AWS Test - Vehicle Failure Prediction

Code

Felipe Gerard

October 2017

Summary

The objective of the current document is to device a fleet preventive maintenance strategy. To do this, we need to predict vehicle failures given a set of telemetries.

We will use an internal company database of previous failures and Machine Learning model to do this. A high-level overview of the process is as follows:

1. Download and prepare the dataset
2. Explore the dataset to identify data quality issues and initial insights
3. Perform Machine Learning on the dataset to predict failure
    a. Decide which type of learner to use
    b. Tune the model's hyperparameters using a subset of the data called training set (months 1-7)
    c. Estimate the performance of the model by training a model on the training set and predicting on the test set (months 8-11)
    d. Train the final, production-ready model using the full dataset

Setup and data preparation

Libraries

Load required libraries. I am a big advocate for Hadely Wickham's team `tidyverse`. I also like `mlr` for Machine Learning. I find it is clearer than `caret`.

Hide

```
library(tidyverse)
library(ggplot2)
library(lubridate)
library(mlr)
library(viridis)
```

Download and save / read data

Download data from the Internet and save it locally. If it already exists, then just read it from local disk.

Hide

```
data_url <- 'http://aws-proserve-data-science.s3.amazonaws.com/device_failure.csv'
file_name <- 'device_failure.csv'
if (!file.exists(file_name)){
  df <- read_csv(data_url, col_types = cols(failure = col_factor(levels = 0:1)))
  write_csv(df, file_name)
} else {
  df <- read_csv(file_name, col_types = cols(failure = col_factor(levels = 0:1)))
}
head(df, 20)
```

date <date>	device <chr>	failure <fctr>	attribute1 <int>	attribute2 <int>	attribute3 <int>	attribute4 <int>	attribute5 <int>	a
2015-01-01	S1F01085	0	215630672	56	0	52	6	
2015-01-01	S1F0166B	0	61370680	0	3	0	6	
2015-01-01	S1F01E6Y	0	173295968	0	0	0	12	
2015-01-01	S1F01JE0	0	79694024	0	0	0	6	
2015-01-01	S1F01R2B	0	135970480	0	0	0	15	
2015-01-01	S1F01TD5	0	68837488	0	0	41	6	
2015-01-01	S1F01XDJ	0	227721632	0	0	0	8	
2015-01-01	S1F023H2	0	141503600	0	0	1	19	
2015-01-01	S1F02A0J	0	8217840	0	1	0	14	
2015-01-01	S1F02DZ2	0	116440096	0	323	9	9	

1-10 of 20 rows | 1-9 of 12 columns Previous **1** 2 Next

Exploratory analysis

Before doing any modelling, we need to ascertain the dataset's quality. To achieve this we will make a few simple plots and tests. In a real environment we would need to put the data through *ad hoc* quality tests, but for the purpose of this exercise the following will suffice.

Missing values

There are no missing values, which is great.

