NYC Flights

Sebastian Sauer

2019-08-01

Contents

Fallstudie NYC Flights

Aufgabe: Vorhersage von Verspätungen der Flüge von den Flughäfen von New York City im Jahr 2013.

Vorbereitung

Pakete laden:

```
library(mosaic)
library(tidyverse)
library(lubridate)
library(corrr)
library(caret)
library(doMC)
library(ranger)
library(sjmisc)
```

Daten laden:

```
library(nycflights13)
data(flights)
glimpse(flights)
#> Observations: 336,776
#> Variables: 19
#> $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013,...
#> $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
#> $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
#> $ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 55...
#> $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 60...
#> $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2...
#> $ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 7...
```

```
#> $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 7...
#> $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -...
#> $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV",...
#> $ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79...
#> $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN...
#> $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR"...
#> $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL"...
#> $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138...
#> $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 94...
#> $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5,...
#> $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013...
```

Explorative Datenanalyse

Wie ist die Verspätung verteilt?

Es gibt zwei Variablen, die Verspätung anzeigen: `arr_delay` (Ankunft) und `dep_delay`.

```
favstats(arr_delay ~ 1, data = flights)
#>   1 min  Q1 median Q3  max     mean       sd      n missing
#> 1 1 -86 -17     -5 14 1272 6.895377 44.63329 327346    9430
favstats(dep_delay ~ 1, data = flights)
#>   1 min Q1 median Q3  max     mean       sd      n missing
#> 1 1 -43 -5     -2 11 1301 12.63907 40.21006 328521    8255
```

Nehmen wir `arr_delay`, da die Streuung in dieser Variable höher ist.

VERTIEFUNG

Möchte man einen Befehl auf mehrere Spalten anwenden, so kann man dafür den Befehl `map()` verwendet. `map()` führt ein Befehl auf jede Spalte eines Dataframes aus. Damit man da Ergebnis in Form eines Dataframes (Tabelle) bekommt, fügt man `_df` an `map()` an:

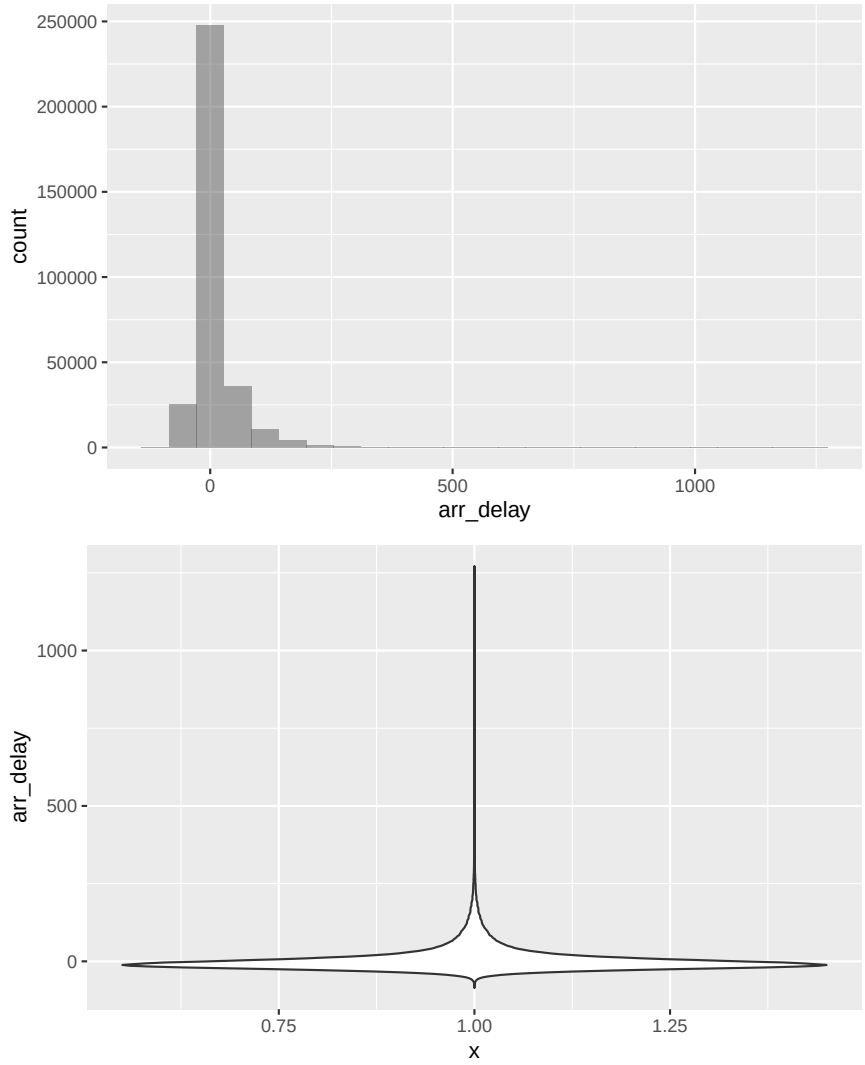
```
flights %>%
  select(arr_delay, dep_delay) %>%
  map_df(favstats)
#>   min  Q1 median Q3  max      mean       sd      n missing
#> 1 -86 -17     -5 14 1272  6.895377 44.63329 327346    9430
#> 2 -43  -5     -2 11 1301 12.639070 40.21006 328521    8255
```

