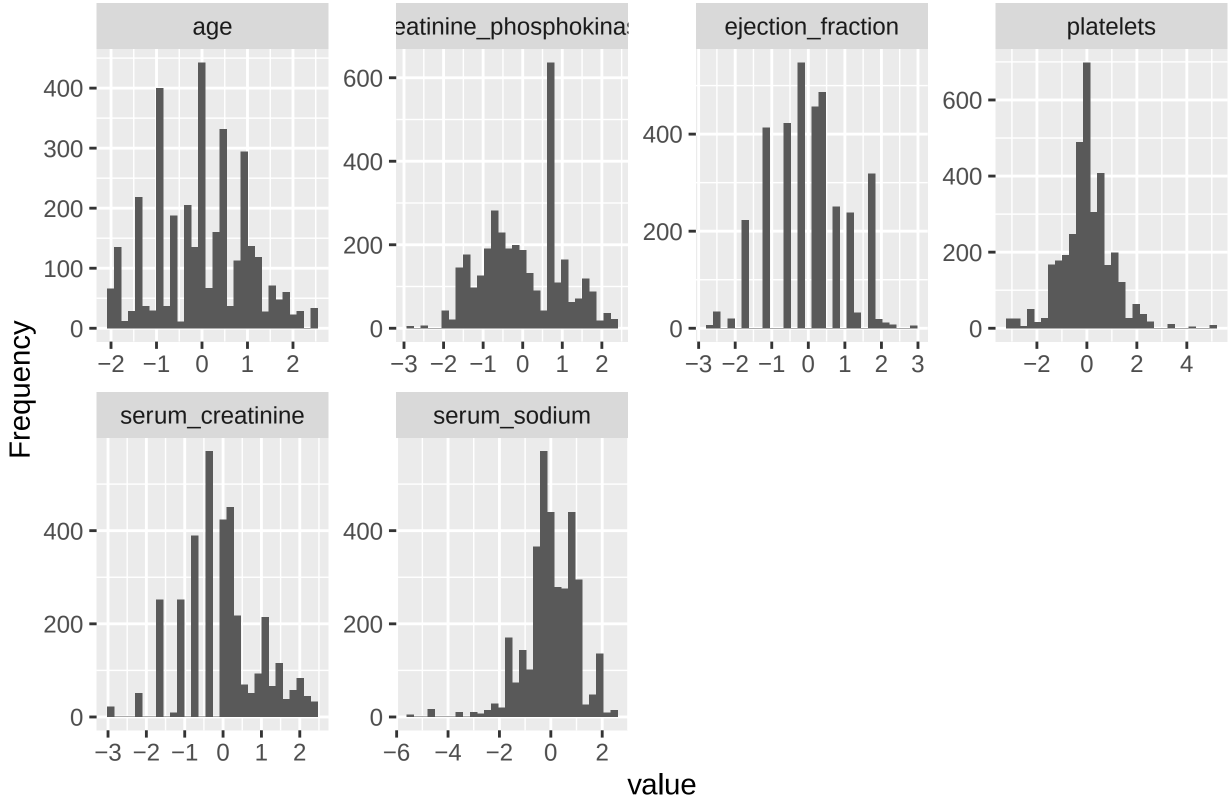
Model 2: XGBoost without time From now on, we will not longer use "time" attribute, as we will assume it leads to data leakage.

We will start replicating previous XGBoost and see how well we can predict. Later on, we will explore another models, as Logit (which would probably be evaluated earlier in an analysis starting from zero).

```
df %>%
  select(-time) -> df

set.seed(99)
df_split <- rsample::initial_split(
  df,
  prop = 0.7,
  strata = death_event
)
```

```
preprocessing_recipe <-
  recipes::recipe(death_event ~ ., data = training(df_split)) %>%
  recipes::step_nzv(all_nominal_predictors()) %>%     # Remove near zero variance
  step_YeoJohnson(all_numeric_predictors()) %>%       # Transform with Yeo-Johnson
  step_normalize(all_numeric_predictors()) %>%        # Normalize variables
  prep()
```

```
train <- recipes::bake(
    preprocessing_recipe,
    new_data = training(df_split)
  )

plot_histogram(train)
```



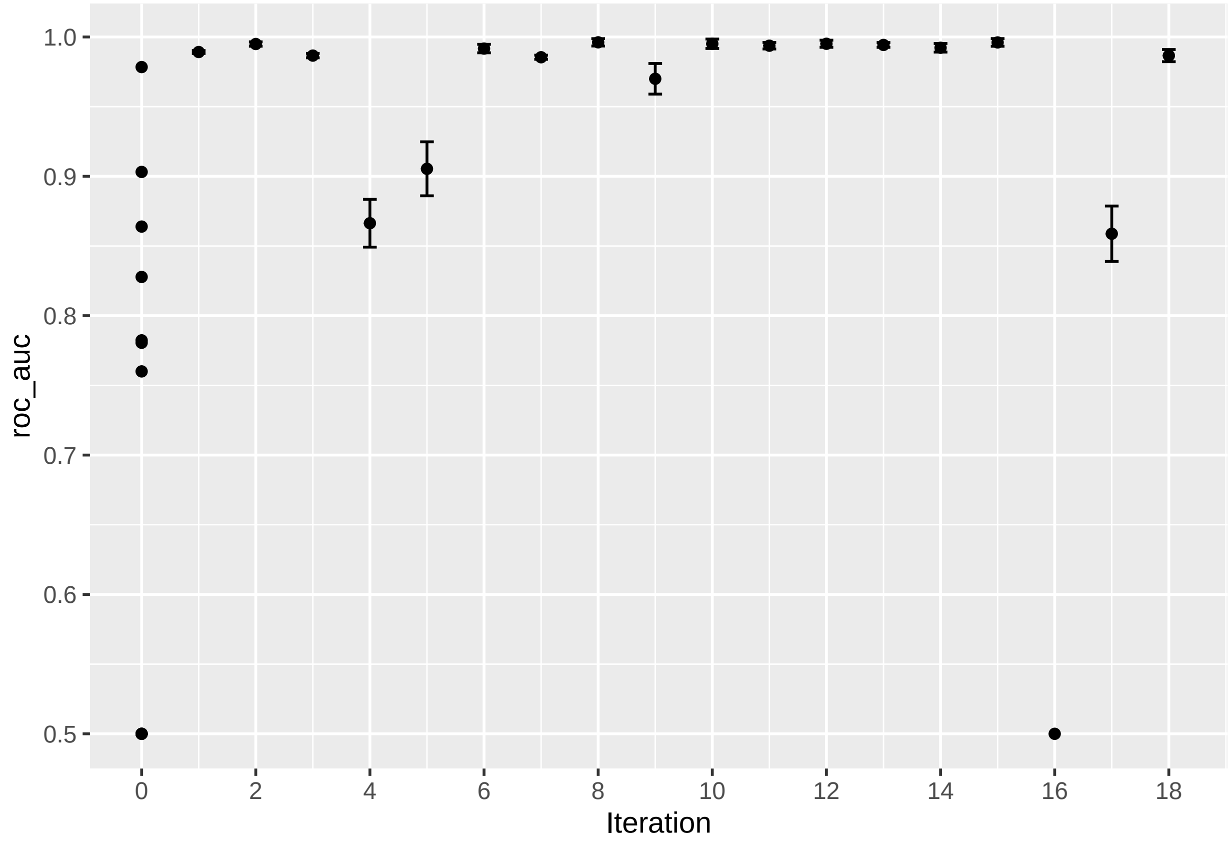
```
df_cv_folds <-
  recipes::bake(
    preprocessing_recipe,
    new_data = training(df_split)
  ) %>%
  rsample::vfold_cv(v = 3)
```

```
## XGBoost model specifications -------

xgboost_model <-
  parsnip::boost_tree(
    trees = 1000,
    tree_depth = tune(), min_n = tune(),
    loss_reduction = 0,
    sample_size = tune(), mtry = 100,
    learn_rate = tune(),
    stop_iter = 10
  ) %>%
  set_engine("xgboost", nthread = 10) %>%
  set_mode("classification")

## Grid specification ---------

xgboost_params <-
  dials::parameters(
    tree_depth(),
    min_n(),
    #loss_reduction(),
    sample_size = sample_prop(),
    #finalize(mtry(), training(df_split)),
    learn_rate()
  )

xgboost_grid <-
  dials::grid_max_entropy(
    xgboost_params,
    size = 10
  )

## Define the workflow ----------

xgboost_wf <-
  workflows::workflow() %>%
  add_model(xgboost_model) %>%
  add_formula(death_event ~ .)

## Tune starting grid -----------

xgboost_ini <- tune::tune_grid(
  object = xgboost_wf,
  resamples = df_cv_folds,
  grid = xgboost_grid,
  control = tune::control_grid(verbose = T)
)

## Tune the model -----------

xgb_bo <-
  xgboost_wf %>%
  tune_bayes(
    resamples = df_cv_folds,
    metrics = metric_set(roc_auc),
    initial = xgboost_ini,
    param_info = xgboost_params,
    iter = 30,
    control = control_bayes(no_improve = 10,verbose = T)
  )

# xgb_bo %>% collect_metrics()

xgb_bo %>%
```

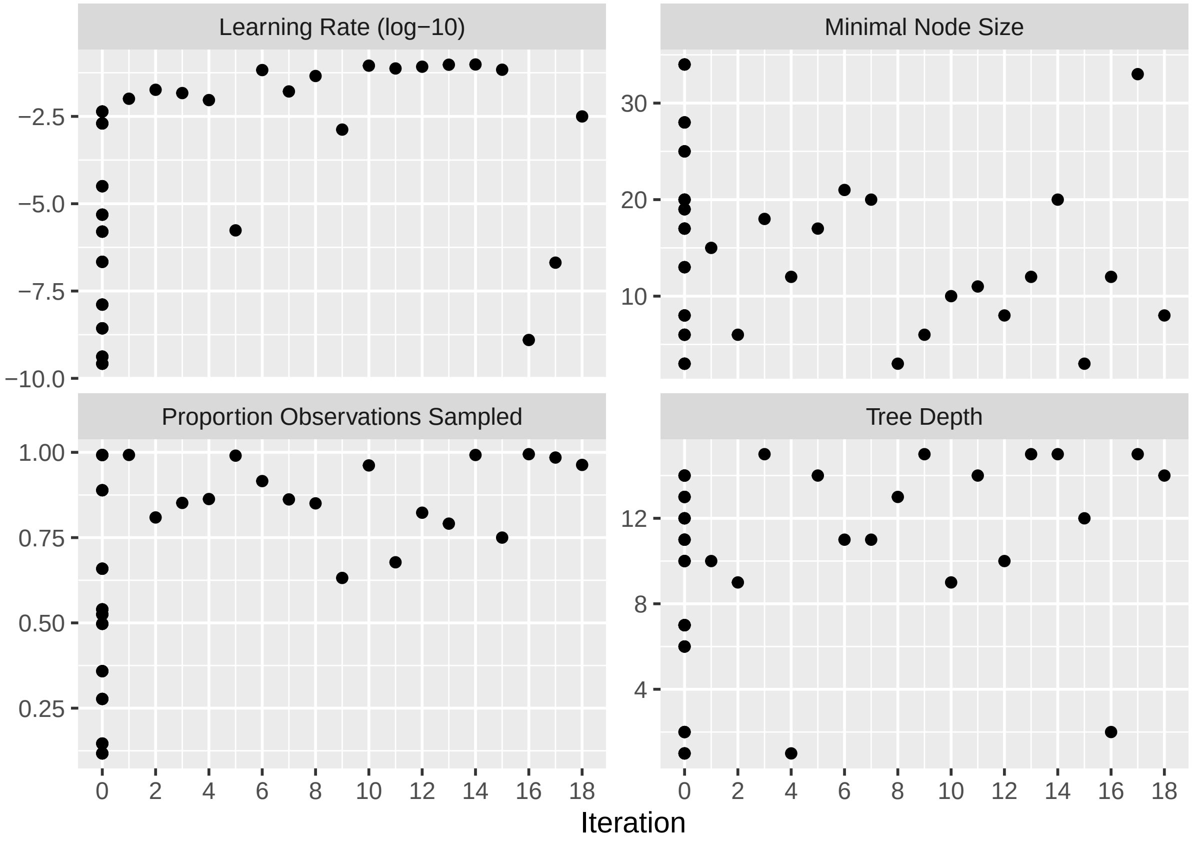
```r
  tune::show_best(metric = "roc_auc") %>%
  knitr::kable()
```

tree_depth	min_n	sample_size	learn_rate	.metric	.estimator	mean	n	std_err	.config	.iter
13	3	0.8503696	0.0453723	roc_auc	binary	0.9961507	3	0.0008329	Iter8	8
12	3	0.7500425	0.0686176	roc_auc	binary	0.9960747	3	0.0008679	Iter15	15
9	10	0.9616494	0.0897341	roc_auc	binary	0.9951070	3	0.0010704	Iter10	10
10	8	0.8229952	0.0836976	roc_auc	binary	0.9951003	3	0.0008027	Iter12	12
9	6	0.8090736	0.0182934	roc_auc	binary	0.9949402	3	0.0004955	Iter2	2