Hide

```
sum(is.na(df))
```

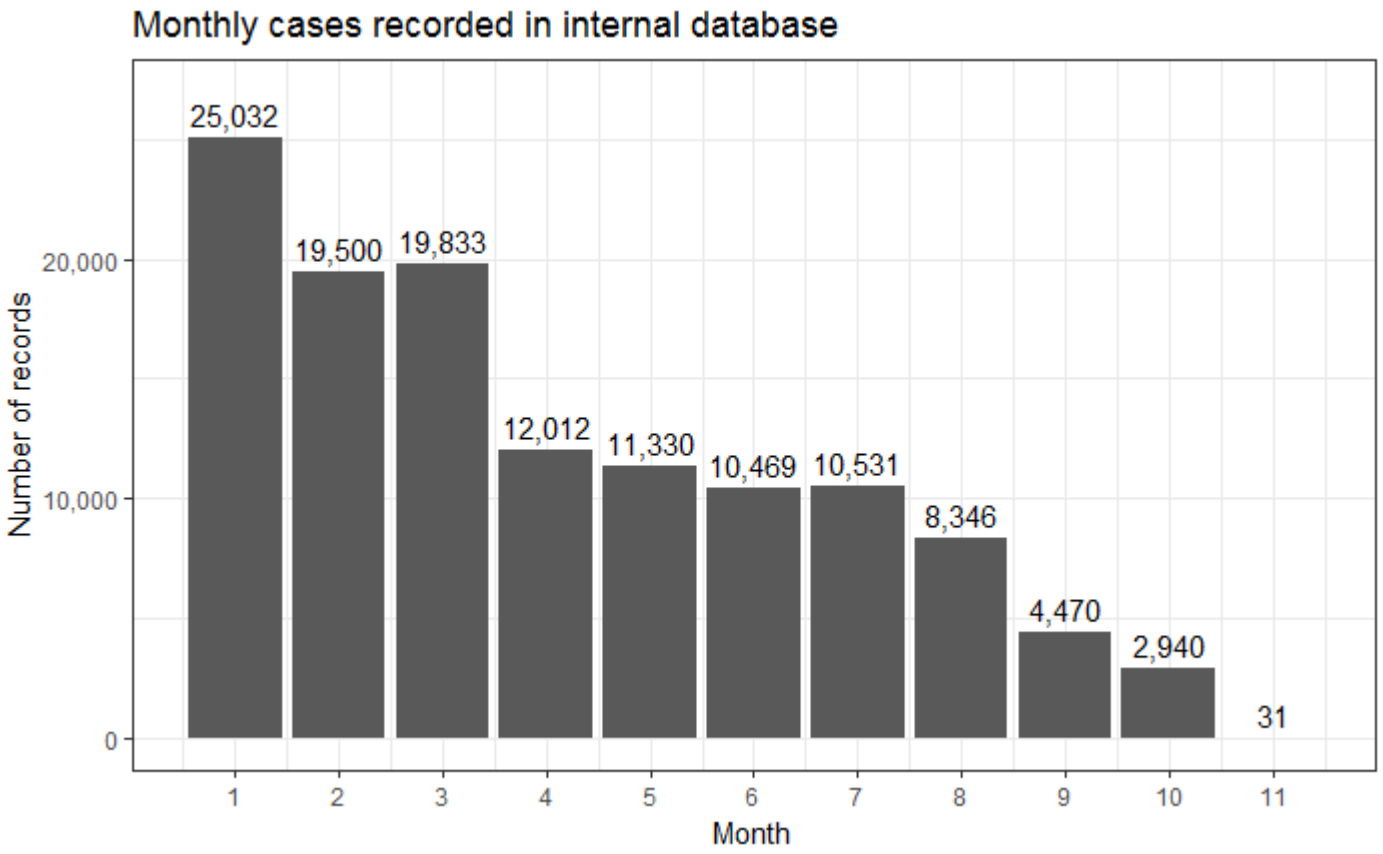
```
[1] 0
```

ID variables and response

For some reason there seem to be many more cases in the first months than in the last ones. We will take this into account when we split the data into train / validation / test.

Hide

```
## Plot cases per month
df %>%
  ggplot(aes(month(date))) +
  geom_bar() +
  geom_text(stat = 'count', aes(label = scales::comma(..count..)), vjust = -.5) +
  scale_x_continuous(breaks = 1:12) +
  scale_y_continuous(labels = scales::comma) +
  coord_cartesian(ylim = c(0, 27000)) +
  theme_bw() +
  labs(
    title = 'Monthly cases recorded in internal database',
    x = 'Month',
    y = 'Number of records'
  )
```



Hide

```
## Monthly statistics table
df %>%
  mutate(month = month(date)) %>%
  group_by(month) %>%
  summarise(
    count = n(),
    n_unique_devices = length(unique(device)),
    n_failures = sum(failure == 1)
  ) %>%
  mutate(
    failure_rate = n_failures / count,
    p_unique_devices = n_unique_devices / count
  ) %>%
  arrange(month)
```

month <dbl>	count <int>	n_unique_devices <int>	n_failures <int>	failure_rate <dbl>	p_unique_devices <dbl>
1	25032	1164	24	0.0009587728	0.04650048
2	19500	726	14	0.0007179487	0.03723077
3	19833	685	9	0.0004537891	0.03453840
4	12012	491	9	0.0007492507	0.04087579
5	11330	424	21	0.0018534863	0.03742277

month <dbl>	count <int>	n_unique_devices <int>	n_failures <int>	failure_rate <dbl>	p_unique_devices <dbl>
6	10469	352	6	0.0005731206	0.03362308
7	10531	346	16	0.0015193239	0.03285538
8	8346	334	4	0.0004792715	0.04001917
9	4470	184	0	0.0000000000	0.04116331
10	2940	146	3	0.0010204082	0.04965986

1-10 of 11 rows Previous **1** 2 Next

Attributes

We will next explore attribute statistics and distributions. It seems that some variables are very concentrated around zero but have a few very large values. We will leave them in the dataset because Random Forest does not have too many problems handling this and also because outliers may contain positive instances.