Für den IQR:

```
flights %>%
  select(arr_delay, dep_delay) %>%
  drop_na() %>%
  map_df(iqr)
#> # A tibble: 1 x 2
#>   arr_delay dep_delay
#>       <dbl>     <dbl>
#> 1        31        16
```

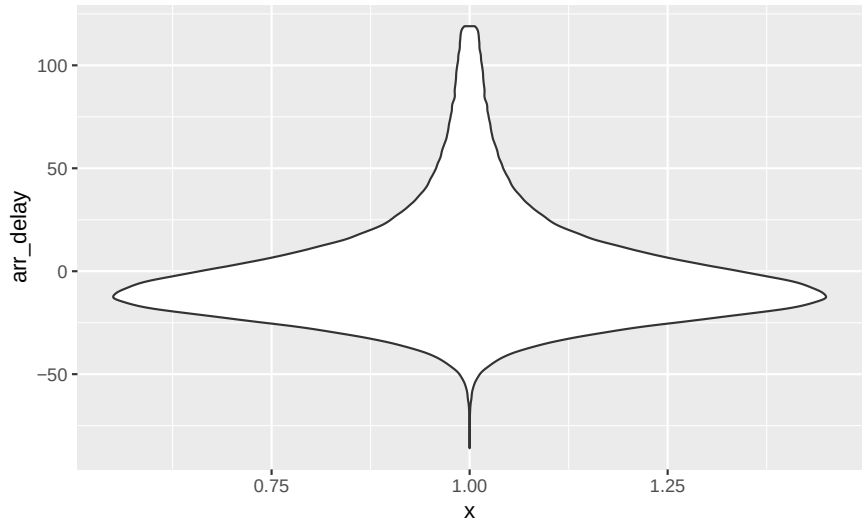
Visualisierung der Verteilung

```
gf_histogram( ~ arr_delay, data = flights)
gf_violin(arr_delay ~ 1, data = flights)
```



Aufgrund des langen rechten Randbereichs (hohe Verspätungswerte) ist das Diagramm nicht sher hilfreich. Begrenzen wir uns besser auf den "inneren" Teil der Flüge (was die Verspätung betrifft).

```
flights %>%
  filter(arr_delay < 120) %>%
  gf_violin(arr_delay ~ 1, data = .)
```



Saisonale Effekte

Es gibt sehr viele potenzielle Ursachen für die Verspätung eines Flugzeugs bzw. eines Flugs. Zu einigen Kandidaten liegen uns Daten vor. Eine naheliegende (obwohl nicht tiefer theoretisch fundierte) Annahme ist, dass es saisonale Einflüsse auf die Verspätung gibt. So könnte Schnee im Winter oder Weihnachtsstress zum Jahreswechsel für Verspätung sorgen. Am Wochenende sind die Menschen entspannter und es wird weniger gereist. Daher könnte es Samstags und Sonntags zu weniger Verspätung kommen.

Nach Jahreszeiten

Berechnen wir die Jahreszeiten:

```
flights2 <- flights %>%
  mutate(season = case_when(
    month %in% c(11, 12, 1, 2, 3) ~ "winter",
    month %in% c(6,7,9) ~ "summer",
    month %in% c(4, 5) ~ "spring",
    TRUE ~ "autumn"
  ))
```

Verspätungen nach Jahreszeiten:

```
favstats(arr_delay ~ season, data = flights2)
#>   season min  Q1 median Q3  max     mean       sd      n missing
#> 1 autumn -68 -18     -6 10  688 2.944260 38.08791  57374     842
#> 2 spring -86 -17     -5 15  931 7.310027 46.03570  55692    1434
#> 3 summer -68 -18     -6 16 1127 9.838901 52.59895  82378    2864
#> 4 winter -70 -15     -3 14 1272 6.600590 40.96247 131902    4290
```

Im Sommer ist die Verspätung am höchsten. Vielleicht ist es besser, gleich auf Monate hin zu untersuchen:

```
favstats(arr_delay ~ month, data = flights2)
#>    month min  Q1 median Q3  max       mean       sd     n missing
#> 1      1 -70 -15     -3 13 1272  6.1299720 40.42390 26398     606
```

```
#> 2      2 -70 -15     -3 13  834  5.6130194 39.52862 23611    1340
#> 3      3 -68 -18     -6 13  915  5.8075765 44.11919 27902     932
#> 4      4 -68 -15     -2 19  931 11.1760630 47.49115 27564     766
#> 5      5 -86 -20     -8 11  875  3.5215088 44.23761 28128     668
#> 6      6 -64 -15     -2 26 1127 16.4813296 56.13087 27075    1168
#> 7      7 -66 -16     -2 27  989 16.7113067 57.11709 28293    1132
#> 8      8 -68 -18     -5 14  490  6.0406524 42.59514 28756     571
#> 9      9 -68 -23    -12  1 1007 -4.0183636 39.71031 27010     564
#> 10    10 -61 -18     -7  7  688 -0.1670627 32.64986 28618     271
#> 11    11 -67 -16     -6  8  796  0.4613474 31.38741 26971     297
#> 12    12 -68 -11      2 25  878 14.8703553 46.13311 27020    1115
```