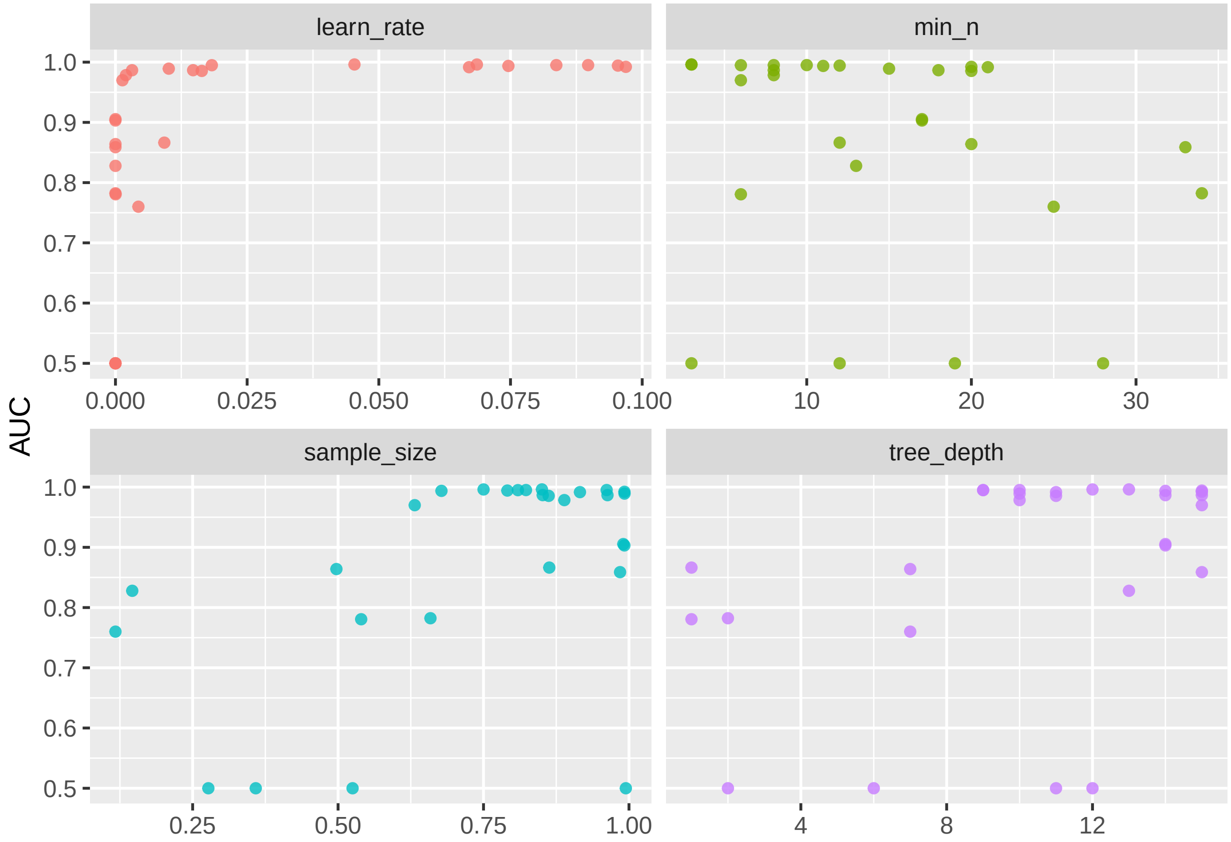
```r
xgb_bo %>%
  autoplot(type = "performance")
```



```r
xgb_bo %>%
  autoplot(type = "parameters")
```



```
xgb_bo %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc") %>%
  dplyr::select(mean, tree_depth:learn_rate) %>%
  pivot_longer(tree_depth:learn_rate,
               values_to = "value",
               names_to = "parameter"
  ) %>%
  ggplot(aes(value, mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE) +
  facet_wrap(~parameter, scales = "free_x") +
  labs(x = NULL, y = "AUC")
```



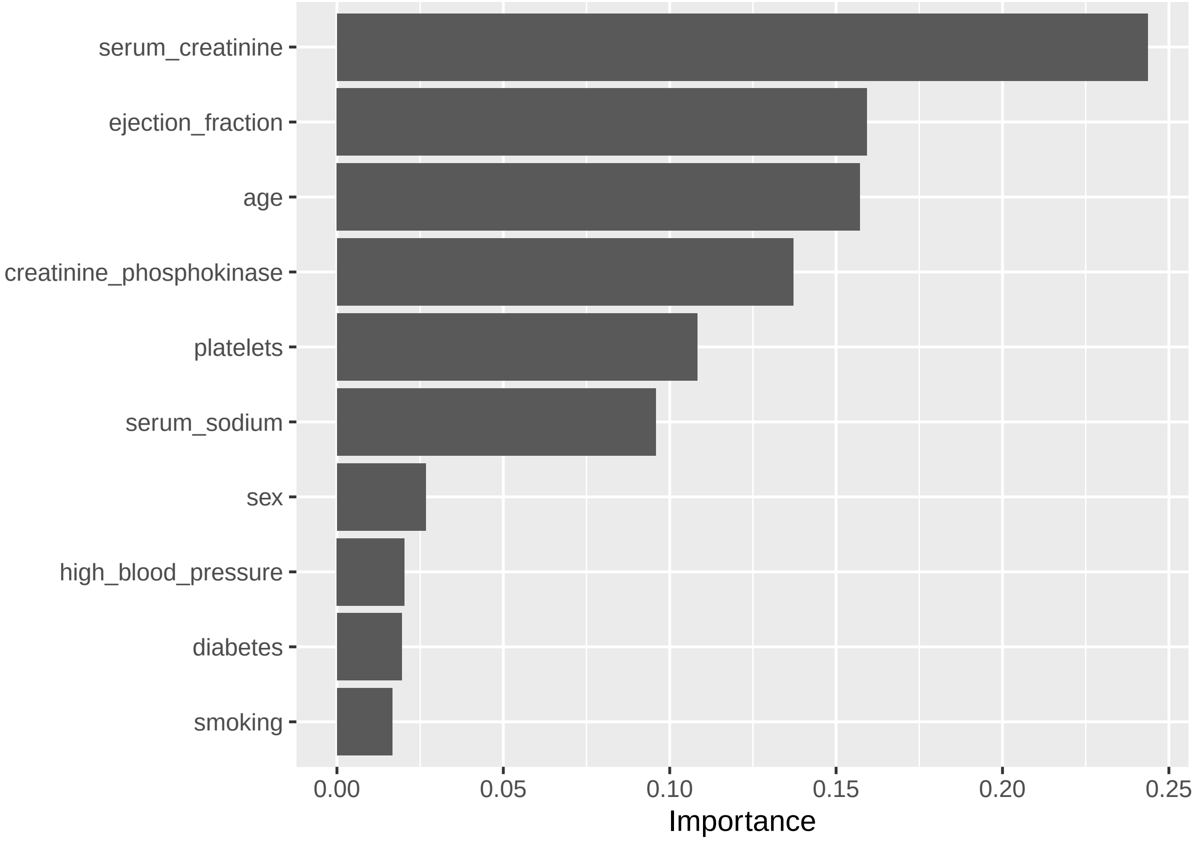
```
xgboost_best_params <- xgb_bo %>%
  tune::select_best()

xgboost_best_params
```

```
## # A tibble: 1 x 5
##   tree_depth min_n sample_size learn_rate .config
##        <int> <int>       <dbl>      <dbl> <chr>
## 1         13     3       0.850     0.0454 Iter8
```

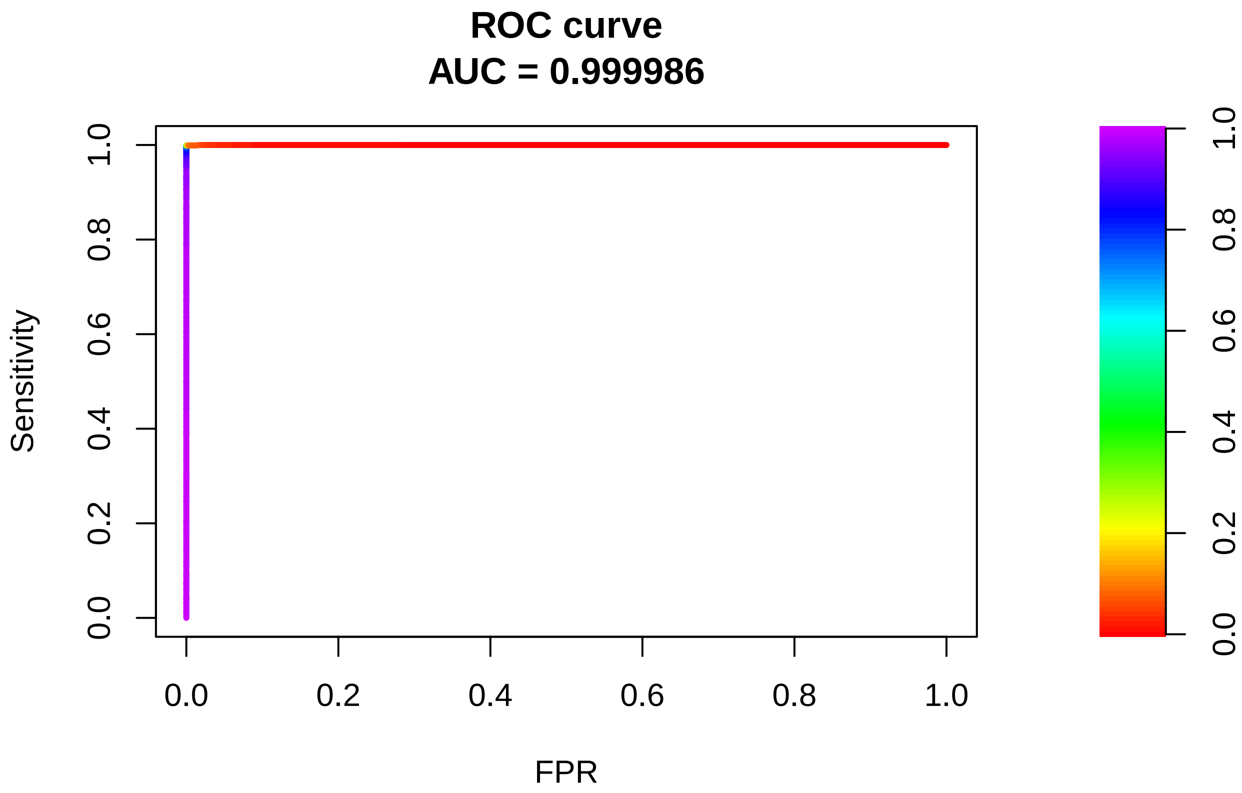
```
xgboost_model %>%
  finalize_model(xgboost_best_params) -> xgboost_model_final

final_xgb <- finalize_workflow(
  xgboost_wf,
  xgboost_best_params
)

final_xgb
```

```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## death_event ~ .
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = 100
##   trees = 1000
##   min_n = 3
##   tree_depth = 13
##   learn_rate = 0.0453722717093075
##   loss_reduction = 0
##   sample_size = 0.850369603180215
##   stop_iter = 10
##
## Engine-Specific Arguments:
##   nthread = 10
##
## Computational engine: xgboost
```

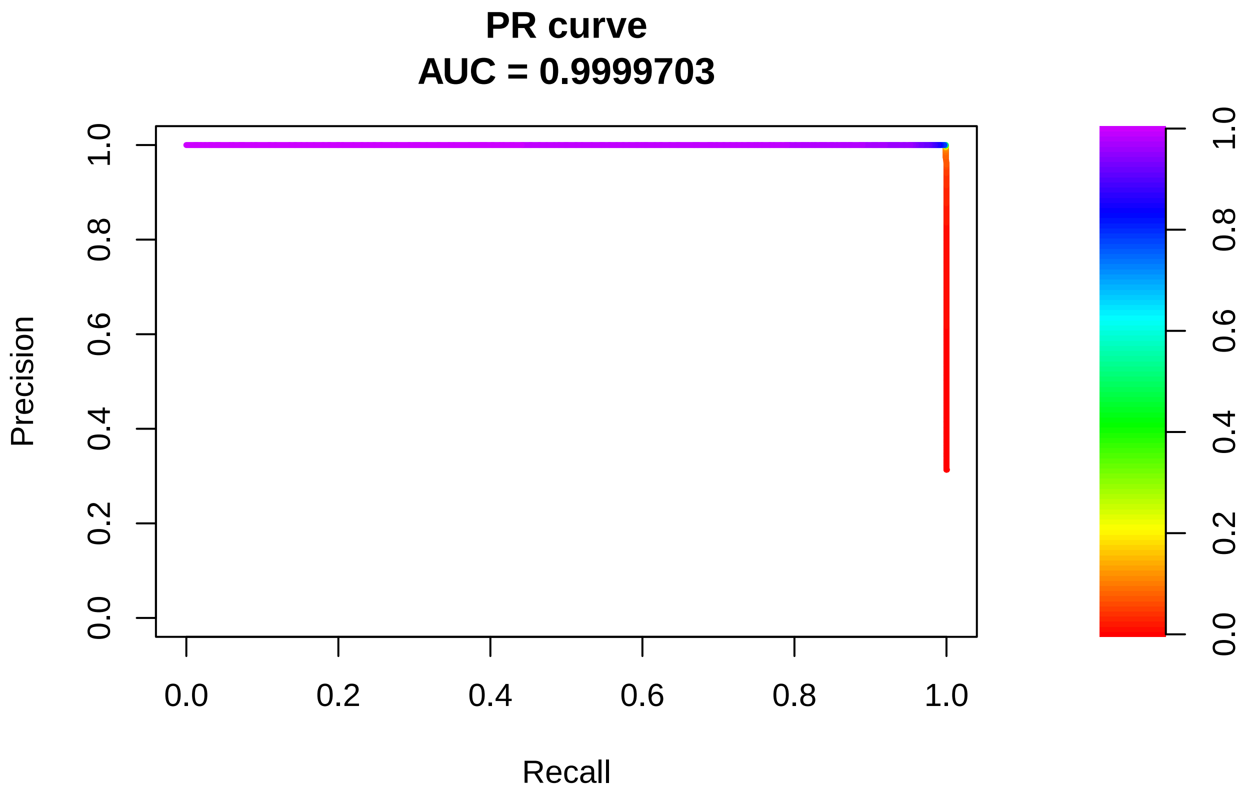
```
final_xgb %>%
  fit(data = training(df_split)) %>%
  extract_fit_parsnip() %>%
  vip(geom = "col")
```



```r
train_processed <- bake(preprocessing_recipe, new_data = training(df_split))

xgboost_model_fit <- xgboost_model_final %>%
  fit(
    formula = death_event ~ .,
    data    = train_processed
  )

train_prediction <- xgboost_model_fit %>%
  predict_classprob.model_fit(new_data = train_processed) %>%
  bind_cols(training(df_split))

train_prediction %>%
  dplyr::select(death_event,'1') %>%
  dplyr::rename(death = death_event, pred = '1') -> train_prediction2

xgb.roc <- roc.curve(weights.class0 = as.numeric(as.character(train_prediction2$death)), scores.class0 =

xgb.pr  <- pr.curve(weights.class0 = as.numeric(as.character(train_prediction2$death)), scores.class0 =

xgb.realiability <- reliability_diagramm(as.numeric(as.character(train_prediction2$death)),train_predic

plot(xgb.roc)
```