Hide

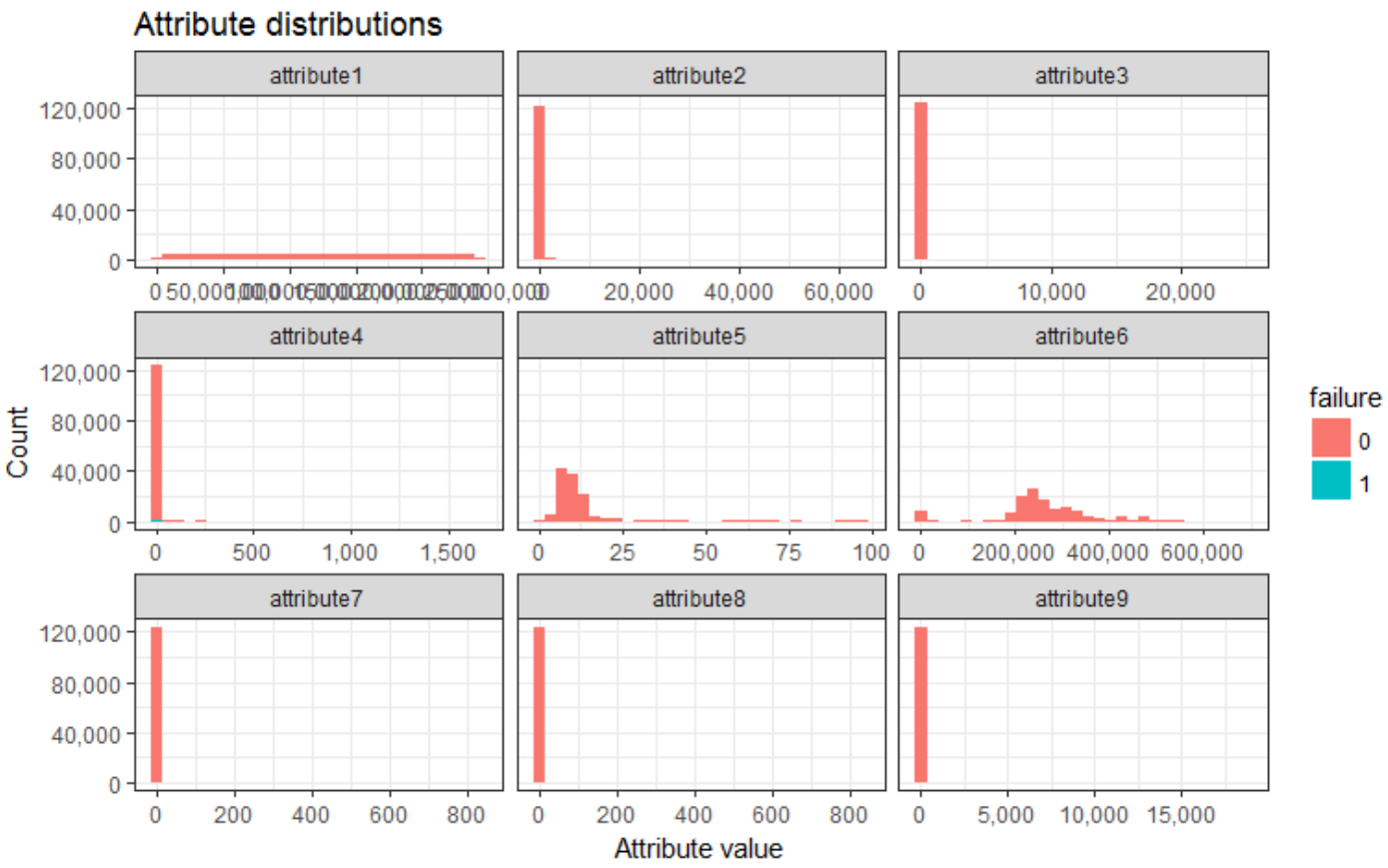
```
## Basic statistics
df %>%
  select(starts_with('attribute')) %>%
  map2_df(., names(.), function(x, name){
    tibble(
      variable = name,
      min = min(x),
      median = median(x),
      mean = mean(x),
      max = max(x)
    )
  })
```

variable <chr>	min <int>	median <dbl>	mean <dbl>	max <int>
attribute1	0	122795744.0	1.223868e+08	244140480
attribute2	0	0.0	1.594848e+02	64968
attribute3	0	0.0	9.940455e+00	24929
attribute4	0	0.0	1.741120e+00	1666
attribute5	1	10.0	1.422269e+01	98
attribute6	8	249799.5	2.601729e+05	689161
attribute7	0	0.0	2.925282e-01	832
attribute8	0	0.0	2.925282e-01	832
attribute9	0	0.0	1.245152e+01	18701