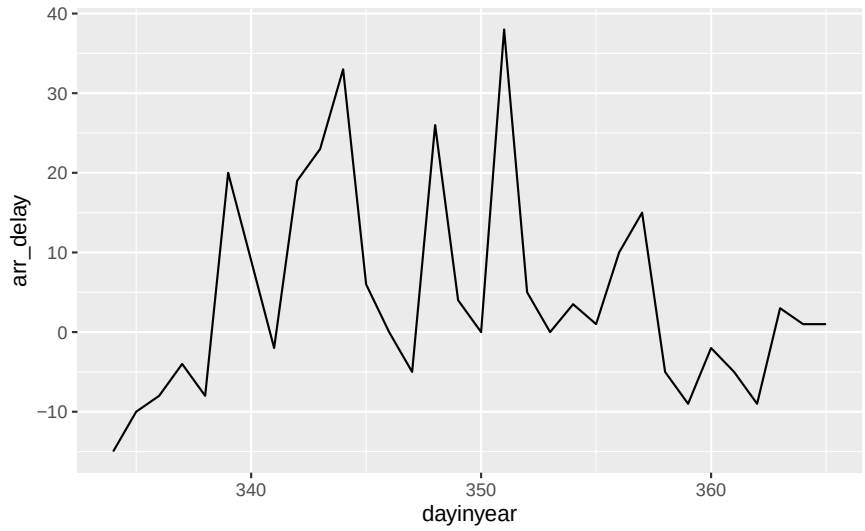
Tatsächlich ist die Verspätung im Mittelwert am höchsten im Juni und Juli. Dabei ist zu beachten, dass die *mediane* Verspätung nur im Dezember positiv ist: Nur im Dezember haben die Flüge in New York im Median eine Verspätung.

Weihnachten

Liegt es an Weihnachten? Schauen wir uns die Tage im Dezember (und Januar?) genauer an. Dazu berechnen wir zuerst einen Spalte, die den Tag (und die Woche) berechnet.

```
flights3 <- flights2 %>%
  mutate(dayinyear = yday(time_hour),
         day_id = 365-(365-dayinyear),
         week = week(time_hour))
```

```
flights3 %>%
  filter( (time_hour > "2013-11-30 23:59:59") ) %>%
  group_by(dayinyear) %>%
  summarise(arr_delay = median(arr_delay, na.rm = TRUE)) %>%
  gf_line(arr_delay ~ dayinyear, data = .)
```



Etwa zwei Wochen vor Jahresende, also noch deutlich vor den Feiertagen, kommt es zu den Verspätungsspitzen. Ob zu dieser Zeit die meisten Menschen in den Weihnachtsurlaub fliegen? Insgesamt lässt diese Betrachtung offenbar keine starken Schlüsse zu.

Wochenende vs. Werktage

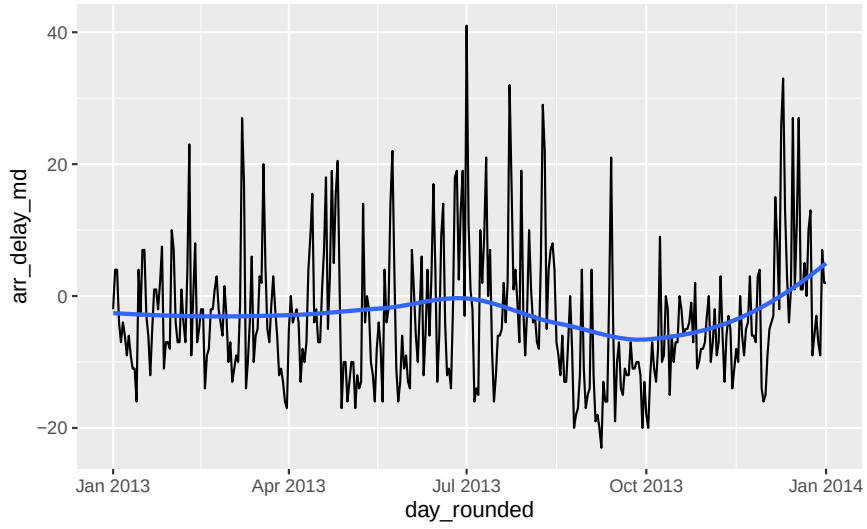
Vielleicht sind die Wochentage die entspannten Tage ohne Verspätung? Schauen wir nach. Man beachte, dass die Woche in Amerika mit Sonntag (1) beginnt, demzufolge ist der Samstag der 7. Tag.

```
flights3 %>%
  mutate(weekend = if_else(wday(time_hour) %in% c(1,7), TRUE, FALSE)) %>%
  group_by(weekend) %>%
  summarise(arr_delay_md = median(arr_delay, na.rm = TRUE))
#> # A tibble: 2 x 2
#>   weekend arr_delay_md
#>   <lgl>          <dbl>
#> 1 FALSE             -4
#> 2 TRUE              -8
```

Aha, das sind 4 Minuten weniger im Median am Wochenende (im Vergleich zu werktags). Im Verhältnis zur Streuung von 31 Minuten (IQR) ist das nicht Nichts, aber auch nicht die Welt.