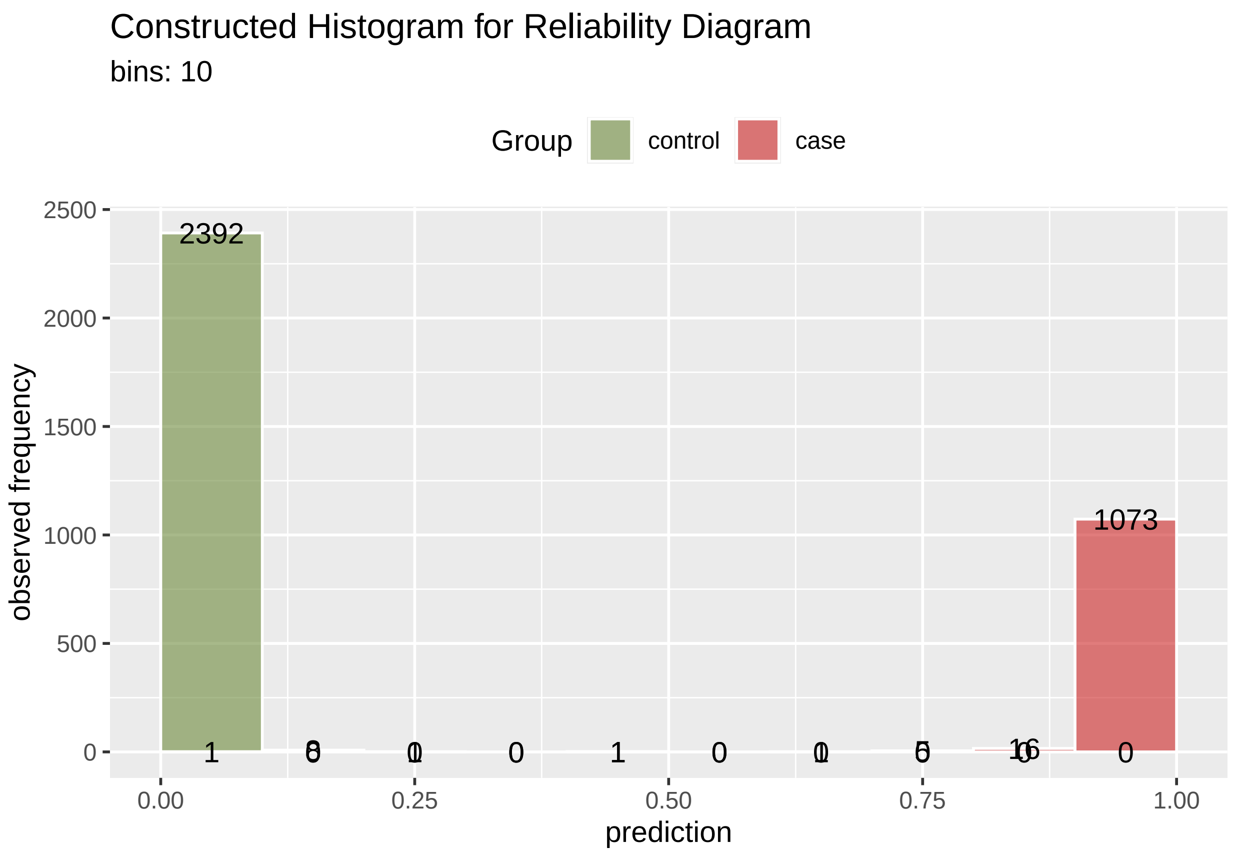
```r
plot(xgb.pr)
```



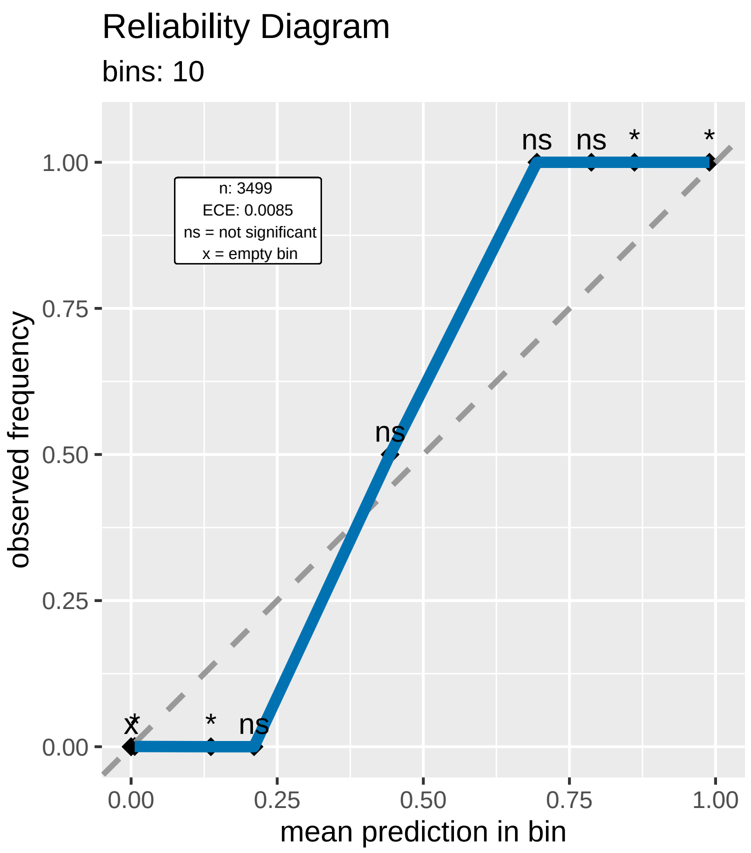
```r
xgb.realiability
```

```
## $calibration_error
## $calibration_error$ECE_equal_width
## [1] 0.00851
##
## $calibration_error$MCE_equal_width
## [1] 0.00389
##
## $calibration_error$ECE_equal_freq
## [1] 0.00534
##
## $calibration_error$MCE_equal_freq
## [1] 0.00187
##
## $calibration_error$RMSE
## [1] 0.02883
##
## $calibration_error$CLE_class_1
## [1] 0.01478
##
## $calibration_error$CLE_class_0
## [1] 0.00677
```

```
##
## $calibration_error$brier
## [1] 0.00083
##
## $calibration_error$brier_class_1
## [1] 6e-04
##
## $calibration_error$brier_class_0
## [1] 0.00023
##
##
## $discrimination_error
## $discrimination_error$sens
## [1] 0.999
##
## $discrimination_error$spec
## [1] 1
##
## $discrimination_error$acc
## [1] 1
##
## $discrimination_error$ppv
## [1] 1
##
## $discrimination_error$npv
## [1] 1
##
## $discrimination_error$cutoff
## [1] 0.443
##
## $discrimination_error$auc
## [1] 1
##
##
## $rd_breaks
## [1] 10
##
## $histogram_plot
```



```
##
## $diagram_plot
```



```
##
## $mean_pred_per_bin
##  [1] 0.006108278 0.136534579 0.210349441 0.000000000 0.443126708 0.000000000
##  [7] 0.694325238 0.787434497 0.861257725 0.989590050
##
## $accuracy_per_bin
##  [1] 0.0004178855 0.0000000000 0.0000000000 0.0000000000 0.5000000000
##  [6] 0.0000000000 1.0000000000 1.0000000000 1.0000000000 1.0000000000
##
## $freq_per_bin
##  [1] 0.6839096885 0.0022863675 0.0002857959 0.0000000000 0.0005715919
##  [6] 0.0000000000 0.0002857959 0.0014289797 0.0045727351 0.3066590454
##
## $sign
##  [1] "*"  "*"  "ns" "x"  "ns" "x"  "ns" "ns" "*"  "*"
```

```
caret::confusionMatrix(train_prediction2$death
                       , factor(as.numeric(train_prediction2$pred > 0.5))
                       , positive = "1", mode = "everything")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 2402    0
##          1    2 1095
##
##                Accuracy : 0.9994
##                  95% CI : (0.9979, 0.9999)
##     No Information Rate : 0.6871
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9987
##
##  Mcnemar's Test P-Value : 0.4795
##
##             Sensitivity : 1.0000
##             Specificity : 0.9992
##          Pos Pred Value : 0.9982
##          Neg Pred Value : 1.0000
##               Precision : 0.9982
##                  Recall : 1.0000
##                      F1 : 0.9991
##              Prevalence : 0.3129
##          Detection Rate : 0.3129
##    Detection Prevalence : 0.3135
##       Balanced Accuracy : 0.9996
##
##        'Positive' Class : 1
##
```

We achieved similar performance without "time" attribute.

We can see in reference 5 and 6 that using the 2 most promising values, serum creatinine and ejection fraction, they achieved with Gradient Boosting techniques to AUC ROC of 0.792 and using all attributes 0.754. This is strange as using all attributes in a machine learning algorithm that has incorporated attribute selection and has hyperparameters to deal with regularization, should lead to a better result than using just some of those attributes. Another observation is that "age" seems to be a more important attribute than "ejection fraction". These results were achieved in a different dataset (they had available 299 patients while Kaggle dataset is 5.000 patients, both data sets have the same attributes availables). Even if they are from the same distribution, just the fact that we have more data to calibrate might lead to the difference observed in the fitting.

Let's see if we could achieve similar results with a simpler model.

Model 3: Logit Unlike tree based models, logit is influenced by redundance between independents attributes (multicollinearity). So, one important step would be attribute selection or regularization (to penalize the use of some attributes). We will use regularization and calibrate the type of regularization (mixture between L1 and L2 regularization) and the amount of regularization (penalty) as hyperparameter tuning. We will use glmnet engine to have both hyperparameter available, as base glm doesn't.

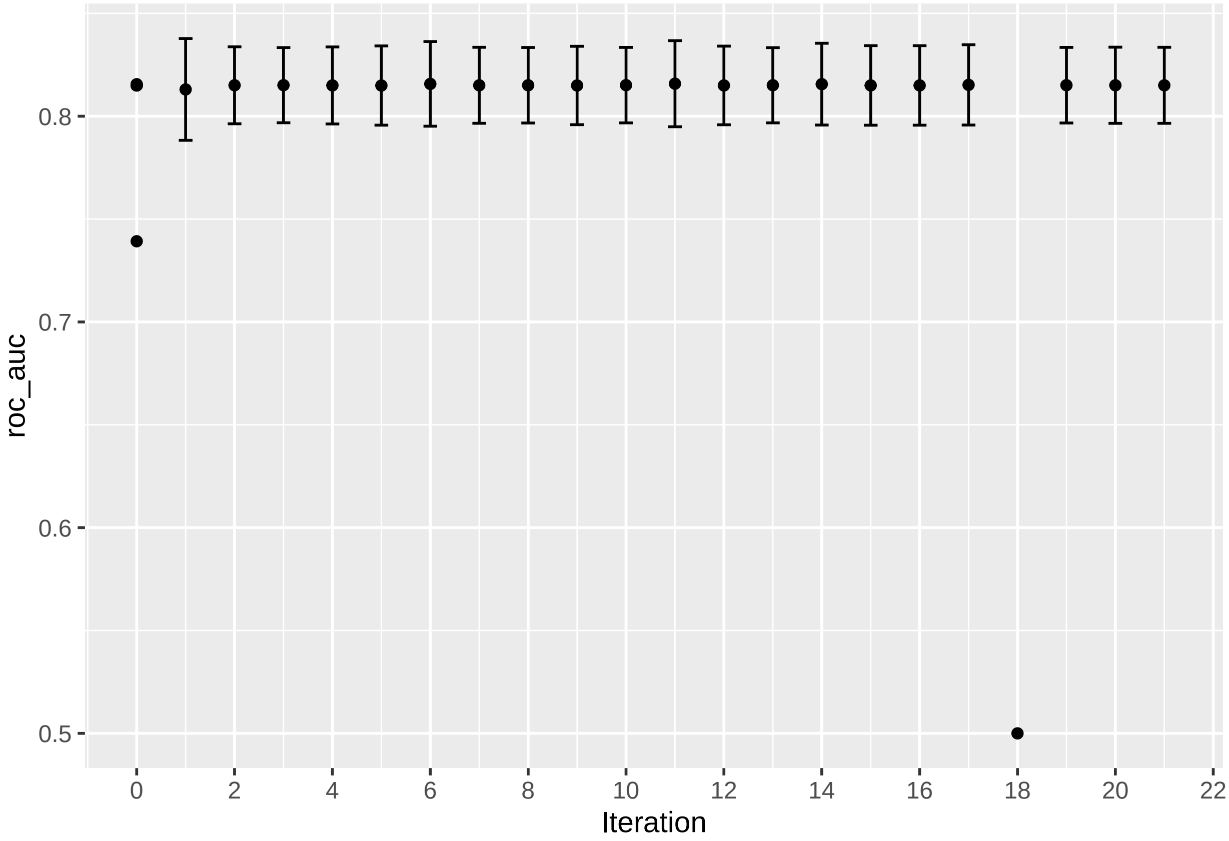
```
## Logit model specifications -------

logit_model <- logistic_reg(
  penalty = tune(),
  mixture = tune()) %>%
  set_engine("glmnet") %>%
  set_mode("classification")
```

```r
## Grid specification ---------

logit_params <-
  dials::parameters(
    mixture(),
    penalty()
  )

xgboost_grid <-
  dials::grid_max_entropy(
    logit_params,
    size = 10
  )

## Define the workflow ----------

logit_wf <-
  workflows::workflow() %>%
  add_model(logit_model) %>%
  add_formula(death_event ~ .)

## Tune starting grid -----------

logit_ini <- tune::tune_grid(
  object = logit_wf,
  resamples = df_cv_folds,
  grid = xgboost_grid,
  control = tune::control_grid(verbose = T)
)

## Tune the model -----------

logit_bo <-
  logit_wf %>%
  tune_bayes(
    resamples = df_cv_folds,
    metrics = metric_set(roc_auc),
    initial = logit_ini,
    param_info = logit_params,
    iter = 30,
    control = control_bayes(no_improve = 10,verbose = T)
  )

# xgb_bo %>% collect_metrics()

logit_bo %>%
  tune::show_best(metric = "roc_auc") %>%
  knitr::kable()
```