9 rows

Hide

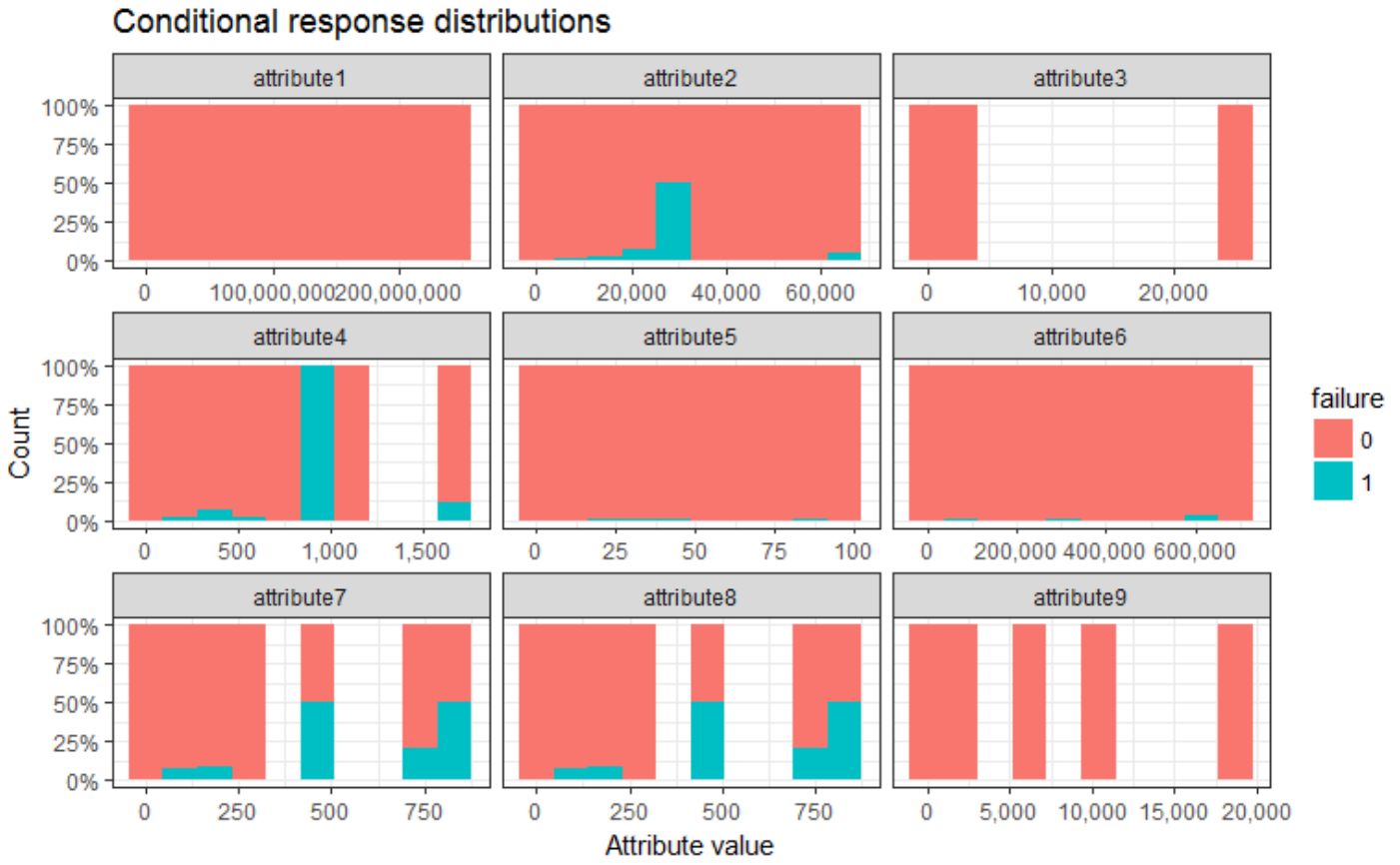
```
## Tables (omitted in final report)
tabs <- df %>%
  select(starts_with('attribute')) %>%
  map(function(x){
    tab <- table(x)
    as.data.frame(tab[1:min(length(tab), 100)])
  })
## Variable distributions
df %>%
  gather(attribute, value, starts_with('attribute')) %>%
  ggplot(aes(value, fill = failure)) +
  geom_histogram() +
  facet_wrap(~ attribute, scales = 'free_x') +
  scale_x_continuous(labels = scales::comma) +
  scale_y_continuous(labels = scales::comma) +
  theme_bw() +
  labs(
    title = 'Attribute distributions',
    x = 'Attribute value',
    y = 'Count'
  )
```



It can be seen that attributes 1, 3, 5, 6 and 9 do not say anything clear about the response. We will leave them in the mix because Random Forest will mostly ignore them and will choose the remaining attributes more often.