Verspätung pro Tag

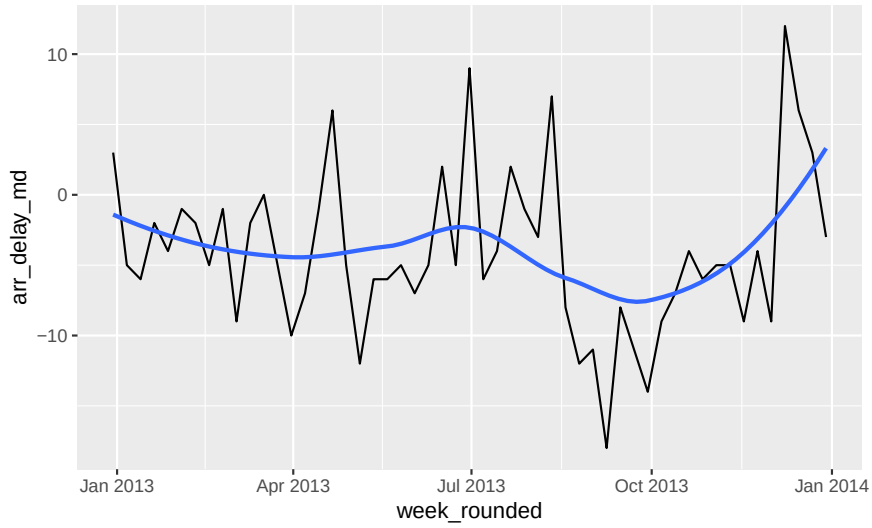
```
flights4 <- flights3 %>%
  mutate(day_rounded = round_date(time_hour, "day"))

flights4 %>%
  group_by(day_rounded) %>%
  summarise(arr_delay_md = median(arr_delay, na.rm = TRUE)) %>%
  gf_line(arr_delay_md ~ day_rounded, data = .) %>%
  gf_smooth()
```



Die Spitzen sind so nicht direkt erschließbar. Betrachten wir abschließend die Verspätungen pro Woche.

```
flights4 %>%
  mutate(week_rounded = round_date(time_hour, "week")) %>%
  group_by(week_rounded) %>%
  summarise(arr_delay_md = median(arr_delay, na.rm = TRUE)) %>%
```

```
  gf_line(arr_delay_md ~ week_rounded, data = .) %>%
  gf_smooth()
```



Der Zacken im Juli könnte mit dem Nationalfeierag in den USA zusammenhängen. Lassen wir diese Untersuchungen an dieser Stelle.

Wetter

Die Wetterdaten sind in einer anderen Tabelle (`weather`), auch im Paket `nycflights13` gespeichert. Über Datum/Zeit können wir die Wetterdaten mit den Flugdaten zusammenführen. Dabei begnügen wir uns mit einer tagesgenauen Präzision, da die Wetterdaten nicht jede Stunde (Minute, Sekunde) abdecken.

```
data(weather)
glimpse(weather)
#> Observations: 26,115
#> Variables: 15
#> $ origin     <chr> "EWR", "EWR", "EWR", "EWR", "EWR", "EWR", "EWR", "E...
#> $ year       <dbl> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 201...
#> $ month      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
#> $ day        <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
#> $ hour       <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, ...
#> $ temp       <dbl> 39.02, 39.02, 39.02, 39.92, 39.02, 37.94, 39.02, 39...
#> $ dewp       <dbl> 26.06, 26.96, 28.04, 28.04, 28.04, 28.04, 28.04, 28...
#> $ humid      <dbl> 59.37, 61.63, 64.43, 62.21, 64.43, 67.21, 64.43, 62...
#> $ wind_dir   <dbl> 270, 250, 240, 250, 260, 240, 240, 250, 260, 260, 2...
#> $ wind_speed <dbl> 10.35702, 8.05546, 11.50780, 12.65858, 12.65858, 11...
#> $ wind_gust  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ precip     <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ pressure   <dbl> 1012.0, 1012.3, 1012.5, 1012.2, 1011.9, 1012.4, 101...
#> $ visib      <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,...
#> $ time_hour  <dttm> 2013-01-01 01:00:00, 2013-01-01 02:00:00, 2013-01-...

weather2 <- weather %>%
  mutate(date_time = round_date(time_hour, "hour"),
```

```
        day_rounded = round_date(time_hour, "day")) %>%
  group_by(day_rounded) %>%
  summarise_at(vars(temp, humid, wind_speed, precip, visib), median, na.rm = TRUE)
```

```
flights5 <- flights4 %>%
  inner_join(weather2, by = c("day_rounded" = "day_rounded"))
```

Wie ist die Korrelation der Wetterdaten mit der Verspätung?

```
flights5 %>%
  select(temp, humid, wind_speed, precip, visib, arr_delay) %>%
  correlate() %>%
  focus(arr_delay)
#> # A tibble: 5 x 2
#>   rowname    arr_delay
#>   <chr>          <dbl>
#> 1 temp          0.0216
#> 2 humid         0.150
#> 3 wind_speed    0.0651
#> 4 precip        0.0465
#> 5 visib        -0.0702
```

Gut, etwas Zusammenhang mit Luftfeuchtigkeit (`humid`), aber ansonsten nicht viel zu sehen. Apropos sehen: Schlechte Sicht geht mit *weniger* Verspätung einher (?).

