Case Study Technical Report

Alessio Andronico

Last edited on June 10 2021

Contents

1 Introduction

1.1 Motivation

Train delays are a commonly encountered problem for rails operators. The causes of such delays are multiple, like, for example, increasing number of passengers, limited rail network capacity, infrastructure problems, administrative issues, or weather conditions.

Given the complexity of some rail networks, predicting train delays - or simply determine which set of factors influence how delays are accrued - is extremely difficult. At the same time, since delays result in changes to schedule, missed connections, and, more generally, inconvenience and irritation for travelers, developing models to effectively tackle these challenges is quite important since such models could drive the implementation of measures aimed at preventing - or at least reducing - some delays.

In this report, I build a data-driven predictive model to forecast if a train will accrue a delay of more than one minute when stopping at a station. An interesting aspect of the work is that, by analyzing the output of the model, one can also gain some insight into the factors that are linked to the accrued delay.

1.2 Report Summary

The results presented here concern the "Belgium Train Times" dataset, which contains arrival and departure information about trains circulating in Belgium on February 22nd 2021.

The aim of the analysis is the development of a machine learning model able to accurately predict if a train will accrue delay when stopping at a station on the route to its final destination. In order to address such a problem I proceed in four steps:

- Cleaning up the data: a necessary first step for all data analyses. During this step I also add some features that are then used for visualization and/or model calibration.
- Setting up the machine learning problem: the dataset contains train delays (in seconds), but I recast the analysis as a classification problem by choosing an arbitrary - although constrained by the available data - threshold of 60 seconds to determine if the train has accrued delay or not.
- Training: I train four different machine learning models and choose the best performing one. Since the classification problem is unbalanced, I use the ROC AUC as the metric of interest, rather than pure accuracy.
- Testing: I show that the selected model has good performance on the test set, indicating that overfitting did not occur during the training phase.

In the Conclusion section, I then discuss possible ways to improve model performance.

2 The data

2.1 Getting the data

For this project I use five R libraries: 1) **tidyverse** for general data manipulation and wrangling; 2) **lubridate** for handling dates; 3) **cowplot** for arranging multiple figures together; 4) **tidymodels** for machine learning; and 5) **vip** to extract feature importance from the final model.

```r
# Load libraries used for the analyses
library(tidyverse)
library(lubridate)
library(cowplot)
library(tidymodels)
library(vip)

# Set the ggplot theme
```

```
theme_set(theme_bw())

# Set center alignment for all figures
knitr::opts_chunk$set(fig.align = 'center', echo = TRUE)
```

The dataset is a csv file hosted on GitHub, which I download using **readr**'s *read_delim* function. When using the default options there are 8 parsing failures (the offending column being "Planned departure time"), so I download a second copy with all entries as strings to use for subsequent cleaning.

```
data_url <- "https://raw.githubusercontent.com/datacamp/careerhub-data/master/"
dataset <- "Belgium%20Train%20Times/belgium-train-times.csv"

# Download with default options (8 parsing failures)
df <- read_delim(paste0(data_url, dataset), delim = ';')

# Download again but keeping all entries as strings
df_char <- read_delim(paste0(data_url, dataset), delim = ';',
                      col_types = cols(.default = col_character()))
```

2.2 Data cleaning

The dataset contains 72609 rows and 18 columns. As a first step to clean it, I look at the parsing failures. These are the NA entries in the column "Planned departure time" that are not found in the dataframe I downloaded parsing all entries as strings:

```
# Indices of rows to fix
na_inds_df <- is.na(df$`Planned departure time`)
na_inds_df_char <- is.na(df_char$`Planned departure time`)
to_fix <- which(na_inds_df & (!na_inds_df_char))
```

When I look at these 8 entries, it becomes clear that they are due to some data entry issue (the year is 1899!):

```
# Planned departure time is in 1899
df_char[to_fix, "Planned departure time"]
```

```
## # A tibble: 8 x 1
##   `Planned departure time`
##   <chr>
## 1 1899-12-30 00:00:00
## 2 1899-12-30 00:00:00
## 3 1899-12-30 00:00:00
## 4 1899-12-30 00:00:00
## 5 1899-12-30 00:00:00
## 6 1899-12-30 00:00:00
## 7 1899-12-30 00:00:00
## 8 1899-12-30 00:00:00
```

Since we have the actual departure date and time and the delay at departure, it is easy to reconstruct the missing information:

```
# Create a datetime column with departure date and time
clean_df <- df %>%
  mutate(planned_dep_datetime = ymd_hms(paste(`Planned departure date`,
                                               `Planned departure time`)))

# Planned departure = actual departure - delay
clean_df$planned_dep_datetime[to_fix] <-
```

```
  clean_df$`Actual departure date`[to_fix] +
  hms(clean_df$`Actual departure time`[to_fix]) -
  seconds(clean_df$`Delay at departure`[to_fix])

# Check that the problem is indeed fixed
clean_df[to_fix, c("Planned departure time", "planned_dep_datetime")]
```

```
## # A tibble: 8 x 2
##   `Planned departure time` planned_dep_datetime
##   <time>                   <dttm>
## 1    NA                    2021-02-23 00:00:00
## 2    NA                    2021-02-23 00:00:00
## 3    NA                    2021-02-23 00:00:00
## 4    NA                    2021-02-23 00:00:00
## 5    NA                    2021-02-23 00:00:00
## 6    NA                    2021-02-23 00:00:00
## 7    NA                    2021-02-23 00:00:00
## 8    NA                    2021-02-23 00:00:00
```

All the parsing failures were therefore due to departures scheduled on February 23rd at 00:00.