Hide

```
## Variable response distributions
df %>%
  gather(attribute, value, starts_with('attribute')) %>%
  ggplot(aes(value, fill = failure)) +
  geom_histogram(bins = 10, position = 'fill') +
  facet_wrap(~ attribute, scales = 'free_x') +
  scale_x_continuous(labels = scales::comma) +
  scale_y_continuous(labels = scales::percent) +
  theme_bw() +
  labs(
    title = 'Conditional response distributions',
    x = 'Attribute value',
    y = 'Count'
  )
```



Machine Learning

Variable transformations

There is no need to transform the variables since we will use Random Forest, which is unaffected by any monotonic transformations.

Hide

```
#eps <- 0.1
ml <- df %>%
  mutate(
    month = month(date)
  )
  # select(failure, starts_with('attribute'))
  # mutate_at(vars(attribute2, attribute3, attribute4, attribute5, attribute7, attribute8, attri
bute9),
  #           funs(log10(ifelse(. <= 0, eps, .))))
```

Train / test split

Tuning and error estimation should take the time factor into account. We will tune the hyperparameters on months 1-7, using 1-5 for training and 6-7 for validation. Then we will train on 1-7 and estimate the final testing error using months 8-11. We will then train the final model on all 11 months.

Hide

```
training_months <- 7
validation_months <- 2
testing_months <- max(ml$month) - training_months
tr <- ml %>%
  filter(month <= training_months)
te <- ml %>%
  filter(month > training_months)
## Months in training set
sort(unique(tr$month))
```

```
[1] 1 2 3 4 5 6 7
```

Hide

```
## Months in testing set
sort(unique(te$month))
```

```
[1]  8  9 10 11
```

Modelling

Setup

We will use the R package `mlr`, which offers a consistent interface for multiple ML R packages. Here we setup the task, learner and resampling strategy (train on 1-5, validate on 6-7).

Hide

```
## Generate training task and final task (i.e. train + test for final model)
gen_task <- function(data) {
  data %>%
    select(failure, starts_with('attribute')) %>%
    as.data.frame() %>%
    makeClassifTask(
      data = .,
      target = 'failure',
      positive = '1'
    )
}
tsk <- gen_task(tr)
final_tsk <- gen_task(ml)
## Define learner
lrn <- makeLearner(
  cl = 'classif.randomForest',
  predict.type = 'prob',
  fix.factors.prediction = TRUE
)
## Resampling strategy
tr_idx <- which(tr$month <= training_months - validation_months)
val_idx <- setdiff(1:nrow(tr), tr_idx)
rs <- makeFixedHoldoutInstance(
  train.inds = tr_idx,
  test.inds = val_idx,
  size = nrow(tr)
)
rs$desc$predict <- 'both'
```

Hyperparameter tuning / grid search

The next step is to optimize the hyperparameters to be used. We do this with a simple grid search.