Modellierung

Datensatz bereinigen

Variablen ohne Varianz

Variablen ohne Varianz sind wertlos für die Vorhersage, also entfernen wir sie. `caret` bietet dazu `nearZeroVar`. Natürlich kann man sich auch die Daten mit bloßem Auge ansehen, dann fällt auf, dass `year` den konstanten Wert 2013 aufweist.

```
flights6 <- flights5 %>%
  select(-year)
```

Fehlende Werte

Probieren wir die rabiate Methode:

```
flights7 <- flights6 %>%
  drop_na()
```

Wie viel Prozent der Fälle haben wir verloren?

```
nrow(flights7)/nrow(flights6)
#> [1] 0.9719892
```

Etwa 3%, das verschmerzen wir.

Z-Skalieren

Für viele Algorithmen ist es nötig (z.B. neuronale Netze), die Prädiktoren vorab zu standardisieren hinsichtlich Mittelwert und Streuung (z-Transformation). Das kann man z.B. so erreichen (via Paket `sjmisc`):

```
flights7a <- std(flights7, suffix = "")
```

Kreuzvalidierungsmethode

Um Überanpassung zu vermeiden, verwenden wir eine 5-fach-Kreuzvalidierung.

```
my_crossval <- trainControl(method = "cv",
                            number = 5,
                            allowParallel = TRUE,
                            verboseIter = FALSE)
```

`allowParallel` erlaubt die Verwendung mehrerer Rechenkerne, sofern initialisiert:

```
doMC::registerDoMC(cores = 2)
```

Achtung: Verdoppeln wir die Anzahl der Kerne, verdoppeln wir damit auch die Menge des benötigten Speichers.

Datensatz reduzieren

Große Datensätze bringen einen Rechner leicht aus der Ruhe. Besonders kategoriale Variablen mit vielen Stufen sind schwierig, da sie (manuell oder je nach Funktion automatisch) in Dummy-Variablen umgewandelt werden müssen.

Begrenzen wir uns daher auf metrische Variablen.

```
flights8 <- flights7a %>%
  select_if(is.numeric)
```

Außerdem dürfen wir nicht vergessen, die andere Verspätungsvariable zu entferne (`dep_delay`).

```
flights9 <- flights8 %>%
  select(-dep_delay)
```

Redundante Variablen

Haben wir noch redundante Variablen?

```
findLinearCombos(flights9)
#> $linearCombos
#> $linearCombos[[1]]
#> [1] 12  4 11
#>
#> $linearCombos[[2]]
#> [1] 14 13
#>
```

```
#>
#> $remove
#> [1] 12 14
```

Ja!

Entfernen wir sie:

```
flights9a <- flights9 %>%
  select(-c(12,14))
```

Datensatz aufteilen

Teilen wir den Datensatz zu 80% in einen Übungsteil bzw. zu 20% in einen Testdatensatz auf.

```
n_uebung <- round(.8 * nrow(flights9a), digits = 0)

uebung_index <- sample(1:nrow(flights9a), size = n_uebung)

uebung_df <- filter(flights9a, row_number() %in% uebung_index)
test_df <- filter(flights9a, !(row_number() %in% uebung_index))
```

Die Gesamtfallzahl muss der Summe aus Übungs- und Test-Datensatz entsprechen:

```
(nrow(uebung_df) + nrow(test_df)) == nrow(flights9a)
#> [1] TRUE
```

Passt.

Modell 1 - Regression

Beginnen wir mit einer einfachen Regression (ohne Interaktionen, Polynome, etc.).[1]

Eine Regression hat keine Tuningparameter.

```
start <- Sys.time()
lm_fit1 <- train(arr_delay ~ .,
                 data = uebung_df,
                 method = "lm",
                 trControl = my_crossval)
end <- Sys.time()

(time_taken_lm1 <- end - start)
#> Time difference of 25.82482 secs


#saveRDS(lm_fit1, file = "lm_fit1.rds")
```

Ohne die Begrenzung auf numerische Variablen hat meine Maschine (16GB Speicher) einen Asthmaanfall bekommen und ist steckengeblieben.[2]

[1] vgl. https://topepo.github.io/caret/train-models-by-tag.html#linear-regression
[2] vgl. https://stackoverflow.com/questions/51248293/error-vector-memory-exhausted-limit-reached-r-3-5-0-macos?rq=1

Das erzeugte Modell hatte in der Datei eine Größe von ca. 360MB.