The second thing that I notice in the dataset is that there are many stations where trains do not actually stop (the planned departure time is the same as the planned departure time). To avoid biasing the analyses, I therefore remove these entries:

```
# Remove stops where trains do not stop
clean_df_stops <- clean_df %>%
  filter(is.na(`Planned departure time`) |
           is.na(`Planned arrival time`) |
           (`Planned departure time` != `Planned arrival time`))
```

After this step the dataset has 30780 rows, less than half of the original dataset.

2.3 Feature Engineering

Before moving on to visually explore the data, I add a last data preprocessing step to further tidy the data and create new features. The steps in the code fragment below do the following:

1. Rename some of the columns having long names.
2. Determine the delay the train had when arriving at a station (*delay_in*), the delay it had when leaving the stating (*delay_out*), and the accrued delay (*accrued_delay* = *delay_out* - *delay_in*).
3. Create a new feature describing the type of train according to the information found at this link.
4. Create two new variables for the planned departure time of the train: *planned_dep_hour* for the hour of departure, and *planned_dep*, a categorical variable grouping together different hours of departure ("Morning", "Mid-day", "Afternoon", and "Night") to use for visualization.

Note that for the type of train I am grouping together "Peak" and "Extra" trains because of their small number and also because, according to the link above, they both correspond to trains added during particularly busy periods during the day.

```
# The type of Belgian trains is available at:
# https://www.belgiantrain.be/en/support/faq/faq-routes-schedules/faq-train-types
final_df <- clean_df_stops %>%
  # When 'Planned arrival time' is NA we are at the departure station
  mutate(delay_in = case_when(is.na(`Planned arrival time`) ~ 0,
                              TRUE ~ `Delay at arrival`)) %>%
  rename(delay_out = `Delay at departure`) %>%
```

```r
  # When 'delay_out' is NA we are at the final station
  filter(!is.na(delay_out)) %>%
  # Group trains by their type
  mutate(type = case_when(
    `Relation` %in% c("EURST", "ICE", "TGV", "THAL") ~ "International",
    `Relation` %in% c("P", "EXTRA") ~ "Peak/Extra",
    str_detect(`Relation`, "IC ") ~ "InterCity",
    TRUE ~ "Local")) %>%
  # Switch to friendlier column names
  rename(id = `Train number`,
         operator = `Railway operator`,
         station = `Stopping place`) %>%
  # Compute accrued delay and create categorical variable for departure time
  mutate(accrued_delay = delay_out - delay_in,
         planned_dep_hour = hour(planned_dep_datetime),
         planned_dep = case_when(planned_dep_hour %in% 4:10 ~ "Morning",
                                 planned_dep_hour %in% 11:15 ~ "Mid-day",
                                 planned_dep_hour %in% 16:20 ~ "Afternoon",
                                 TRUE ~ "Night")) %>%
  # Subset of columns used for the analyses
  select(station, id, operator, type, delay_in, delay_out,
         planned_dep_hour, planned_dep, accrued_delay)
```

2.4 Exploratory Data Analysis

Now that the data are in a more suitable form, I can start to explore them in depth.

The vast majority of the entries in the dataset correspond to local and Intercity trains:

```r
table(final_df$type)
```

```
##
##     InterCity International         Local     Peak/Extra
##         12896            40         12385          1937
```

Moreover, as expected, most of the entries are stops recorded during the day:

```r
table(final_df$planned_dep)
```

```
##
## Afternoon   Mid-day   Morning     Night
##      8075      7529      9106      2548
```

If we look at the variable of interest (*accrued_delay*) it is clear that, by far, the vast majority of trains does not accrue delay when stopping: the distribution of accrued delays has a sharp peak at 0, but it does have long tails, especially to the right (Figure 1A below).

```r
# Density of accrued delays
p1 <- final_df %>%
  ggplot(aes(x = accrued_delay)) +
  geom_density() +
  labs(x = "Accrued Delay (s)", y = "Density")

# Accrued delay vs delay at arrival
p2 <- final_df %>%
  ggplot(aes(x = delay_in, y = accrued_delay)) +
  geom_point(alpha = 0.3) +
  geom_hline(yintercept = 1, color = "red") +
  labs(x = "Delay at Arrival (s)", y = "Accrued Delay (s)")

plot_grid(p1, p2, labels = "AUTO")
```