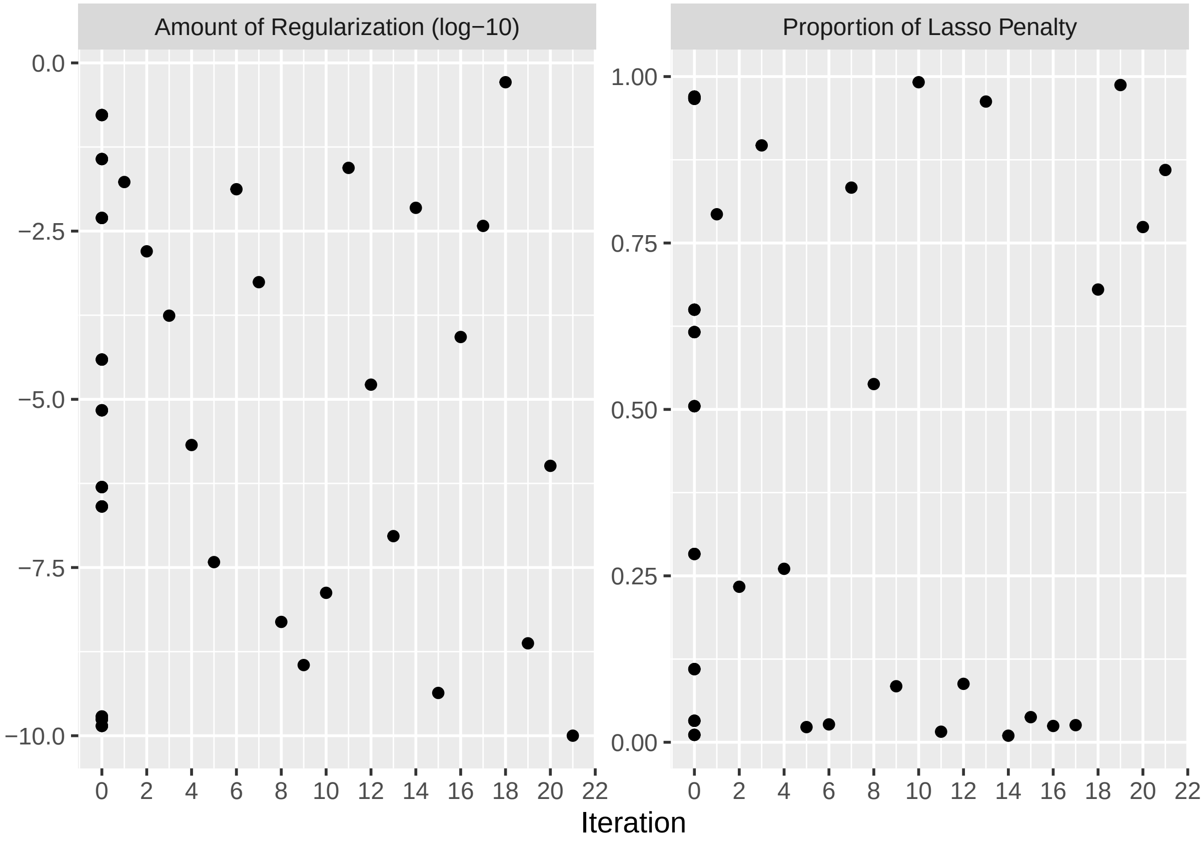
mixture	penalty	.metric	.estimator	mean	n	std_err	.config	.iter
0.0158632	0.0275206	roc_auc	binary	0.8158141	3	0.0065747	Iter11	11
0.0268543	0.0132603	roc_auc	binary	0.8157182	3	0.0064640	Iter6	6

mixture	penalty	.metric	.estimator	mean	n	std_err	.config	.iter
0.0098964	0.0070183	roc_auc	binary	0.8155939	3	0.0062435	Iter14	14
0.6161724	0.0049626	roc_auc	binary	0.8155599	3	0.0057837	Preprocessor1_Model06	0
0.1099041	0.0372552	roc_auc	binary	0.8153245	3	0.0067698	Preprocessor1_Model03	0

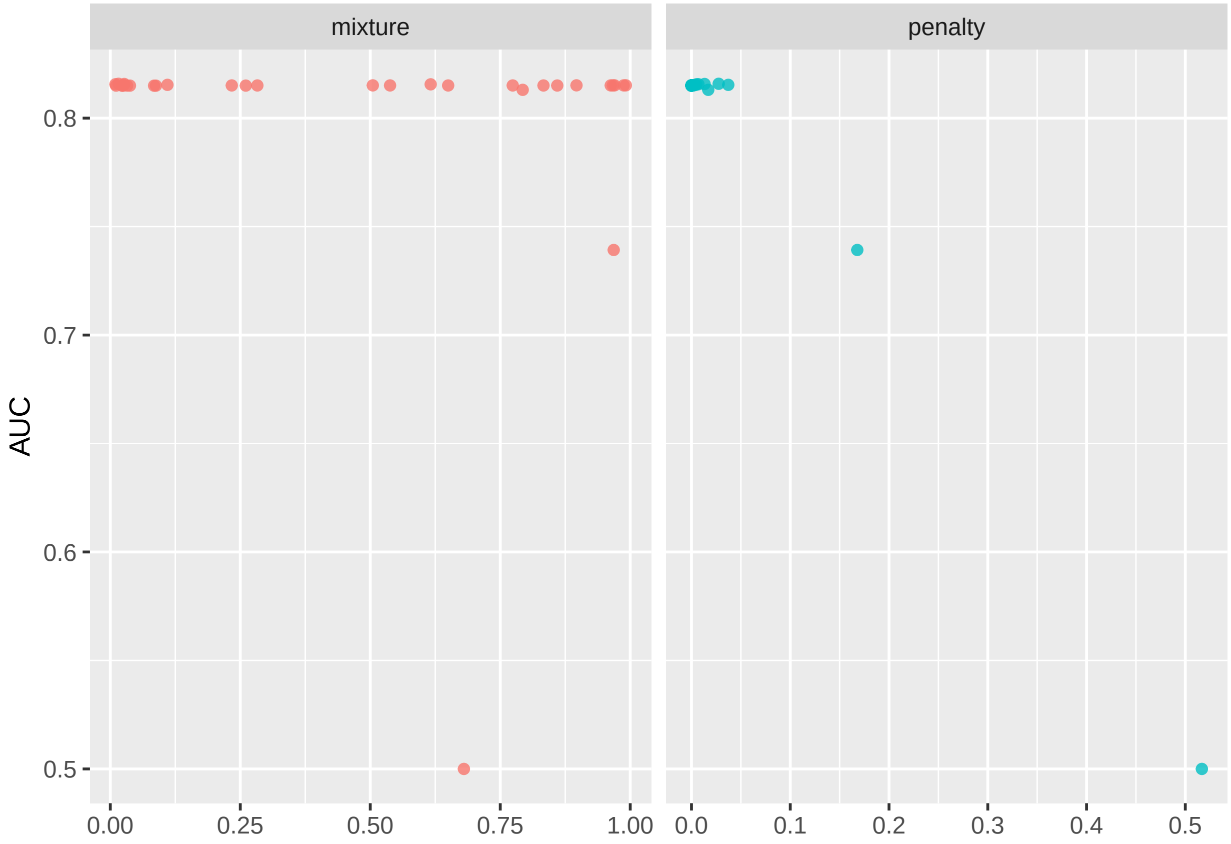
```r
logit_bo %>%
  autoplot(type = "performance")
```



```r
logit_bo %>%
  autoplot(type = "parameters")
```



```
logit_bo %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc") %>%
  dplyr::select(mean, mixture:penalty) %>%
  pivot_longer(mixture:penalty,
               values_to = "value",
               names_to = "parameter"
  ) %>%
  ggplot(aes(value, mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE) +
  facet_wrap(~parameter, scales = "free_x") +
  labs(x = NULL, y = "AUC")
```



```
logit_best_params <- logit_bo %>%
  tune::select_best()

logit_best_params
```

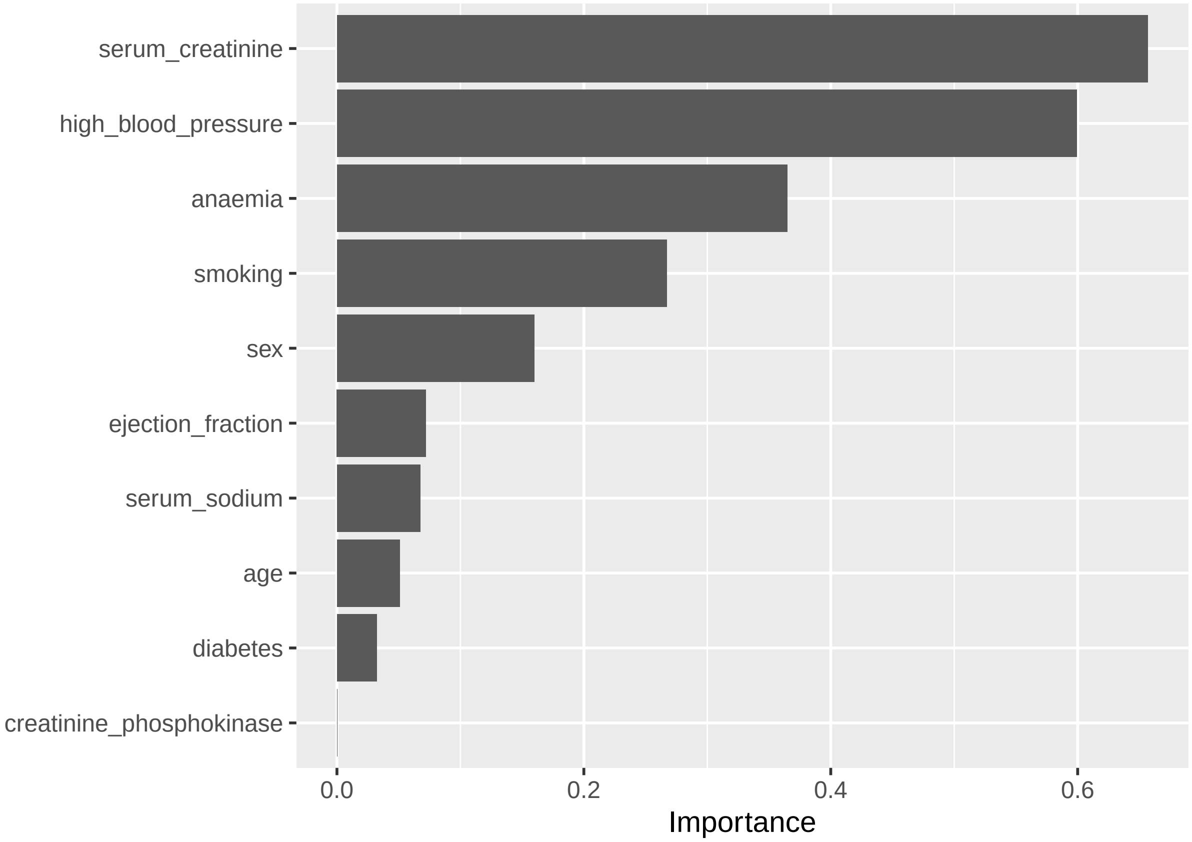
```
## # A tibble: 1 x 3
##   mixture penalty .config
##     <dbl>   <dbl> <chr>
## 1  0.0159  0.0275 Iter11
```

Best result achieved consider both L1 and L2 regularization (approximately equal) with a very small penalty.

```
logit_model %>%
  finalize_model(logit_best_params) -> logit_model_final

final_logit <- finalize_workflow(
  logit_wf,
  logit_best_params
)

final_logit
```