Die Koeffizienten des Modells lassen sich auf übliche Weise bestimmen:

```
summary(lm_fit1)
#>
#> Call:
#> lm(formula = .outcome ~ ., data = dat)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.1331 -0.4664 -0.1819  0.1809 29.4201
#>
#> Coefficients:
#>                  Estimate Std. Error  t value Pr(>|t|)
#> (Intercept)     0.0006767  0.0017679    0.383  0.70189
#> month           8.5536830  0.3026523   28.262  < 2e-16 ***
#> day             0.7325911  0.0255662   28.655  < 2e-16 ***
#> dep_time        0.7870591  0.0061406  128.173  < 2e-16 ***
#> sched_dep_time -0.5949524  0.0435803  -13.652  < 2e-16 ***
#> arr_time       -0.2712591  0.0029136  -93.102  < 2e-16 ***
#> sched_arr_time  0.0960639  0.0036036   26.657  < 2e-16 ***
#> flight          0.0371456  0.0020263   18.332  < 2e-16 ***
#> air_time        1.6411560  0.0141639  115.869  < 2e-16 ***
#> distance       -1.6623510  0.0142112 -116.975  < 2e-16 ***
#> hour            0.1226310  0.0430038    2.852  0.00435 **
#> dayinyear      -8.9888118  0.3186638  -28.208  < 2e-16 ***
#> week            0.3882411  0.0910250    4.265    2e-05 ***
#> temp            0.0742258  0.0023826   31.154  < 2e-16 ***
#> humid           0.1524809  0.0022081   69.055  < 2e-16 ***
#> wind_speed      0.0971761  0.0019647   49.461  < 2e-16 ***
#> precip         -0.0012373  0.0019184   -0.645  0.51894
#> visib           0.0063319  0.0022010    2.877  0.00402 **
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.9041 on 261512 degrees of freedom
#> Multiple R-squared:  0.1867, Adjusted R-squared:  0.1867
#> F-statistic:  3532 on 17 and 261512 DF,  p-value: < 2.2e-16
```

Die Prädiktorenrelevanz kann man über `varImp()` abfragen.

```
varImp(lm_fit1)
#> lm variable importance
#>
#>               Overall
#> dep_time      100.000
#> distance       91.219
#> air_time       90.352
#> arr_time       72.499
#> humid          53.643
#> wind_speed     38.279
#> temp           23.923
#> day            21.964
```

```
#> month           21.656
#> dayinyear       21.613
#> sched_arr_time  20.397
#> flight          13.869
#> sched_dep_time  10.199
#> week             2.839
#> visib            1.750
#> hour             1.730
#> precip           0.000
```

Modell 2 - Random Forest

Im Gegensatz zur Regression gibt es bei Random-Forest-Modellen Tuningparameter, und zwar die Anzahl der Variablen pro Baum, hier mit `.mtry` bezeichnet. Eine Faustregel für diesen Parameter ist $\sqrt(k)$, hier also etwa oder 6.

```
rf_grid <- data.frame(
  .mtry = c(4, 5, 6, 7),
  .splitrule = "variance",
  .min.node.size = 5)

rf_grid
#>   .mtry .splitrule .min.node.size
#> 1     4   variance              5
#> 2     5   variance              5
#> 3     6   variance              5
#> 4     7   variance              5
```

Um Zeit zu sparen, verringern wir die Stichprobengröße auf 1000:

```
uebung_df_small <- sample_n(uebung_df, size = 1000)
```

Dann berechnen wir das Modell; gibt man keine Hinweise auf Variation von Tuningparametern, so wählt die Funktion Standardwerte.

```
start <- Sys.time()
rf_fit1 <- train(arr_delay ~ .,
                 data = uebung_df_small,
                 method = "ranger",
                 trControl = my_crossval)
end <- Sys.time()

(time_taken <- end - start)
#> Time difference of 23.08801 secs

#saveRDS(rf_fit1, file = "lm_fit1.rds")
#readRDS("lm_fit1.rds")
```

Einen Überblick über das berechnete Modell kann man sich so ausgeben lassen:

```
rf_fit1
#> Random Forest
#>
#> 1000 samples
#>   17 predictor
#>
#> No pre-processing
#> Resampling: Cross-Validated (5 fold)
#> Summary of sample sizes: 800, 801, 801, 799, 799
#> Resampling results across tuning parameters:
#>
#>   mtry  splitrule   RMSE       Rsquared   MAE
#>    2    variance    0.8253691  0.4129177  0.5272027
#>    2    extratrees  0.8590837  0.3932634  0.5362638
#>    9    variance    0.6857243  0.5992653  0.4387015
#>    9    extratrees  0.7129957  0.5855666  0.4599433
#>   17    variance    0.6646842  0.6178340  0.4125599
#>   17    extratrees  0.6652444  0.6377894  0.4281044
#>
#> Tuning parameter 'min.node.size' was held constant at a value of 5
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were mtry = 17, splitrule =
#>  variance and min.node.size = 5.
```

Im resultierenden Objekt sind eine Vielzahl von Informationen zu finden. So kann man sich den Modellkandidaten mit den besten Werten ausgeben lassen:

```
rf_fit1$bestTune
#>   mtry splitrule min.node.size
#> 5   17  variance             5
```

Aber was ist das Kriteriem, das optimiert wird?

```
rf_fit1$metric
#> [1] "RMSE"
```