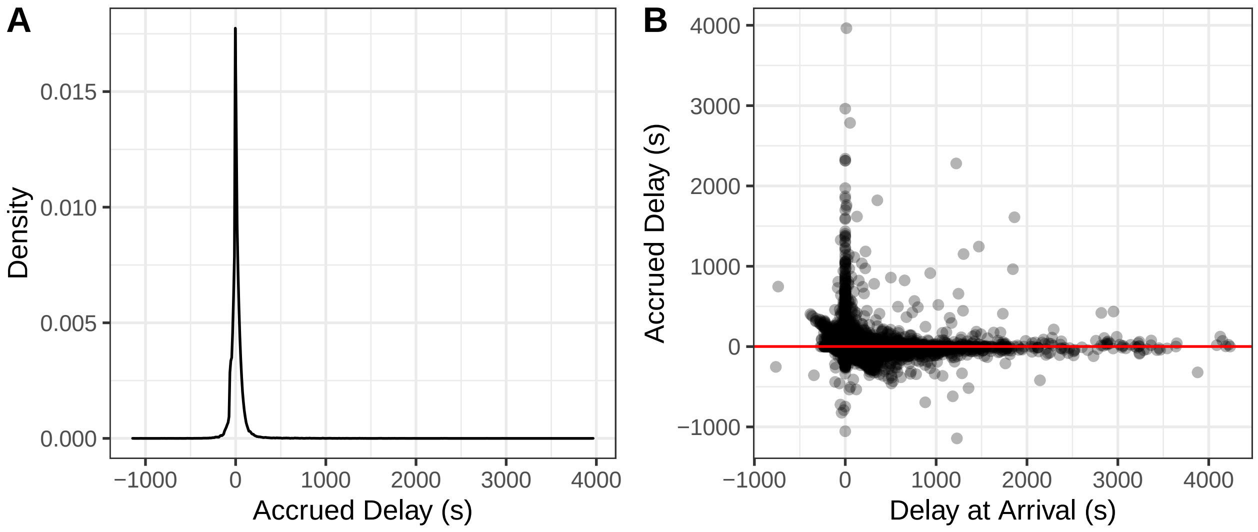

Figure 1: A) Distribution of accrued delays. B) Accrued delay vs delay at arrival.

Another useful way to visualize the data is to plot the accrued delay versus the delay at arrival (Figure 1B above): most points lie near the line $y = 0$ (red line), which correspond to a delay at departure equal to the delay at arrival. However, there are also quite a few trains that arrived on time at the stopping station ($delay\_in \sim 0$) and left the station being considerably late.

It is also interesting to look at the accrued delays versus the departure time or the train type:

```
# Accrued delay vs departure time
p1 <- final_df %>%
  mutate(planned_dep = factor(planned_dep,
                              levels = c("Morning", "Mid-day",
                                         "Afternoon", "Night"))) %>%
  ggplot(aes(x = planned_dep, y = accrued_delay)) +
  geom_boxplot() +
  labs(x = "Planned Departure Time", y = "Accrued Delay (s)")

# Accrued delay vs train type
p2 <- final_df %>%
  ggplot(aes(x = type, y = accrued_delay)) +
  geom_boxplot() +
  labs(x = "Train Type", y = "Accrued Delay (s)")

plot_grid(p1, p2, labels = "AUTO")
```

Once again, the distributions appear to have very long tails, so that it is difficult to see differences in the medians. However, it is quite apparent that morning and afternoon stops exhibit a larger variance in accrued delays (Figure 2A), and that the same holds true for stops of local and Intercity trains (Figure 2B).

Another piece of information that we can extract from the available data is the number of train stops at each station, a proxy for the size/importance of a station. If, for example, big stations were busier than

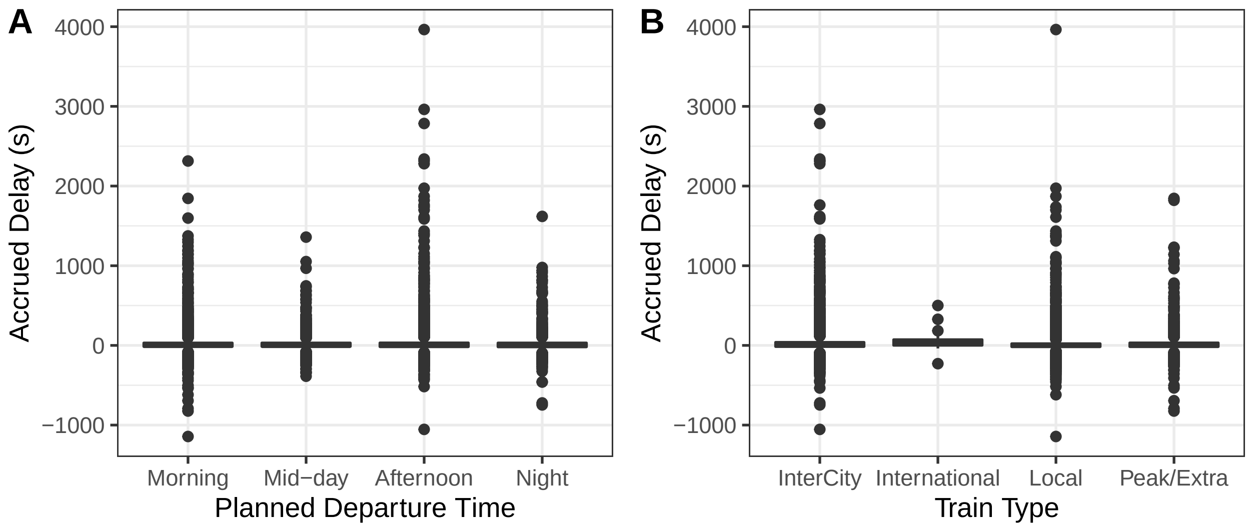

Figure 2: Accrued delay vs planned departure time (A) and train type (B).