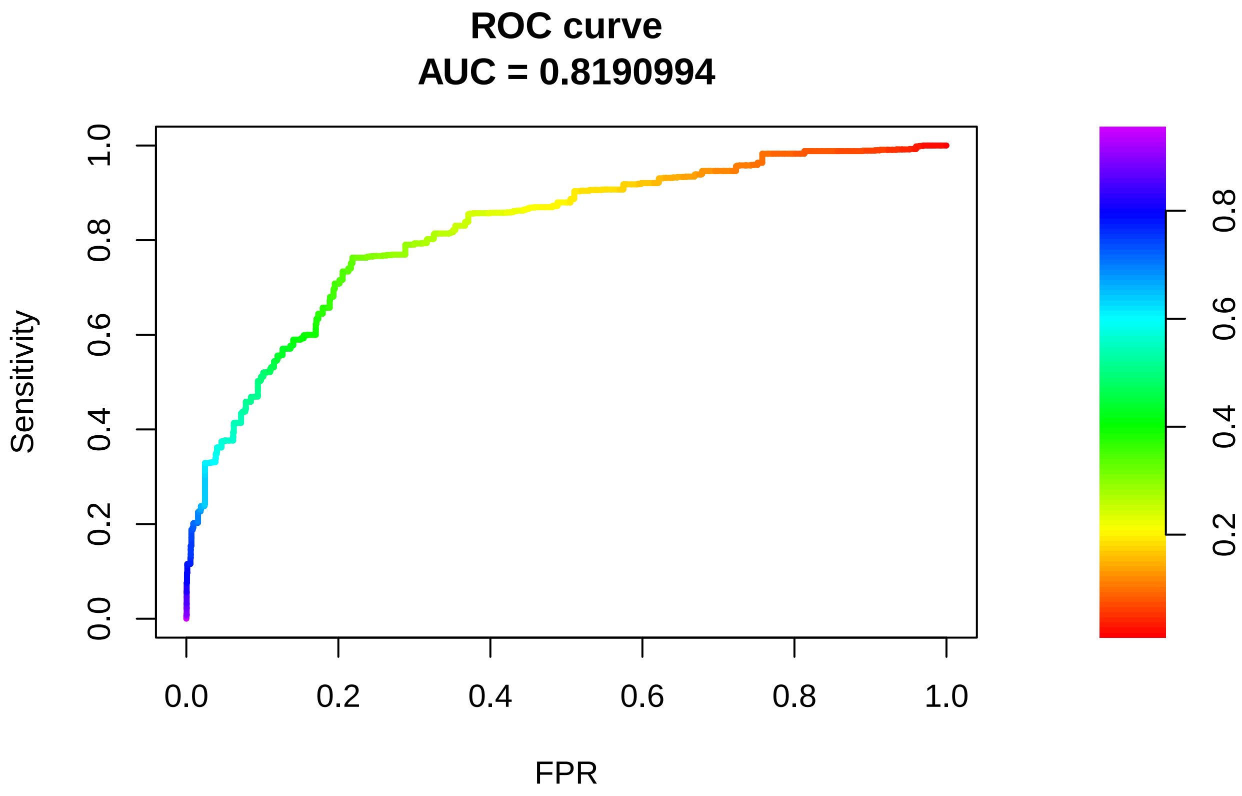
```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: logistic_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## death_event ~ .
##
## -- Model -----------------------------------------------------------------------
## Logistic Regression Model Specification (classification)
##
## Main Arguments:
##   penalty = 0.0275205672343568
##   mixture = 0.0158631955775432
##
## Computational engine: glmnet
```

```
final_logit %>%
  fit(data = training(df_split)) %>%
  extract_fit_parsnip() %>%
  vip(geom = "col")
```

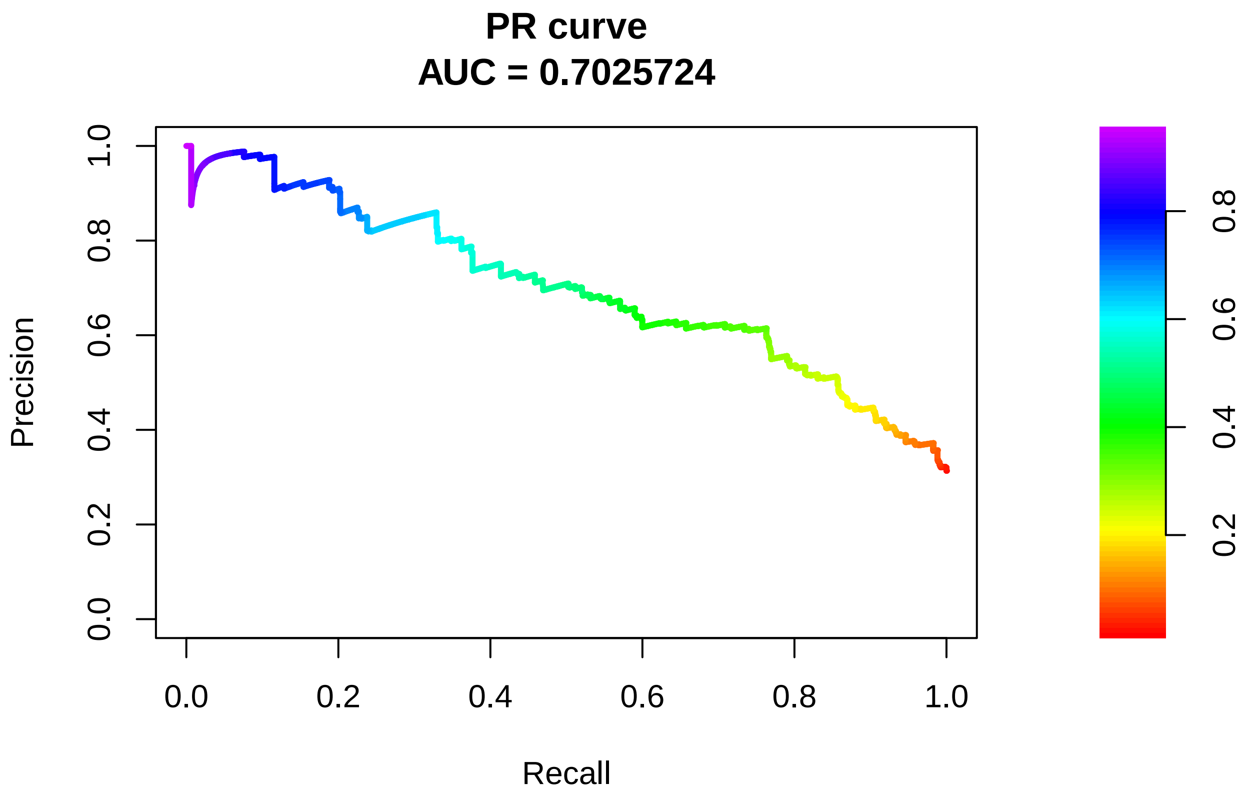


We can see that most important variable is serum creatinine in both XGBoost and Logit model, but the rest of the ranking differ a lot between them.

```
train_processed <- bake(preprocessing_recipe, new_data = training(df_split))

logit_model_fit <- logit_model_final %>%
  fit(
    formula = death_event ~ .,
    data    = train_processed
  )

train_prediction <- logit_model_fit %>%
  predict_classprob.model_fit(new_data = train_processed) %>%
  bind_cols(training(df_split))

train_prediction %>%
  dplyr::select(death_event,'1') %>%
  dplyr::rename(death = death_event, pred = '1') -> train_prediction2

xgb.roc <- roc.curve(weights.class0 = as.numeric(as.character(train_prediction2$death)), scores.class0 =

xgb.pr  <- pr.curve(weights.class0 = as.numeric(as.character(train_prediction2$death)), scores.class0 =

xgb.realiability <- reliability_diagramm(as.numeric(as.character(train_prediction2$death)),train_predic

plot(xgb.roc)
```

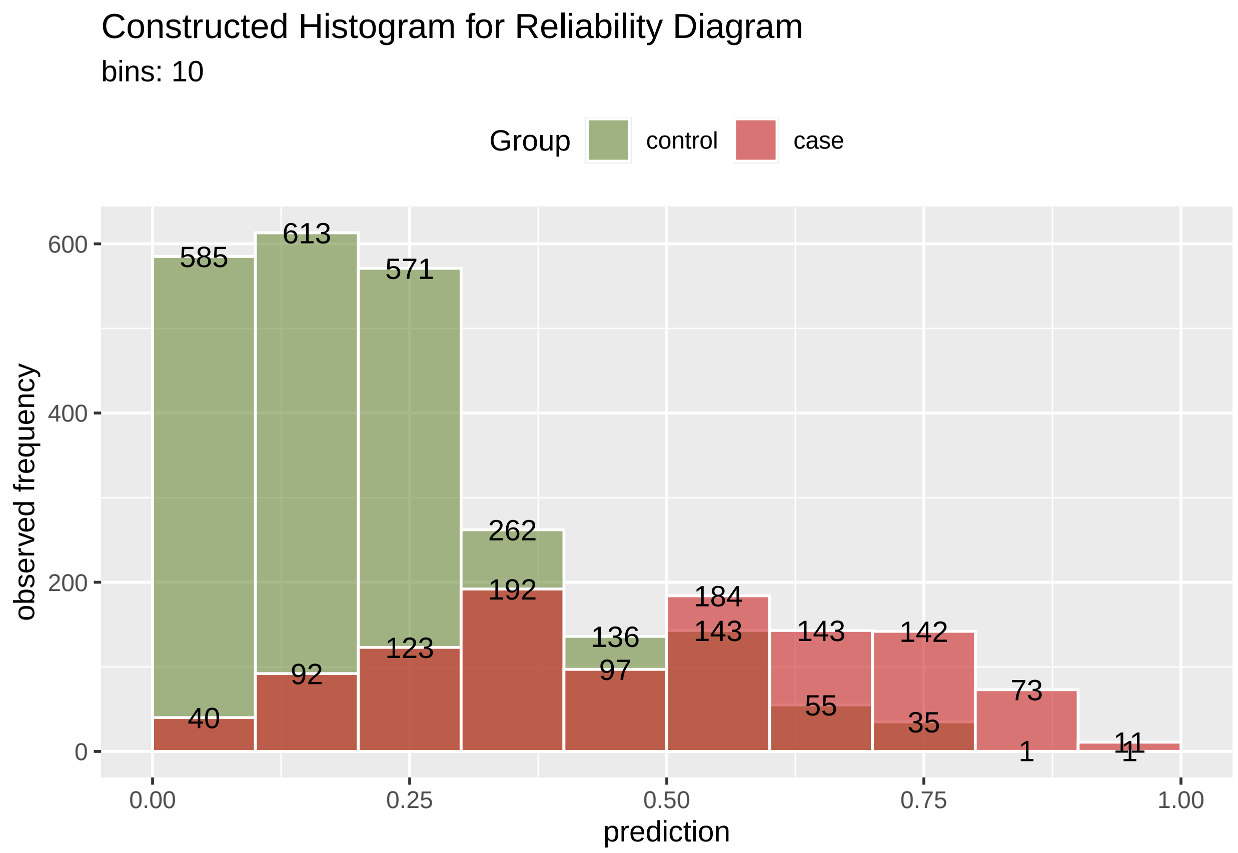


```
plot(xgb.pr)
```

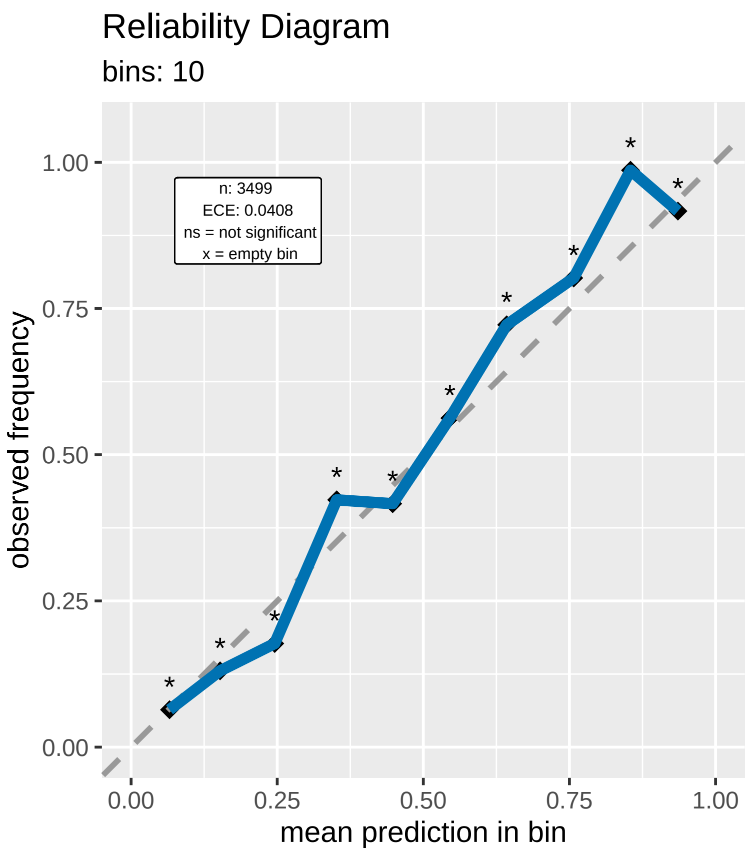


```
xgb.realiability
```

```
## $calibration_error
## $calibration_error$ECE_equal_width
## [1] 0.04082
##
## $calibration_error$MCE_equal_width
## [1] 0.01358
##
## $calibration_error$ECE_equal_freq
## [1] 0.05117
##
## $calibration_error$MCE_equal_freq
## [1] 0.01535
##
## $calibration_error$RMSE
## [1] 0.39179
##
## $calibration_error$CLE_class_1
## [1] 0.51157
##
## $calibration_error$CLE_class_0
## [1] 0.23363
##
## $calibration_error$brier
##      1
## 0.1535
##
## $calibration_error$brier_class_1
##    2403
## 0.09753
##
## $calibration_error$brier_class_0
##       1
## 0.05596
##
##
## $discrimination_error
## $discrimination_error$sens
## [1] 0.763
##
## $discrimination_error$spec
## [1] 0.781
##
## $discrimination_error$acc
## [1] 0.776
##
## $discrimination_error$ppv
## [1] 0.615
##
## $discrimination_error$npv
## [1] 0.878
##
## $discrimination_error$cutoff
## [1] 0.332
##
## $discrimination_error$auc
## [1] 0.819
##
##
## $rd_breaks
## [1] 10
##
## $histogram_plot
```



```
##
## $diagram_plot
```



```
##
## $mean_pred_per_bin
##  [1] 0.06572619 0.15222871 0.24570425 0.35179297 0.44750919 0.54541879
##  [7] 0.64262137 0.75729613 0.85451291 0.93560702
##
## $accuracy_per_bin
##  [1] 0.0640000 0.1304965 0.1772334 0.4229075 0.4163090 0.5626911 0.7222222
##  [8] 0.8022599 0.9864865 0.9166667
##
## $freq_per_bin
##  [1] 0.178622464 0.201486139 0.198342384 0.129751358 0.066590454 0.093455273
##  [7] 0.056587596 0.050585882 0.021148900 0.003429551
##
## $sign
##  [1] "*" "*" "*" "*" "*" "*" "*" "*" "*" "*"
```

In this case, is not evident which probability cut off we should consider. If we use 0.5, we have the following performance indicators:

```
caret::confusionMatrix(train_prediction2$death
                       , factor(as.numeric(train_prediction2$pred > 0.5))
                       , positive = "1", mode = "everything")
```

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

```
##           Reference
## Prediction    0    1
##          0 2167  235
##          1  544  553
##
##                Accuracy : 0.7774
##                  95% CI : (0.7632, 0.7911)
##     No Information Rate : 0.7748
##     P-Value [Acc > NIR] : 0.3667
##
##                   Kappa : 0.4399
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.7018
##             Specificity : 0.7993
##          Pos Pred Value : 0.5041
##          Neg Pred Value : 0.9022
##               Precision : 0.5041
##                  Recall : 0.7018
##                      F1 : 0.5867
##              Prevalence : 0.2252
##          Detection Rate : 0.1580
##    Detection Prevalence : 0.3135
##       Balanced Accuracy : 0.7506
##
##        'Positive' Class : 1
##
```

With a cut-off of 0.5, we achieve a Precision of 0.51 (51% of the people that the model says will not survive effectively won't) and a Recall of 0.69 (69% of people that will not survive is found by this model with this cut-off). We could see this as a point in PR curve. If this is good or bad, it will depend of how we will use the model.