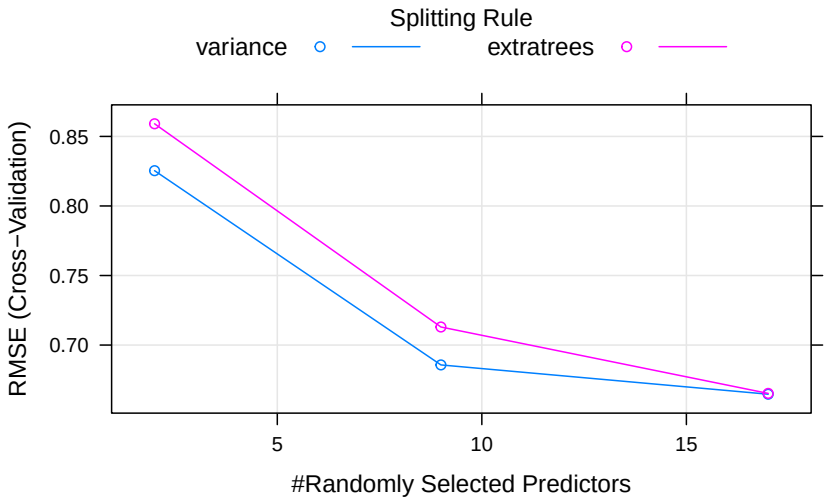
Es wird nach dem *Root Mean Square Error* optimiert.

Weitere Infos zum Algorithmus bekommt man z.B. so:

```
modelLookup("ranger")
#>    model     parameter                         label forReg forClass
#> 1 ranger          mtry #Randomly Selected Predictors   TRUE     TRUE
#> 2 ranger     splitrule               Splitting Rule   TRUE     TRUE
#> 3 ranger min.node.size           Minimal Node Size   TRUE     TRUE
#>   probModel
#> 1      TRUE
#> 2      TRUE
#> 3      TRUE
```

Wir sehen, dass das Modell drei Tuningparameter hat (wobei `caret` den Parameter `min.node.size` konstant hielt).

```
plot(rf_fit1)
```



Möchte man eine bestimmte Anzahl an Kandidatenmodelle prüfen lassen, so kann man das mit `tuneLength` tun:

```
start <- Sys.time()
rf_fit2 <- train(arr_delay ~ .,
                 data = uebung_df_small,
                 method = "ranger",
                 trControl = my_crossval,
                 tuneLength = 4)
end <- Sys.time()


(time_taken <- end - start)
#> Time difference of 36.44905 secs

saveRDS(rf_fit2, file = "lm_fit2.rds")
```

```
rf_fit2
#> Random Forest
#>
#> 1000 samples
#>   17 predictor
#>
#> No pre-processing
#> Resampling: Cross-Validated (5 fold)
#> Summary of sample sizes: 799, 800, 800, 801, 800
#> Resampling results across tuning parameters:
#>
#>   mtry  splitrule   RMSE       Rsquared   MAE
#>    2    variance    0.8203238  0.4175518  0.5321533
#>    2    extratrees  0.8525036  0.4032347  0.5362929
#>    7    variance    0.7002828  0.5966789  0.4562868
```

```
#>    7    extratrees  0.7199919  0.5847068  0.4712169
#>   12    variance    0.6630210  0.6427126  0.4264476
#>   12    extratrees  0.6755618  0.6339950  0.4431332
#>   17    variance    0.6575474  0.6444369  0.4155007
#>   17    extratrees  0.6563739  0.6510467  0.4311083
#>
#> Tuning parameter 'min.node.size' was held constant at a value of 5
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were mtry = 17, splitrule =
#>  extratrees and min.node.size = 5.
```

Mit `tuneGrid` kann man die Werte der Modellkandidaten genau einstellen.

```
start <- Sys.time()
rf_fit3 <- train(arr_delay ~ .,
                 data = uebung_df_small,
                 method = "ranger",
                 trControl = my_crossval,
                 tuneGrid = rf_grid)
end <- Sys.time()


(time_taken <- end - start)
#> Time difference of 8.39345 secs

# saveRDS(rf_fit3, file = "lm_fit3.rds")
```

```
rf_fit3
#> Random Forest
#>
#> 1000 samples
#>   17 predictor
#>
#> No pre-processing
#> Resampling: Cross-Validated (5 fold)
#> Summary of sample sizes: 800, 800, 801, 799, 800
#> Resampling results across tuning parameters:
#>
#>   mtry  RMSE       Rsquared   MAE
#>   4     0.7553441  0.5083364  0.4943232
#>   5     0.7254933  0.5542789  0.4785002
#>   6     0.7045373  0.5814886  0.4632984
#>   7     0.6926108  0.6007891  0.4535480
#>
#> Tuning parameter 'splitrule' was held constant at a value of
#>  variance
#> Tuning parameter 'min.node.size' was held constant at a value
#>  of 5
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were mtry = 7, splitrule =
#>  variance and min.node.size = 5.
```

Modell 3 - Neuronales Netz

Neuronale Netze benötigen große Datensätze, daher ist unser kleiner Datensatz mit $n = 1000$ sicher zu klein (gerade in Anbetracht zur Zahl der Features). Aus Vergleichsbarkeitsgründen und um die Rechenkosten zu schätzen, bietet es sich aber an, zunächst mit einem kleinen Datensatz zu arbeiten:

```
start <- Sys.time()
nn_fit1 <- train(arr_delay ~ .,
                 data = uebung_df_small,
                 method = "nnet",
                 trControl = my_crossval,
                 linout = TRUE)
#> # weights:  96
#> initial  value 1293.844765
#> iter  10 value 810.417182
#> iter  20 value 599.392707
#> iter  30 value 449.445228
#> iter  40 value 364.643107
#> iter  50 value 289.611282
#> iter  60 value 238.267314
#> iter  70 value 209.798648
#> iter  80 value 175.217991
#> iter  90 value 162.966931
#> iter 100 value 152.345308
#> final  value 152.345308
#> stopped after 100 iterations
end <- Sys.time()


(time_taken <- end - start)
#> Time difference of 4.283968 secs

saveRDS(nn_fit1, file = "nn_fit1.rds")
```

Der Parameter `linout = TRUE` verhindert eine Aktivierungsfunktion, die den Wertebereich auf [0,1] beschränken würde.