smaller ones and therefore more likely to have delays, this could be a useful variable when predicting the delay accrued by a train.

```r
# Compute the number of train stops for each station
ntrains_by_station <- final_df %>%
  group_by(station) %>%
  summarize(n = n()) %>%
  ungroup()

# Filter the 10 largest (by number of train stops) stations
largest_stations <- ntrains_by_station %>%
  arrange(desc(n)) %>%
  dplyr::slice(1:10) %>%
  mutate(station = factor(station, levels = rev(station)))

# Number of train stops for the 10 largest stations
p1 <- largest_stations %>%
  ggplot(aes(x = station, y = n)) +
  geom_col() +
  coord_flip() +
  labs(x = "Station", y = "Number of Train Stops")

# Accrued delay distributions for the 10 largest stations
p2 <- largest_stations %>%
  left_join(final_df, by = "station") %>%
  mutate(station = factor(station, levels = rev(largest_stations$station))) %>%
  ggplot(aes(x = station, y = accrued_delay)) +
  geom_boxplot() +
  coord_flip() +
  labs(x = "Station", y = "Accrued Delay (s)")

plot_grid(p1, p2, labels = "AUTO")
```

Figure 3A shows that there are indeed vast differences among stations: the top three (all within Brussels) recorded more than 900 train stops while the fifth station (Mechelen) recorded about half the stops. However,

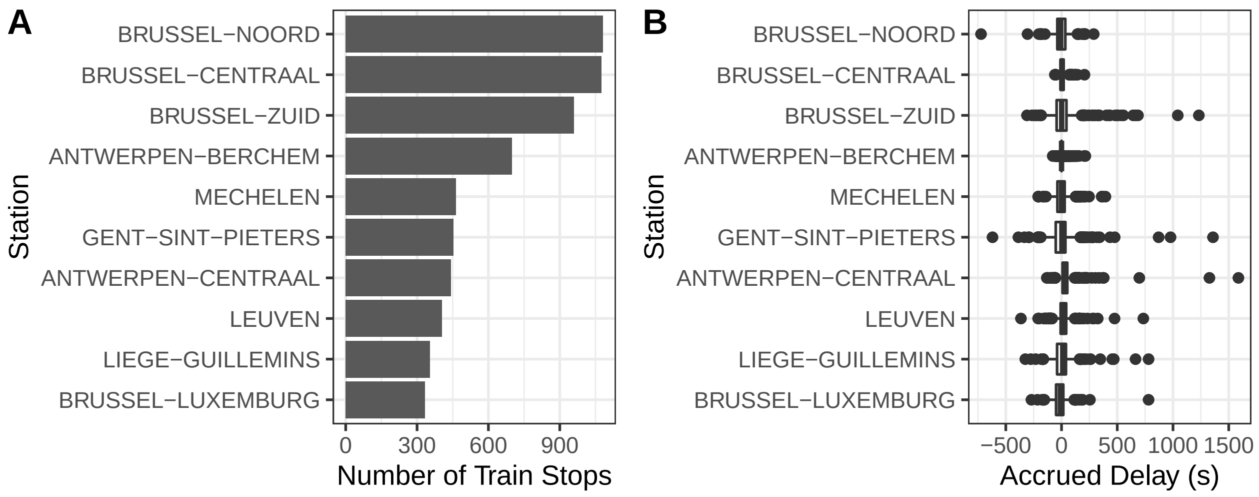

Figure 3: Number of train stops (A) and accrued delay distribution (B) for the 10 largest stations.

as seen in Figure 4B, a larger station does not necessarily imply a longer accrued delay.

2.5 Setting up the Machine Learning Problem

The interquartile range of the distribution of accrued delays is [-16, 31] seconds, so 50% of the available data represent very small delays. Since I do not think that trying to predict such small delays would be very useful, I decide to instead focus on delays longer than one minute and recast the analysis as a classification problem.

The one-minute threshold is chosen here for convenience: using larger thresholds would result in fewer examples of stops where trains accrued delays, therefore making the learning problem harder.

Once the threshold fixed, I can create the target variable (*is_late*), which, in this case, represents weather the train accrued more than 60 seconds of delay while stopping at a train station:

```
# Create the target variable (is_late) and filter out unused columns
ml_df <- final_df %>%
  left_join(ntrains_by_station, by = "station") %>%
  mutate(is_late = factor(case_when(accrued_delay > 60 ~ "Y", TRUE ~ "N"),
                          levels = c("Y", "N"))) %>%
  select(-station, -id, -delay_out, -accrued_delay, -planned_dep, -operator)
```

As the code fragment above shows, for the following analyses, I remove the station name (*station*) and train number (*id*) (both of which are not useful for prediction), the columns *delay_out* and *accrued_delay* (which would allow to perfectly predict the target variable), and the planned departure time (*planned_dep*) (but I keep the departure hour). I also remove the column *operator*, since all but 13 trains belong to SNCB/NMBS:

```
table(final_df$operator)
```

```
##
## EUROSTARFR  SNCB/NMBS   THI-FACT
##          2      27245         11
```

The four features that are left to predict the target variable are therefore:

1. The type of train (*type*);
2. The delay (in seconds) at arrival (*delay_in*);
3. The planned departure hour (*planned_dep_hour*);
4. The number of train stops recorded on February 22nd at the station where the train is stopping (*n*).