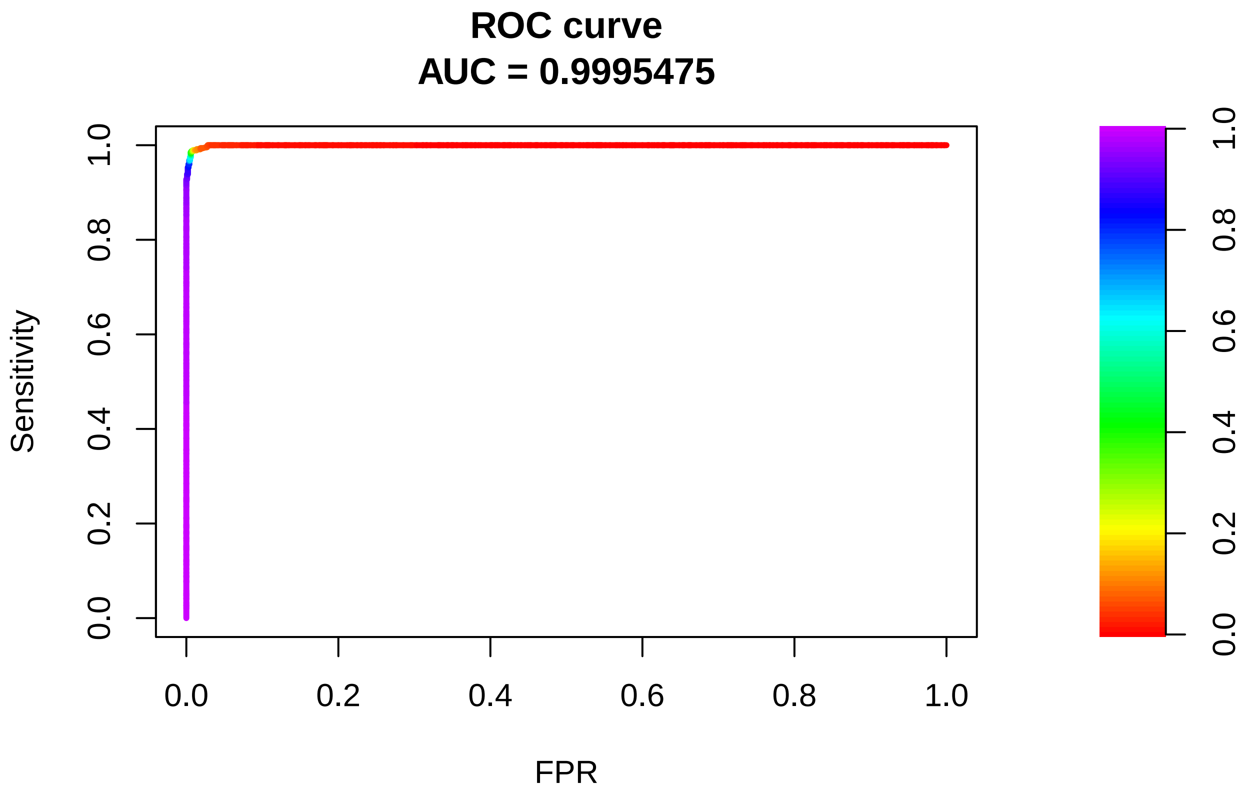
For example, if we will do some invasive treatment to those that the model point as high death risk, maybe we would want a higher Precision. Higher precision has the trade of that will make worse our recall. Let's say I want a Precision of at least 80% (only 20% of the people pointed by the model received an invasive treatment unnecessarily), then I would move in the blue area of PR curve (cut-off around 0.8), but I would expect a Recall around 20% (only 20% of the true deaths would be treated with this strategy). In the other hand, If I only want to check more often people with higher probability and this doesn't cost much as an error (If I check often someone that doesn´t need it's not a big deal), We could afford with the same model to lower Precision to achieve a higher Recall (yellow area represent cut-off around 0.2 and has expected Recall of 0.8 and Precision of 0.4).

This decision should be taken considering how the model will be used and exceeds the information given in Kaggle.

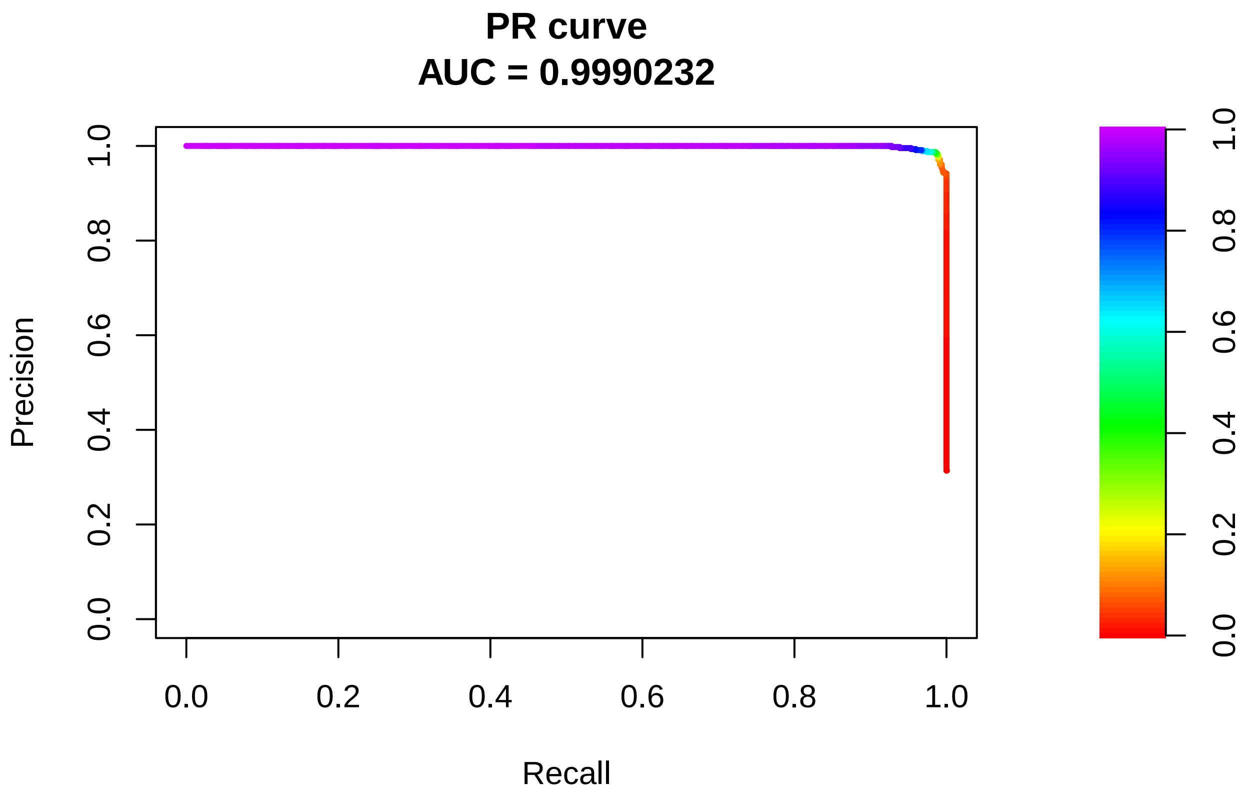
However, Logit model would be an option if we can afford to sacrifice prediction by explainability (If I need to know the input attribute combination that makes the model to predict that score). This is because Logit model, in the end, can be written in a formula and we can see attribute weights. If we can turn XGBoost in an explainable tool, then Logit wouldn't be our choice. There is when SHAP Values has something to say. With it, XGBoost can be no longer considered a black box tool.

Final Model Performance We choose model 2 as our final model. We need to see how the model perform in a test sample that was not used to train.

```r
test_processed <- bake(preprocessing_recipe, new_data = testing(df_split))

test_prediction <- xgboost_model_fit %>%
  predict_classprob.model_fit(new_data = test_processed) %>%
  bind_cols(testing(df_split))

test_prediction %>%
  dplyr::select(death_event,'1') %>%
  dplyr::rename(death = death_event, pred = '1') -> test_prediction2

xgb.roc <- roc.curve(weights.class0 = as.numeric(as.character(test_prediction2$death)), scores.class0 =

xgb.pr  <- pr.curve(weights.class0 = as.numeric(as.character(test_prediction2$death)), scores.class0 = t

xgb.realiability <- reliability_diagramm(as.numeric(as.character(test_prediction2$death)),test_predicti

plot(xgb.roc)
```