Hide

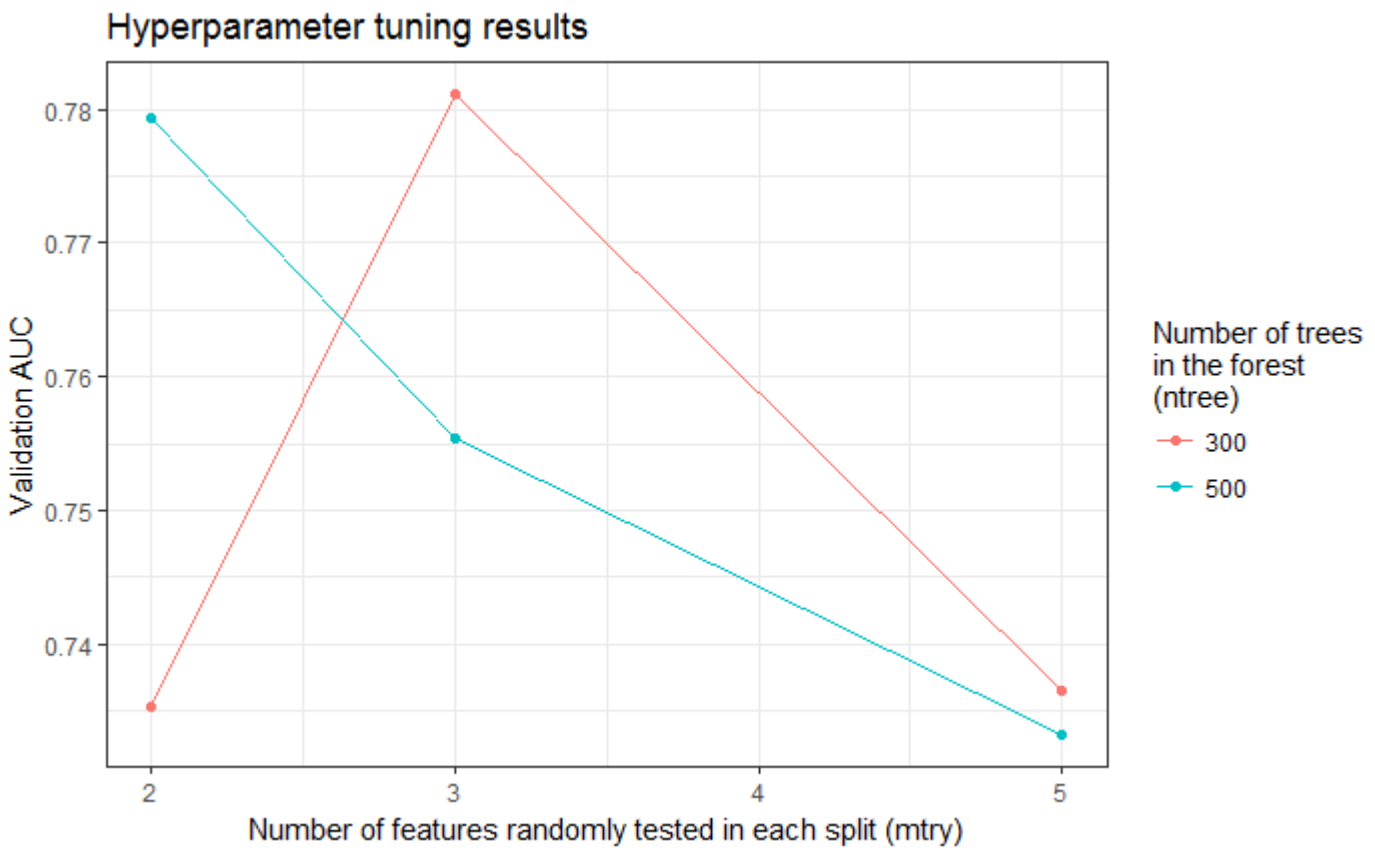
```
pars <- makeParamSet(
  makeDiscreteParam('mtry', values = c(2,3,5)),
  makeDiscreteParam('ntree', values = c(300, 500))
)
ctrl <- makeTuneControlGrid()
set.seed(1234)
tune <- tuneParams(
  learner = lrn,
  task = tsk,
  resampling = rs,
  measures = list(auc, mmce, tp, fn, tn, fp, featperc, timetrain, timepredict, timeboth),
  par.set = pars,
  control = ctrl,
  show.info = TRUE
)
```

```
[Tune] Started tuning learner classif.randomForest for parameter set:
          Type len Def  Constr Req Tunable Trafo
mtry  discrete   -   -   2,3,5   -    TRUE     -
ntree discrete   -   - 300,500   -    TRUE     -
With control class: TuneControlGrid
Imputation value: -0Imputation value: 1Imputation value: -0Imputation value: InfImputation valu
e: -0Imputation value: InfImputation value: 1Imputation value: InfImputation value: InfImputatio
n value: Inf
[Tune-x] 1: mtry=2; ntree=300
[Tune-y] 1: auc.test.mean=0.735,mmce.test.mean=0.00105,tp.test.mean=   0,fn.test.mean=  22,tn.te
st.mean=2.1e+04,fp.test.mean=   0,featperc.test.mean=   1,timetrain.test.mean=36.1,timepredict.t
est.mean=0.58,timeboth.test.mean=36.7; time: 0.6 min
[Tune-x] 2: mtry=3; ntree=300
[Tune-y] 2: auc.test.mean=0.781,mmce.test.mean=0.00105,tp.test.mean=   0,fn.test.mean=  22,tn.te
st.mean=2.1e+04,fp.test.mean=   0,featperc.test.mean=   1,timetrain.test.mean=36.2,timepredict.t
est.mean=0.38,timeboth.test.mean=36.5; time: 0.6 min
[Tune-x] 3: mtry=5; ntree=300
[Tune-y] 3: auc.test.mean=0.736,mmce.test.mean=0.00105,tp.test.mean=   0,fn.test.mean=  22,tn.te
st.mean=2.1e+04,fp.test.mean=   0,featperc.test.mean=   1,timetrain.test.mean=44.3,timepredict.t
est.mean= 0.5,timeboth.test.mean=44.8; time: 0.8 min
[Tune-x] 4: mtry=2; ntree=500
[Tune-y] 4: auc.test.mean=0.779,mmce.test.mean=0.00105,tp.test.mean=   0,fn.test.mean=  22,tn.te
st.mean=2.1e+04,fp.test.mean=   0,featperc.test.mean=   1,timetrain.test.mean=  57,timepredict.t
est.mean=0.75,timeboth.test.mean=57.8; time: 1.0 min
[Tune-x] 5: mtry=3; ntree=500
[Tune-y] 5: auc.test.mean=0.755,mmce.test.mean=0.00105,tp.test.mean=   0,fn.test.mean=  22,tn.te
st.mean=2.1e+04,fp.test.mean=   0,featperc.test.mean=   1,timetrain.test.mean=  64,timepredict.t
est.mean=0.65,timeboth.test.mean=64.6; time: 1.1 min
[Tune-x] 6: mtry=5; ntree=500
[Tune-y] 6: auc.test.mean=0.733,mmce.test.mean=0.00105,tp.test.mean=   0,fn.test.mean=  22,tn.te
st.mean=2.1e+04,fp.test.mean=   0,featperc.test.mean=   1,timetrain.test.mean=74.2,timepredict.t
est.mean=0.58,timeboth.test.mean=74.8; time: 1.3 min
[Tune] Result: mtry=3; ntree=300 : auc.test.mean=0.781,mmce.test.mean=0.00105,tp.test.mean=   0,
fn.test.mean=  22,tn.test.mean=2.1e+04,fp.test.mean=   0,featperc.test.mean=   1,timetrain.test.
mean=36.2,timepredict.test.mean=0.38,timeboth.test.mean=36.5
```