The choices above give rise to a classification problem where the negative class (no delay, or “N”) is dominant:

```
table(ml_df$is_late)
```

```
##
##     Y     N
##  3405 23853
```

3 Model Training

Since the classification problem is unbalanced, I decide to use the ROC AUC as the metric to optimize: this provides the required trade-off between sensitivity and specificity. In fact, since about 90% of the labels in the dataset are “N”, a model that always predicts “N” (no 60s delay is ever accrued) would have an accuracy of about 0.9, a perfect specificity = 1.0, but a sensitivity = 0.0, thus being of little practical use.

3.1 Training and Test Sets

I split the dataset into training (75% of the data) and test set (the remaining 25%) and use 10-fold cross-validation to tune the hyperparameters of the models.

```
# Split dataset into training and test set
set.seed(321)
data_split <- ml_df %>%
  # 75% training / 25% test
  initial_split(prop = 0.75, strata = is_late)
data_train <- data_split %>% training()
data_test <- data_split %>% testing()

# Cross-validation folds (10 folds, no repeats) - same folds for all models
set.seed(55)
train_folds <- data_train %>% vfold_cv(v = 10, repeats = 1)

# Metrics monitored during training (only the ROC AUC)
monitored_metrics <- metric_set(roc_auc)

# Get number of cores available
cores <- parallel::detectCores()
```

Given the strongly nonlinear relationship in Figure 1B, I decide to focus on 4 nonlinear Machine Learning methods:

1. Random Forests.
2. Gradient Boosting (in particular XGBoost).
3. K-Nearest Neighbors.
4. Neural Networks.

3.2 Random Forest

For the random forest model, I use the **ranger** engine, no preprocessing of the data, and the *tune_grid* function from the **tune** library to search for the best hyperparameters[1].

```
################################################################################
# Random forest
################################################################################
```

[1]Running the code provided in this report without changing the variable “perform_tuning” will download my own tuning results from a public repository hosted on GitHub instead of training the models (which take several hours).

```r
# Define the model
# Parameters: https://parsnip.tidymodels.org/reference/rand_forest.html
rf_model <- rand_forest(trees = tune(), min_n = tune(), mtry = tune()) %>%
  # Use all but one core
  set_engine("ranger", num.threads = cores - 1, importance = "impurity") %>%
  set_mode("classification")

# Preprocessing: none
rf_pre <- recipe(is_late ~ ., data = data_train)

# Worflow
rf_wkfl <- workflow() %>%
  add_model(rf_model) %>%
  add_recipe(rf_pre)

# Tuning
perform_tuning <- FALSE
if (perform_tuning) { # Tuning took 1h34m
  set.seed(345)
  t0 <- Sys.time()
  # Use default (space filling) grid of tune_grid
  rf_tuning <- rf_wkfl %>%
    tune_grid(train_folds,
              grid = 100, # 100 parameters combinations
              control = control_grid(verbose = TRUE, save_pred = TRUE),
              metrics = monitored_metrics)
  t_rf <- Sys.time() - t0

  # Save tuning results
  best_rf <- rf_tuning %>%
    select_best(metric = "roc_auc")
  roc_rf <- rf_tuning %>%
    collect_predictions(parameters = best_rf) %>%
    roc_curve(is_late, .pred_Y)
  show_best_rf <- rf_tuning %>%
    show_best()
  write_csv(best_rf, "Tuning/best_rf.csv")
  write_csv(roc_rf, "Tuning/roc_rf.csv")
  write_csv(show_best_rf, "Tuning/show_best_rf.csv")
} else {
  # Load results from GitHub
  res_url <- "https://raw.githubusercontent.com/aandroni/datacamp-certification/main/tuning/"
  best_rf <- read_csv(paste0(res_url, "best_rf.csv"))
  roc_rf <- read_csv(paste0(res_url, "roc_rf.csv"))
  show_best_rf <- read_csv(paste0(res_url, "show_best_rf.csv"))
}
```

3.3 Gradient Boosting (XGBoost)

The code for tuning the gradient boosting model is quite similar to the previous one, except that I set the engine to **xgboost** and dummy-encode the only nominal variable (the train type) - the last step being required when using XGBoost.

```r
################################################################################
# Gradient boosting
################################################################################
# Define the model
# Parameters: https://parsnip.tidymodels.org/reference/boost_tree.html
xgb_model <- boost_tree(tree_depth = tune(), learn_rate = tune(),
                        loss_reduction = tune(), min_n = tune(),
                        sample_size = tune(), trees = tune()) %>%
  set_engine("xgboost", num.threads = cores - 1) %>%
  set_mode("classification")

# Preprocessing: dummy-encode nominal variables
xgb_pre <- recipe(is_late ~ ., data = data_train) %>%
  step_dummy(all_nominal(), -all_outcomes())

# Worflow
xgb_wkfl <- workflow() %>%
  add_recipe(xgb_pre) %>%
  add_model(xgb_model)

# Tuning
perform_tuning <- FALSE
if (perform_tuning) { # Tuning took 3h36m
  set.seed(345)
  t0 <- Sys.time()
  # Use default (space filling) grid of tune_grid
  xgb_tuning <- xgb_wkfl %>%
    tune_grid(train_folds,
              grid = 100, # 100 parameters combinations
              control = control_grid(verbose = TRUE, save_pred = TRUE),
              metrics = monitored_metrics)
  t_xgb <- Sys.time() - t0

  # Save tuning results
  best_xgb <- xgb_tuning %>%
    select_best(metric = "roc_auc")
  roc_xgb <- xgb_tuning %>%
    collect_predictions(parameters = best_xgb) %>%
    roc_curve(is_late, .pred_Y)
  show_best_xgb <- xgb_tuning %>%
    show_best()
  write_csv(best_xgb, "Tuning/best_xgb.csv")
  write_csv(roc_xgb, "Tuning/roc_xgb.csv")
  write_csv(show_best_xgb, "Tuning/show_best_xgb.csv")
} else {
  # Load results from GitHub
  res_url <- "https://raw.githubusercontent.com/aandroni/datacamp-certification/main/tuning/"
  best_xgb <- read_csv(paste0(res_url, "best_xgb.csv"))
  roc_xgb <- read_csv(paste0(res_url, "roc_xgb.csv"))
  show_best_xgb <- read_csv(paste0(res_url, "show_best_xgb.csv"))
}
```