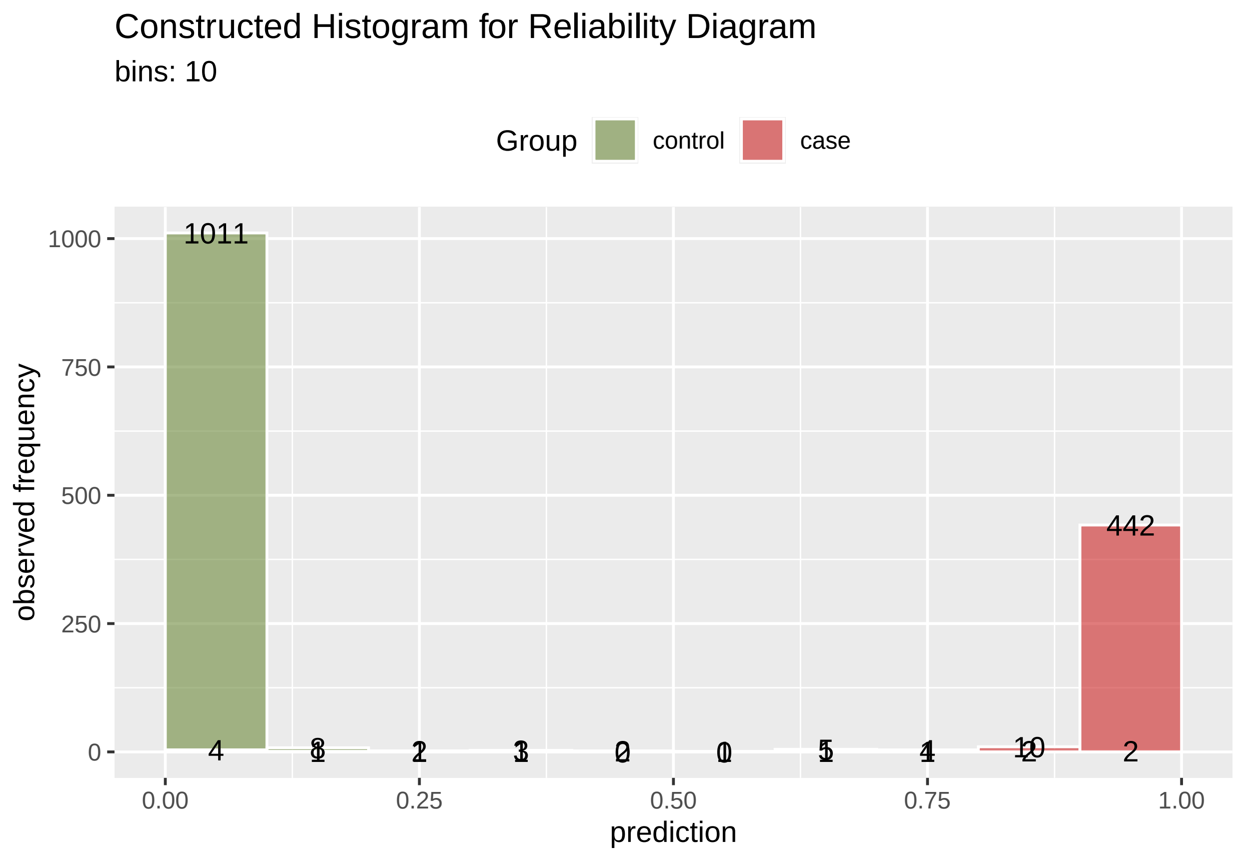
```r
plot(xgb.pr)
```



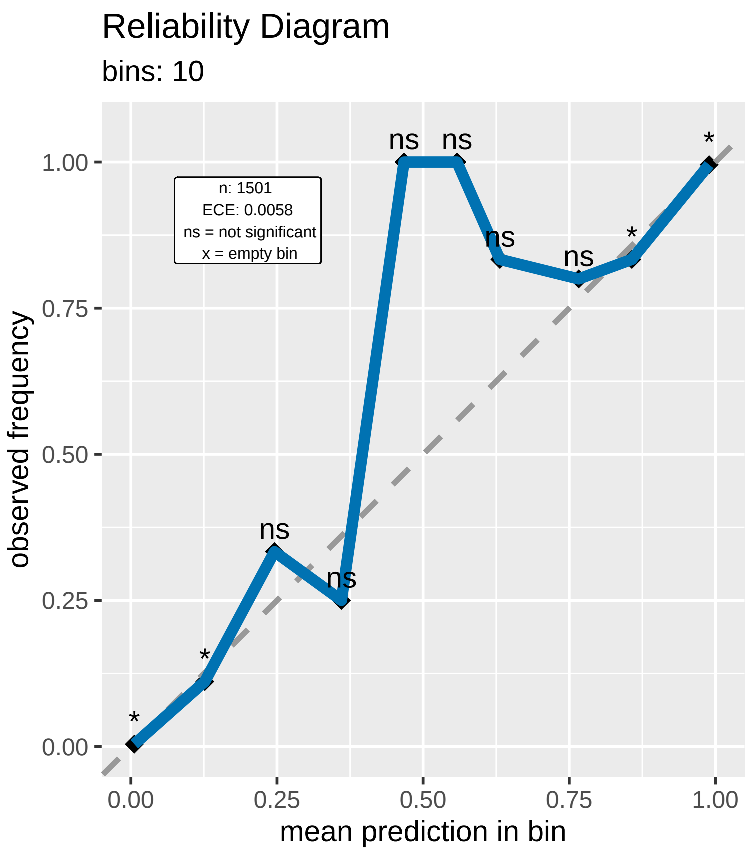
```
xgb.realiability
```

```
## $calibration_error
## $calibration_error$ECE_equal_width
## [1] 0.0058
##
## $calibration_error$MCE_equal_width
## [1] 0.00181
##
## $calibration_error$ECE_equal_freq
## [1] 0.00549
##
## $calibration_error$MCE_equal_freq
## [1] 0.00149
##
## $calibration_error$RMSE
## [1] 0.08892
##
## $calibration_error$CLE_class_1
## [1] 0.03422
##
## $calibration_error$CLE_class_0
## [1] 0.01272
##
## $calibration_error$brier
## [1] 0.00791
```

```
##
## $calibration_error$brier_class_1
## [1] 0.00474
##
## $calibration_error$brier_class_0
## [1] 0.00317
##
##
## $discrimination_error
## $discrimination_error$sens
## [1] 0.989
##
## $discrimination_error$spec
## [1] 0.991
##
## $discrimination_error$acc
## [1] 0.991
##
## $discrimination_error$ppv
## [1] 0.981
##
## $discrimination_error$npv
## [1] 0.995
##
## $discrimination_error$cutoff
## [1] 0.26
##
## $discrimination_error$auc
## [1] 1
##
##
## $rd_breaks
## [1] 10
##
## $histogram_plot
```



```
##
## $diagram_plot
```



```
##
## $mean_pred_per_bin
##  [1] 0.005886626 0.126504620 0.245710452 0.360273674 0.467259884 0.557971984
##  [7] 0.631369288 0.766228503 0.857167960 0.989391492
##
## $accuracy_per_bin
##  [1] 0.003940887 0.111111111 0.333333333 0.250000000 1.000000000 1.000000000
##  [7] 0.833333333 0.800000000 0.833333333 0.995495495
##
## $freq_per_bin
##  [1] 0.6762158561 0.0059960027 0.0019986676 0.0026648901 0.0013324450
##  [6] 0.0006662225 0.0039973351 0.0033311126 0.0079946702 0.2958027981
##
## $sign
##  [1] "*"  "*"  "ns" "ns" "ns" "ns" "ns" "ns" "*"  "*"
```

```
caret::confusionMatrix(test_prediction2$death
                       , factor(as.numeric(test_prediction2$pred > 0.5))
                       , positive = "1", mode = "everything")
```

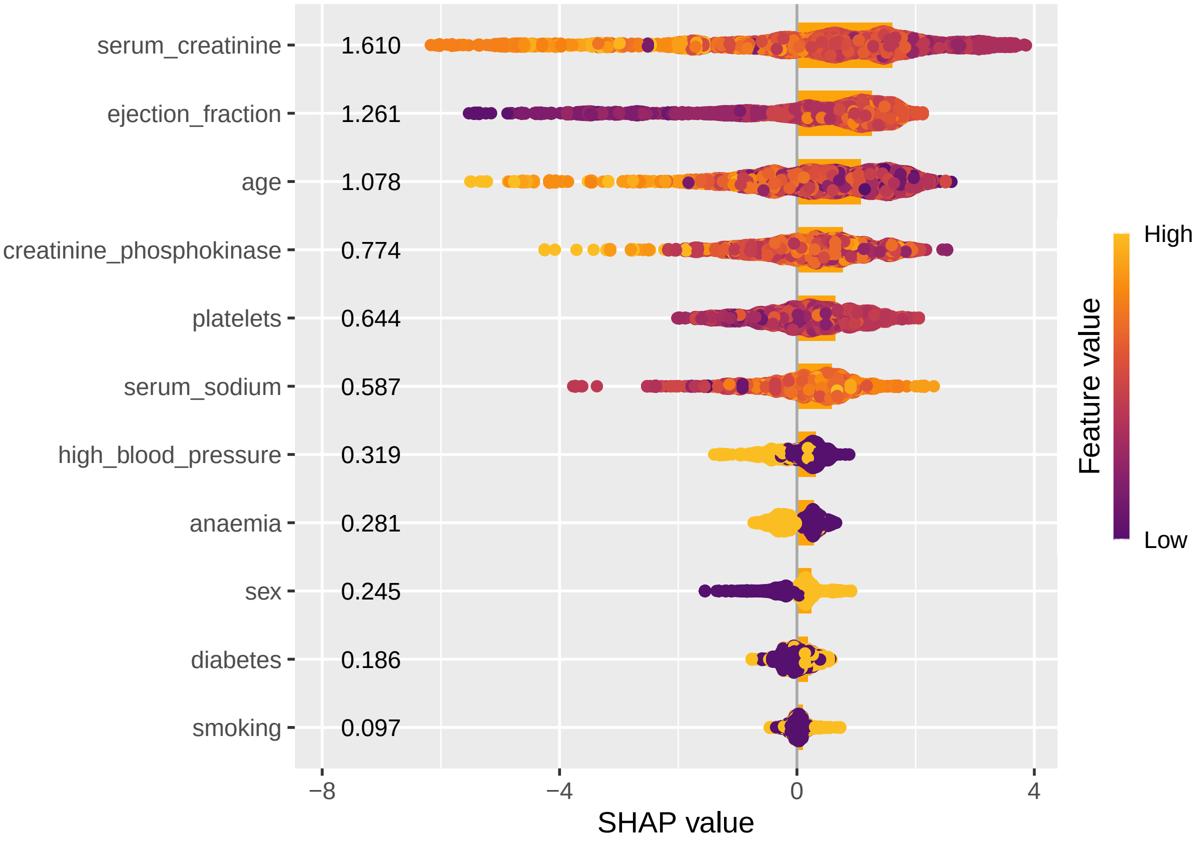
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1024    6
##          1    9  462
##
##                Accuracy : 0.99
##                  95% CI : (0.9836, 0.9944)
##     No Information Rate : 0.6882
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9768
##
##  Mcnemar's Test P-Value : 0.6056
##
##             Sensitivity : 0.9872
##             Specificity : 0.9913
##          Pos Pred Value : 0.9809
##          Neg Pred Value : 0.9942
##               Precision : 0.9809
##                  Recall : 0.9872
##                      F1 : 0.9840
##              Prevalence : 0.3118
##          Detection Rate : 0.3078
##    Detection Prevalence : 0.3138
##       Balanced Accuracy : 0.9892
##
##        'Positive' Class : 1
##
```

We can see that our model extrapolate well to test set and achieve an AUC ROC of 0.999.

Model Explanation

We will use SHAP Values in the XGBoost model without "time" attribute to understand how the model makes predictions. SHAP Values will not model the subyacent reality, only explain a black box model as XGBoost.

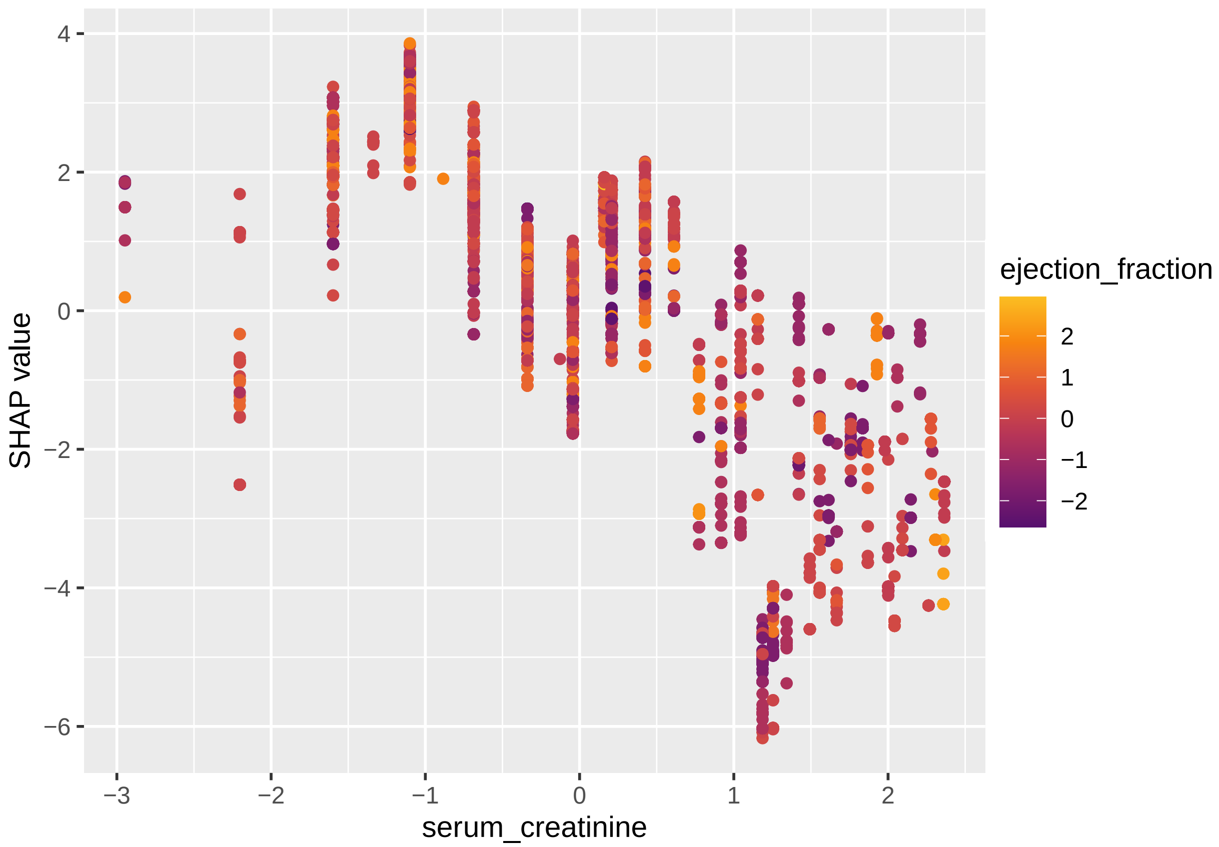
```r
library(shapviz)

df2  <- bake(preprocessing_recipe
                          , new_data = df
                          , has_role("predictor")
                          , composition = "matrix")

shap <- shapviz(extract_fit_engine(xgboost_model_fit)
                , X_pred = df2
                , which_class = '1')

sv_importance(shap, kind = "both", show_numbers = TRUE, bee_width = 0.2)
```



We can see that most important values to make predictions are serum creatinine, ejection fraction and age. This is consistent with "vip" gain importance.

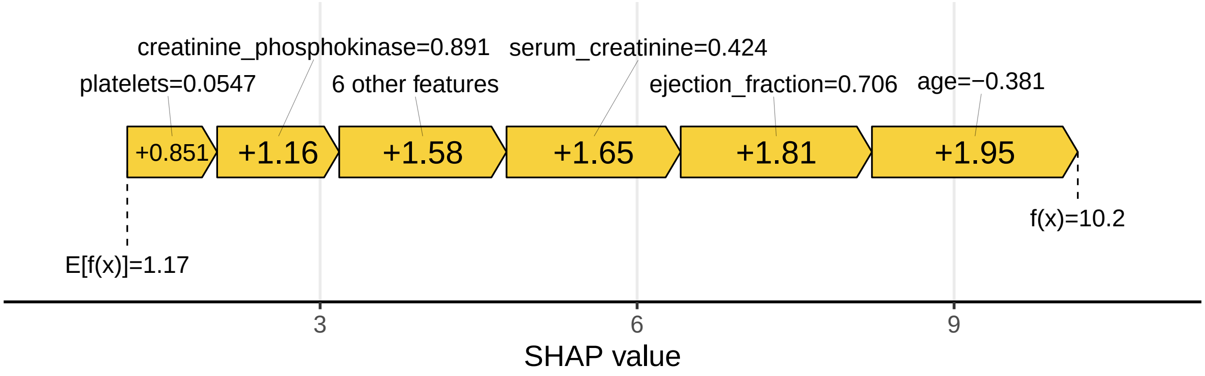
Another advantage of SHAP Values, is that we can see how attributes interact to each other:

```
sv_dependence(shap, "serum_creatinine", color_var = "ejection_fraction")
```

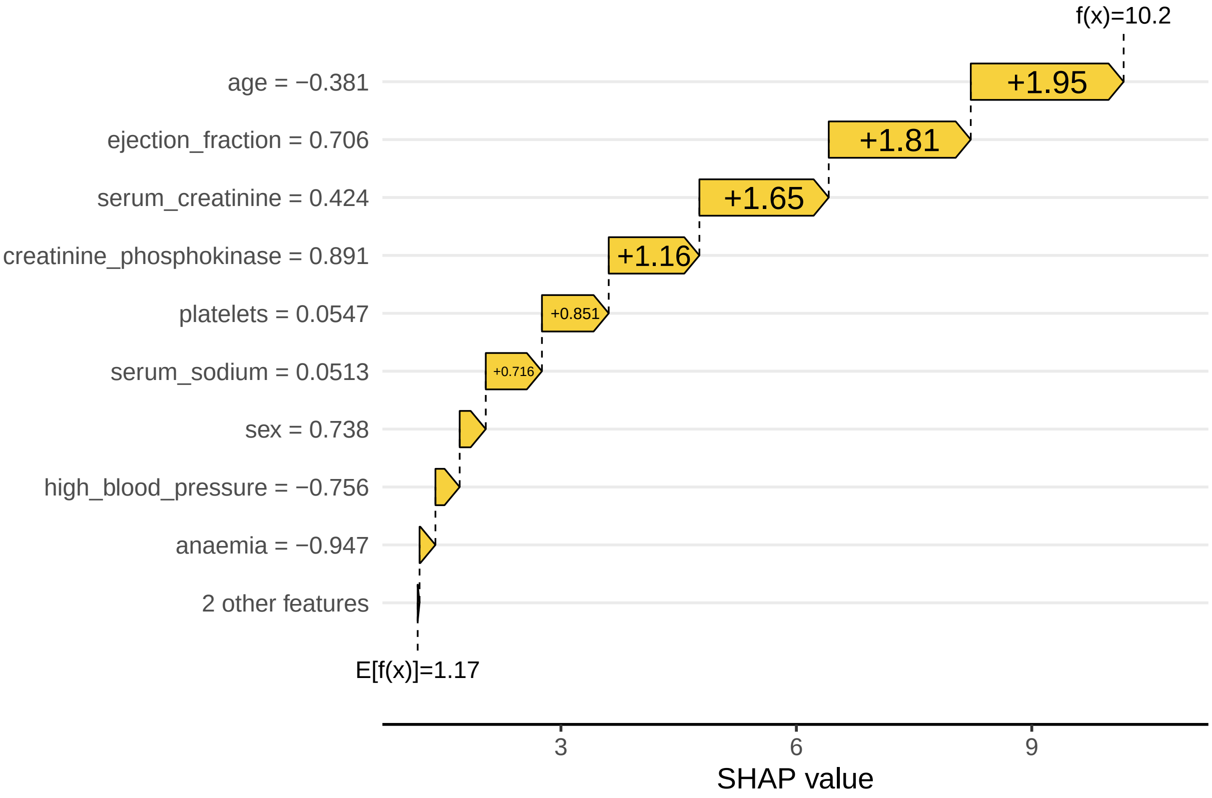


And we can decompose the probability output of our model in the contribution of every single input:

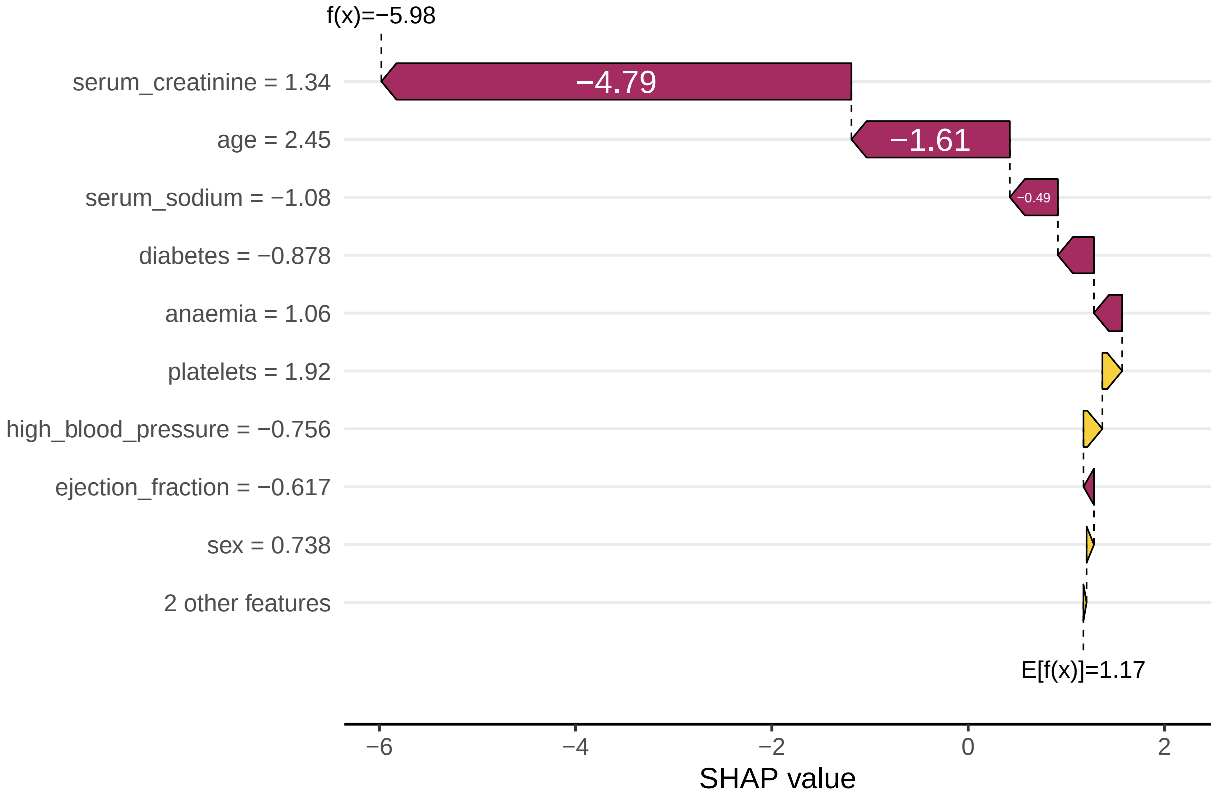
```
sv_force(shap, row_id = 1L)
```



```
sv_waterfall(shap, row_id = 1L)
```



```r
sv_waterfall(shap, row_id = 5L)
```



Two issues with interpreting this are that SHAP Values are not directly probabilities (or sum of probabilities) and that attribute units are transformed by our recipe (see negative values of age).

The first issue can be addressed by applying this formula:

$$Probability = \frac{1}{1 + e^{(-\sum_{k=0}^{l} SHAP_k)}}$$

with "l" the number of attributes, and $SHAP_0 = \mathbb{E}[f(x)]$ the base SHAP value. With this, we can transform SHAP to probabilities.

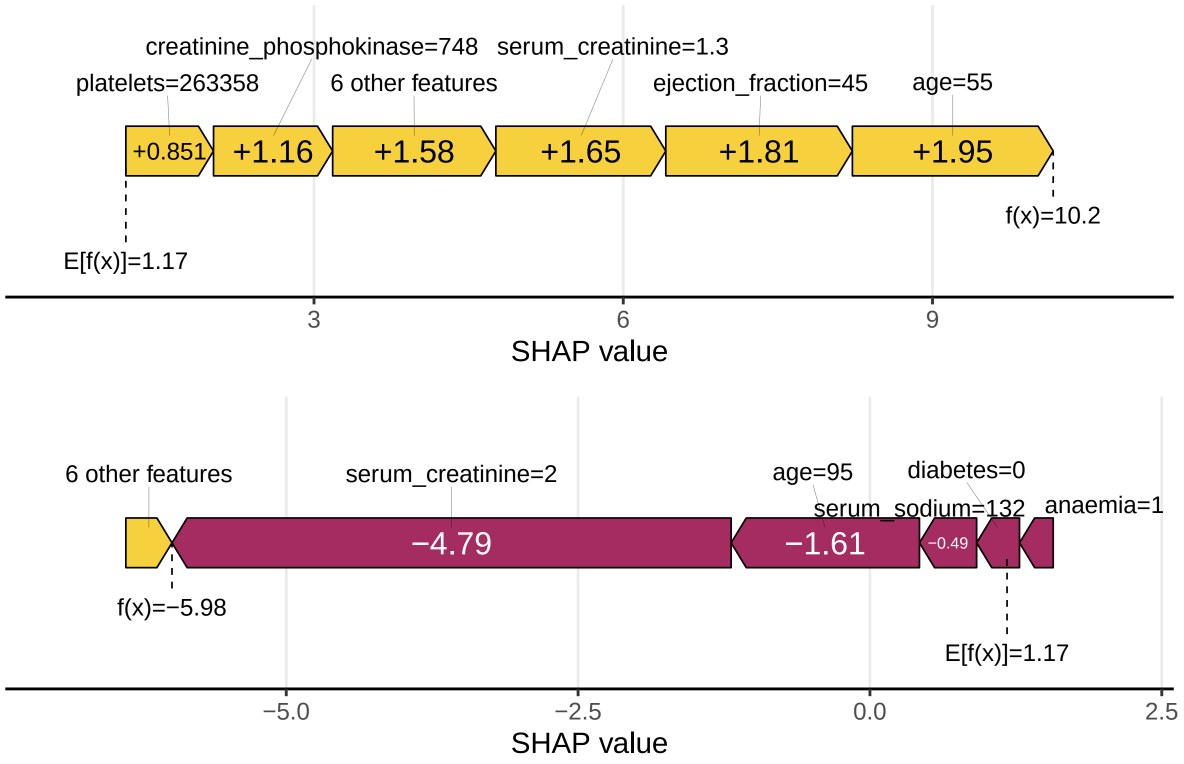
```
shap$S %>%
  data.frame() %>%
  tibble() -> shap.df
shap.df$baseline <- shap$baseline

shap.df$total_shap <- rowSums(shap.df)
shap.df$probability <- 1/(1+exp(-shap.df$total_shap))

shap.df %>%
  head(10) %>%
  select(total_shap, probability)
```

```
## # A tibble: 10 x 2
##    total_shap probability
##         <dbl>       <dbl>
##  1       10.2      1.00
##  2        6.19     0.998
##  3        5.91     0.997
##  4        2.62     0.932
##  5       -5.98     0.00253
##  6        5.34     0.995
##  7        4.44     0.988
##  8        5.23     0.995
##  9        7.59     0.999
## 10        5.86     0.997
```

We can see that a total SHAP of 10.17 is related to a probability of 99.9%; and a total SHAP of -5.97 is related to a probability of 0.2%. This helps to interpret SHAP values in terms of probability.

We should still interpret attribute SHAP values in terms of the original attributes values (and not the ones that we input in XGBoost model as they are transformed by Yeo Johnson and later normalized).

```r
shap2 <- shap

df %>%
  select(- death_event) -> shap2$X

sv_force(shap2, row_id = 1L) / sv_force(shap2, row_id = 5L)
```



Results

We found a model that reach an out of sample ROC AUC of 0.999, an XGBoost tuned, and we use SHAP Values to open break the opacity of the model.

We can interpret individual predictions of the model based in the original attributes. We can gain insights about what the model does with every attribute. For example, it seems that hearth failure at the age of 55 is related (accordingly to the model) to lower survival probability than if the same condition appears at the age of 95. This might seems counter-intuitive at first (we could normally expect that older people might have less probability to survive), but the fact that a younger person has a heart failure might mean more, related to the seriousness of the underlying condition.

It would be good to check with doctors if some of this insights makes them sense. For example, serum creatinine.

"*Creatinine is a waste product in your blood that comes from your muscles. Healthy kidneys filter creatinine out of your blood through your urine.*

Your serum creatinine level is based on a blood test that measures the amount of creatinine in your blood. It tells how well your kidneys are working. When your kidneys are not working well, your serum creatinine level goes up." (see References 7)

Considering this, it doesn't make sense that higher levels of creatinine serum in the second case are related with negative SHAP value (decreases probability of death) if compared with the first case in the example. If we see that a level of 2 for serum creatinine is out of the normal range for males and women, but 1.3 might be a normal level, it makes even less sense.

In the case of ejection fraction, "*Ejection fraction (EF) is a measurement, expressed as a percentage, of how much blood the left ventricle pumps out with each contraction. An ejection fraction of 60 percent means that 60 percent of the total amount of blood in the left ventricle is pushed out with each heartbeat. A normal heart's ejection fraction is between 55 and 70 percent.*" (see References 8). In this case, it does make sense that the first example has an increase of death probability because he has an ejection fraction of 45, which is lower than the normal level.

Conclusions

We could find an almost perfect model (AUC ROC 0.999 in a test set) with an XGBoost. It is possible to use this model probability to assess differently patients accordingly to survival probability. This is true even if we remove "time" attribute for being suspicious of data leakage.

Simpler models, such as Logit, didn't achieve similar levels of performance. The big gap between XGBoost and Logit leads to deal with the more complex model and make an extra effort to make it interpretable. We didn't cover other kinds of models (neural network, SVM, etc.) because the results where promising enough with models checked so far (that were also inspired in previous research).

Using SHAP Values we can open individual probabilities in the relevance of every single input. Some of this inputs might be controlled by doctors, some others not. The most promising attributes found by the model, serum creatinine and ejection fraction are the ones that previous research find also relevants. Other factors, as "age" are also relevant (even if they are not directly mentioned in other research).

To correctly interpret this results, it would be good to check with experts in this area.

It would be advisable too to check data collection, in case this good results are consecuence of data leakage of some kind. We have no information of the moment doctors collect samples (just after hearth failure? during follow up period? if the second, same concerns about "time" attribute might arise).

References

1. Heart Failure Clinical Records. (2020). UCI Machine Learning Repository. https://doi.org/10.24432/C5Z89R.

2. Kaggle dataset: Heart Failure Prediction - Clinical Records. Aadarsh Velu. Kaggle Dataset

3. Kaggle notebook 1: Hearth Failure Predict. Yilmaz Akgul. Kaggle notebook 1

4. Kaggle notebook 2: Heart Failure Prediction - Voting Classifier .99%. Aadarsh Velu. Kaggle notebook 2

5. Table 4 Survival prediction results on all clinical features – mean of 100 executions. Table 1

6. Table 9 Survival prediction results on serum creatinine and ejection fraction – mean of 100 executions. Table 2

7. Serum Creatinine Test. Serum Creatinine

8. Ejection Fraction. Ejection Fraction