We can see the effect of multiple hyperparameters in this plot and table.

Hide

```
tune_df <- generateHyperParsEffectData(tune)
ggplot(tune_df$data, aes(mtry, auc.test.mean, col = factor(ntree))) +
  geom_line() +
  geom_point() +
  scale_color_discrete('Number of trees\nin the forest\n(ntree)') +
  theme_bw() +
  labs(
    title = 'Hyperparameter tuning results',
    x = 'Number of features randomly tested in each split (mtry)',
    y = 'Validation AUC'
  )
```



Hide

```
tune_df$data %>% arrange(desc(auc.test.mean))
```

… <int>	ntree <int>	auc.test.mean <dbl>	mmce.test.mean <dbl>	tp.test.mean <dbl>	fn.test.mean <dbl>	tn.test.mean <dbl>	fp.te
3	300	0.7812134	0.001047619	0	22	20978	
2	500	0.7793598	0.001047619	0	22	20978	
3	500	0.7554256	0.001047619	0	22	20978	
5	300	0.7364707	0.001047619	0	22	20978	
2	300	0.7353743	0.001047619	0	22	20978	
5	500	0.7331783	0.001047619	0	22	20978	

6 rows | 1-8 of 14 columns

Once we have chosen the best hyperparameters, we construct the optimal learner.

Hide

```
lrn_opt <- lrn %>%
  setHyperPars(par.vals = tune$x)
cat('Optimal hyperparameters:\n')
```

```
Optimal hyperparameters:
```

Hide

```
tune$x %>%
  walk2(., names(.), function(val, name){
    cat(sprintf('%s: %s\n', name, as.character(val)))
  })
```

```
mtry: 3
ntree: 300
```