3.4 K-Nearest Neighbors

For the k-nearest neighbors model tuning, I set the engine to **kknn** and normalize all numerical variables. Nominal variables are automatically taken care of by **kknn**.

```
################################################################################
# KNN
################################################################################
# Define the model
# Parameters: https://parsnip.tidymodels.org/reference/nearest_neighbor.html
knn_model <- nearest_neighbor(neighbors = tune(), dist_power = tune(),
                              weight_func = tune()) %>%
  set_engine("kknn", num.threads = cores - 1) %>%
  set_mode("classification")

# Preprocessing: normalize numeric features
knn_pre <- recipe(is_late ~ ., data = data_train) %>%
  step_normalize(all_numeric())

# Worflow
knn_wkfl <- workflow() %>%
  add_recipe(knn_pre) %>%
  add_model(knn_model)

# Tuning
perform_tuning <- FALSE
if (perform_tuning) { # Tuning took 5h18m
  set.seed(345)
  t0 <- Sys.time()
  # Use default (space filling) grid of tune_grid
  knn_tuning <- knn_wkfl %>%
    tune_grid(train_folds,
              grid = 100, # 100 parameters combinations
              control = control_grid(verbose = TRUE, save_pred = TRUE),
              metrics = monitored_metrics)
  t_knn <- Sys.time() - t0

  # Save tuning results
  best_knn <- knn_tuning %>%
    select_best(metric = "roc_auc")
  roc_knn <- knn_tuning %>%
    collect_predictions(parameters = best_knn) %>%
    roc_curve(is_late, .pred_Y)
  show_best_knn <- knn_tuning %>%
    show_best()
  write_csv(best_knn, "Tuning/best_knn.csv")
  write_csv(roc_knn, "Tuning/roc_knn.csv")
  write_csv(show_best_knn, "Tuning/show_best_knn.csv")
} else {
  # Load results from GitHub
  res_url <- "https://raw.githubusercontent.com/aandroni/datacamp-certification/main/tuning/"
  best_knn <- read_csv(paste0(res_url, "best_knn.csv"))
  roc_knn <- read_csv(paste0(res_url, "roc_knn.csv"))
  show_best_knn <- read_csv(paste0(res_url, "show_best_knn.csv"))
}
```

3.5 Neural Network

Finally, to tune the neural network model I use the **nnet** engine, normalize all numerical variables, and dummy-encode the nominal variable (the train type).

```r
################################################################################
# Neural net
################################################################################
# Define the model
# Parameters: https://parsnip.tidymodels.org/reference/mlp.html
nnet_model <- mlp(hidden_units = tune(), penalty = tune(), epochs = tune()) %>%
  set_engine("nnet", num.threads = cores - 1) %>%
  set_mode("classification")

# Preprocessing: normalize numeric variables and dummy-encode nominal variables
nnet_pre <- recipe(is_late ~ ., data = data_train) %>%
  step_normalize(all_numeric()) %>%
  step_dummy(all_nominal(), -all_outcomes())

# Worflow
nnet_wkfl <- workflow() %>%
  add_recipe(nnet_pre) %>%
  add_model(nnet_model)

# Tuning
perform_tuning <- FALSE
if (perform_tuning) { # Tuning took 42m
  set.seed(345)
  t0 <- Sys.time()
  # Use default (space filling) grid of tune_grid
  nnet_tuning <- nnet_wkfl %>%
    tune_grid(train_folds,
              grid = 100, # 100 parameters combinations
              control = control_grid(verbose = TRUE, save_pred = TRUE),
              metrics = monitored_metrics)
  t_nnet <- Sys.time() - t0

  # Save tuning results
  best_nnet <- nnet_tuning %>%
    select_best(metric = "roc_auc")
  roc_nnet <- nnet_tuning %>%
    collect_predictions(parameters = best_nnet) %>%
    roc_curve(is_late, .pred_Y)
  show_best_nnet <- nnet_tuning %>%
    show_best()
  write_csv(best_nnet, "Tuning/best_nnet.csv")
  write_csv(roc_nnet, "Tuning/roc_nnet.csv")
  write_csv(show_best_nnet, "Tuning/show_best_nnet.csv")
} else {
  # Load results from GitHub
  res_url <- "https://raw.githubusercontent.com/aandroni/datacamp-certification/main/tuning/"
  best_nnet <- read_csv(paste0(res_url, "best_nnet.csv"))
  roc_nnet <- read_csv(paste0(res_url, "roc_nnet.csv"))
  show_best_nnet <- read_csv(paste0(res_url, "show_best_nnet.csv"))
}
```