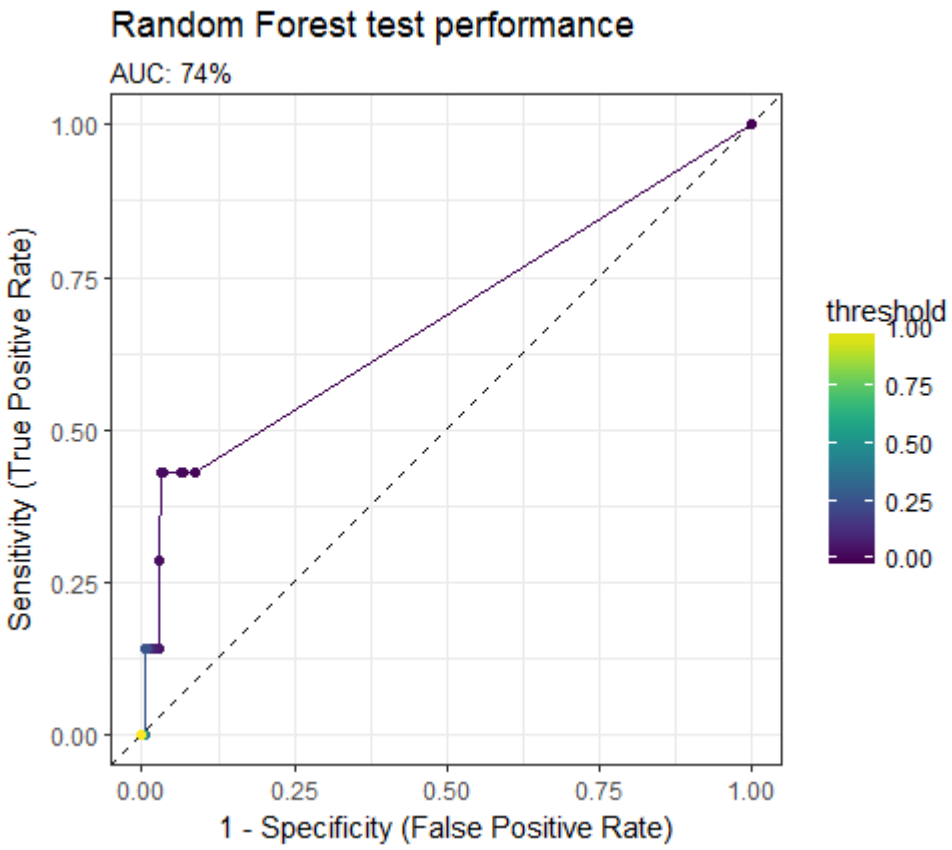
Test performance

Before training the final model, we need to have a reliable estimation of the testing error of the model. To this end, we train on the full training set (months 1-7) and test on the test set (months 8-11).

The results show that a sensitivity of about 40% can be achieved with a tradeoff of 5-10% false positives. Given the fact that failures are scarce (albeit expensive), it is important to have a very low false positive rate. Otherwise a large amount of trips will be flagged with no reason. In this case ~40% of failures can be successfully prevented (sensitivity), while flagging only 1 out of every 20 trips (5%).

Hide

```
set.seed(1234)
mod <- train(
  learner = lrn_opt,
  task = tsk
)
pred <- predict(
  object = mod,
  newdata = te %>% as.data.frame()
)
perf <- performance(pred, list(auc, mmce))
roc_df <- generateThreshVsPerfData(
  obj = pred,
  measures = list(tpr, fpr),
  gridsize = 200
)
ggplot(roc_df$data, aes(fpr, tpr, col = threshold)) +
  geom_abline(slope = 1, intercept = 0, linetype = 'dashed') +
  geom_line() +
  geom_point() +
  scale_color_viridis() +
  coord_equal() +
  theme_bw() +
  labs(
    title = 'Random Forest test performance',
    subtitle = sprintf('AUC: %.0f%%', 100*perf[['auc']]),
    x = '1 - Specificity (False Positive Rate)',
    y = 'Sensitivity (True Positive Rate)'
  )
```



Bonus: Given that we are using a Random Forest, we can also take a look at the variable importance.

Hide

```
mod$learner.model$importance %>%
  as.data.frame() %>%
  rownames_to_column('attribute') %>%
  arrange(desc(MeanDecreaseGini))
```

attribute <chr>	**MeanDecreaseGini** <dbl>
attribute1	49.728130
attribute6	44.158811
attribute2	17.010976
attribute4	15.090509
attribute5	12.708736
attribute8	8.150864
attribute7	7.495739
attribute9	5.971406
attribute3	2.415575

9 rows

Train final model

We now train a final model using all the data. This is the model which we would actually put into the production environment.

Hide

```
set.seed(1234)
mod_final <- train(
  learner = lrn_opt,
  task = final_tsk
)
mod_final
```

```
Model for learner.id=classif.randomForest; learner.class=classif.randomForest
Trained on: task.id = .; obs = 124494; features = 9
Hyperparameters: mtry=3,ntree=300
```