3.6 Training Results

Once the models have been trained, we can inspect and compare their performance:

```r
roc_rf %>% mutate(Model = "RF") %>%
  bind_rows(roc_xgb %>% mutate(Model = "XGB")) %>%
  bind_rows(roc_knn %>% mutate(Model = "KNN")) %>%
  bind_rows(roc_nnet %>% mutate(Model = "NNet")) %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity, color = Model)) +
  geom_line() +
  geom_abline(linetype = "dashed") +
  coord_equal() +
  labs(x = "1 - Specificity", y = "Sensitivity")
```

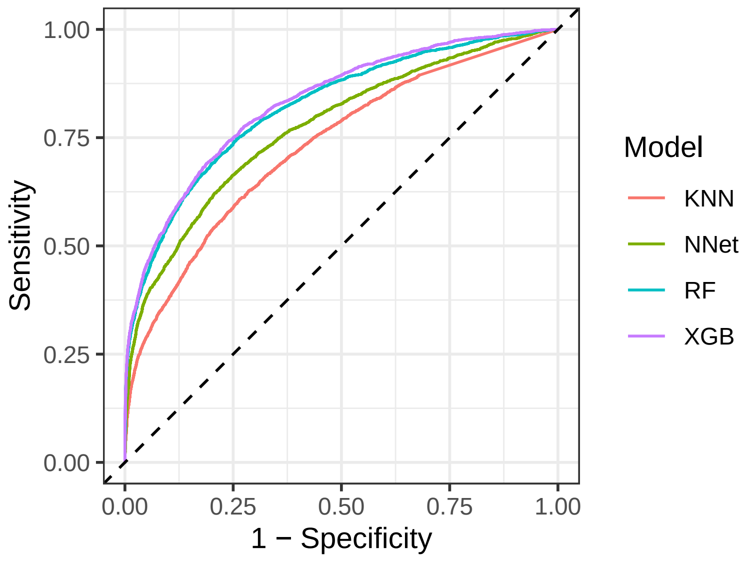

Figure 4: ROC curve of the four models.

Figure 4 shows that XGBoost is the best performing model with respect to the chosen metric, followed quite closely by the random forest model. On the other hand, the KNN and neural network models do not perform as well as the two tree-based models.

The final (cross-validated) ROC AUC of the tuned XGBoost model is 0.83, which represents good, although not astounding, performance.

4 Model Generalization

4.1 Selected Model Test Set Performance

I can now finalize the workflow (in **tidymodels** parlance) and assess if the tuned XGBoost model is able to generalize to unseen data, i.e. the test set.

```r
# Finalize XGBoost workflow
set.seed(321)
final_xgb <- xgb_wkfl %>%
  finalize_workflow(best_xgb) %>%
  last_fit(data_split)
```

The ROC AUC on the test set is 0.83, identical to the estimate obtained via cross-validation, which indicates that the model did not overfit during training.

As expected, the model has very good specificity (0.99), but relatively low sensitivity (0.31).

4.2 Feature Importance

Using the library **vip**, I can now look at how useful the different features were - according to the final XGBoost model - for the classification task:

```
# Pull the fit and get feature importance
final_xgb %>%
  pluck(".workflow", 1) %>%
  pull_workflow_fit() %>%
  vip()
```

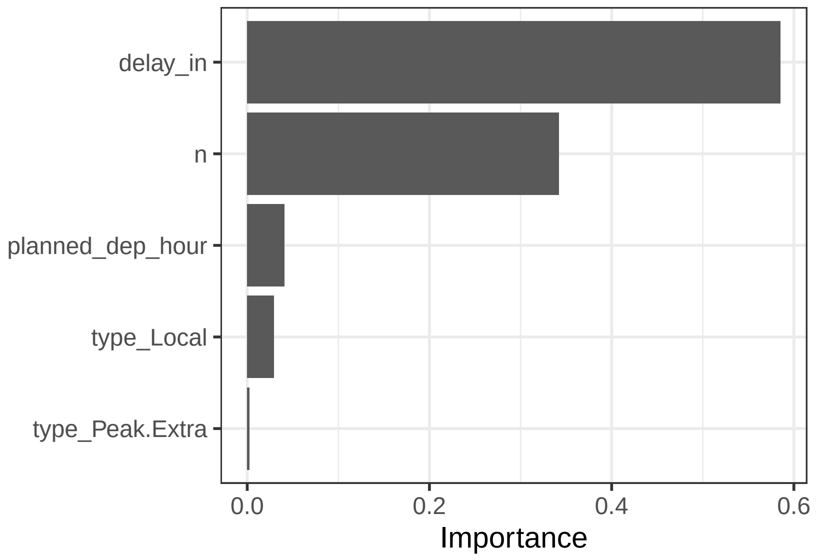

Figure 5: Feature importance for the tuned XGBoost model.

We see (Figure 5) that most of the weight was given to just two features: the delay at arrival (*delay_in*), and the "size" of stopping station (*n*). The other variables' contribution is relatively low. This suggests that perhaps adding other variables may help to improve model performance.

4.3 A Closer Look at the False Negatives

Since the final model weak spot is its sensitivity, it is instructive to look at the distribution of delays at arrival for the false negatives:

```
# Get predictions
xgb_pred <- final_xgb %>% collect_predictions()

# False negatives correspond to prediction = "N" but true class = "Y"
inds_fn <- which(xgb_pred$is_late == "Y" & xgb_pred$`.pred_class` == "N")

# Flag false negatives
to_plot <- data_test %>% mutate(`False Negative` = "N")
to_plot$`False Negative`[inds_fn] <- "Y"
# Order levels: first Y then N
to_plot$`False Negative` <- factor(to_plot$`False Negative`,
                                   levels = c("Y", "N"))
```

```r
# Plot the distributions
ggplot(to_plot, aes(x = delay_in, fill = `False Negative`)) +
  geom_density(alpha = 0.5) +
  scale_x_continuous(limits = c(-100, 60)) +
  labs(x = "Delay at Arrival (s)", y = "Density")
```

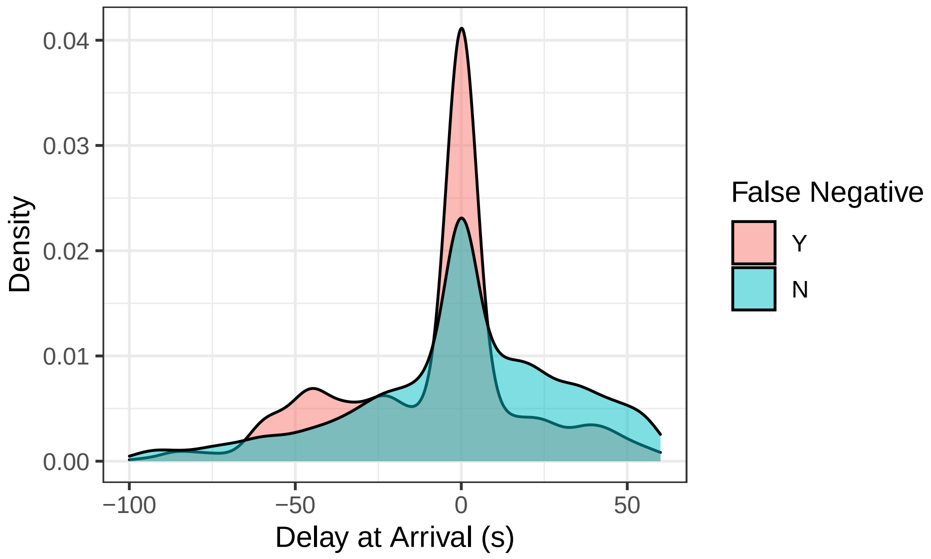

Figure 6: Distribution of delays at arrival.

Figure 6 shows that at least part of the model's problem is that it incorrectly predicted as "not late" many trains that arrived about a minute earlier than scheduled at the stopping station (the red bumpy area on the left of the figure). This could be due for example to technical problems occurred at the station - and for which the model had no information at all.

5 Conclusion and Recommendations

In this report I analyzed the "Belgium Train Times" dataset and trained different machine learning models to predict if a train will accrue more than one minute of delay while stopping at a station.

The main results can be summarized as follows:

1. Among the four tested, the best performing model relied on the XGBoost library and was able to achieve a ROC AUC of 0.83 on both training and test sets.
2. The most important feature for the classification task were the delay at arrival (*delay_in*), and the "size" of the stopping station (n).
3. The final model had very good specificity but relatively low sensitivity. One of the problem of the model was that it incorrectly classified trains that arrived about a minute earlier than scheduled at the stopping station.

While a ROC AUC of 0.83 represents good performance, especially considering the imbalance of the classification task, there is certainly room for improvement. In my opinion, two choices are the most promising for boosting the model sensitivity:

1. **Using More Data**. Although having almost 30000 rows, the dataset that I analyzed here only contains the information about trains circulating on February 22nd 2021, i.e. only one day worth of data. Expanding the time frame of the dataset would provide more example that the machine learning

algorithms could use to learn from: this is especially true for the delays (i.e. the positive class in the analysis), which accounted for only 12% of the available data. Additionally, if more data were available, one could also test more data-hungry models like deep neural networks, although at the cost of longer training times.

2. **Using Additional Data Sources**. Instead of adding more of the same type of data points, one could also use additional datasets, for example meteorological data or information about train accidents. By downloading these two datasets I found out that, on February 22nd, there was no rain, a temperature of about 12 degrees for the entire day, and only one train accident (at Assesse). Therefore, for the analyses presented above, these additional data sources could not be used to improve the performance of the models, but in a context where data for multiple days - or months and even years - were used, they could be interesting to consider. Meteorological data could be used for example to explore and anticipate seasonal patterns in train delays, while the information about train accidents could be used to create additional features flagging stations where accidents occur frequently (and where accrued delays would therefore be more likely).

Since train delays negatively affect travel conditions, the model presented here provides a proof of concept of how models of this type - once refined and properly adjusted for real-time applications - could be advantageously used to inform the implementation of measures that limit the disruption to passenger journeys. Since data-driven, the advantage of this type of approach is that model performance will increase - up to a certain point - over time since more and more data become available for training.