Das ging schnell. Vergrößen wir den Datensatz:

```
start <- Sys.time()
nn_fit2 <- train(arr_delay ~ .,
                 data = uebung_df,
                 method = "nnet",
                 trControl = my_crossval,
                 linout = TRUE)
#> # weights:  96
#> initial  value 282715.082894
#> iter  10 value 231584.339546
#> iter  20 value 178470.397954
#> iter  30 value 153481.591084
#> iter  40 value 135576.240699
#> iter  50 value 116695.609467
#> iter  60 value 110859.369898
#> iter  70 value 103110.157901
```

```
#> iter  80 value 95147.150020
#> iter  90 value 85477.968554
#> iter 100 value 74358.691471
#> final  value 74358.691471
#> stopped after 100 iterations
end <- Sys.time()


(time_taken <- end - start)
#> Time difference of 13.16125 mins

# saveRDS(nn_fit2, file = "nn_fit2.rds")
```

Es gibt verschiedene Implementierungen von neuronalen Netzen, die in `caret` angesteuert werden können, z.B. `neuralnet`. Es verfügt über 3 Modellparameter:

```
modelLookup("neuralnet")
#>       model parameter                    label forReg forClass probModel
#> 1 neuralnet    layer1 #Hidden Units in Layer 1   TRUE    FALSE     FALSE
#> 2 neuralnet    layer2 #Hidden Units in Layer 2   TRUE    FALSE     FALSE
#> 3 neuralnet    layer3 #Hidden Units in Layer 3   TRUE    FALSE     FALSE
```

```
getModelInfo("neuralnet")
#> $neuralnet
#> $neuralnet$label
#> [1] "Neural Network"
#>
#> $neuralnet$library
#> [1] "neuralnet"
#>
#> $neuralnet$loop
#> NULL
#>
#> $neuralnet$type
#> [1] "Regression"
#>
#> $neuralnet$parameters
#>   parameter   class                     label
#> 1    layer1 numeric #Hidden Units in Layer 1
#> 2    layer2 numeric #Hidden Units in Layer 2
#> 3    layer3 numeric #Hidden Units in Layer 3
#>
#> $neuralnet$grid
#> function(x, y, len = NULL, search = "grid") {
#>                     if(search == "grid") {
#>                       out <- expand.grid(layer1 = ((1:len) * 2) - 1, layer2 = 0, layer3 = 0)
#>                     } else {
#>                       out <- data.frame(layer1 = sample(2:20, replace = TRUE, size = len),
#>                                         layer2 = sample(c(0, 2:20), replace = TRUE, size = len),
#>                                         layer3 = sample(c(0, 2:20), replace = TRUE, size = len))
#>                     }
#>                     out
#>                   }
```

```
#>
#> $neuralnet$fit
#> function(x, y, wts, param, lev, last, classProbs, ...) {
#>                      colNames <- colnames(x)
#>                      dat <- if(is.data.frame(x)) x else as.data.frame(x)
#>                      dat$.outcome <- y
#>                      form <- as.formula(paste(".outcome ~",paste(colNames, collapse = "+")))
#>                      if(param$layer1 == 0) stop("the first layer must have at least one hidden unit")
#>                      if(param$layer2 == 0 & param$layer2 > 0) stop("the second layer must have at lea
#>                      nodes <- c(param$layer1)
#>                      if(param$layer2 > 0) {
#>                        nodes <- c(nodes, param$layer2)
#>                        if(param$layer3 > 0) nodes <- c(nodes, param$layer3)
#>                      }
#>                      neuralnet::neuralnet(form, data = dat, hidden = nodes, ...)
#>                    }
#>
#> $neuralnet$predict
#> function(modelFit, newdata, submodels = NULL) {
#>                      newdata <- newdata[, modelFit$model.list$variables, drop = FALSE]
#>                      neuralnet::compute(modelFit, covariate = newdata)$net.result[,1]
#>                    }
#>
#> $neuralnet$prob
#> NULL
#>
#> $neuralnet$tags
#> [1] "Neural Network"
#>
#> $neuralnet$sort
#> function(x) x[order(x$layer1, x$layer2, x$layer3),]
```

Vergleich der Modellgüten

Wie gut sagen die Modelle den Test-Datensatz vorher? Vergleichen wir die Modelle.

Prognosen für den Test-Datensatz berechnen

Erstellen wir uns einen Datensatz mit den Vorhersagen (und den beobachteten Werten):

```
test_preds <- test_df %>%
  select(arr_delay) %>%
  mutate(lm1_pred = predict(lm_fit1, newdata = test_df))

test_preds <- test_df %>%
  select(arr_delay) %>%
  mutate(lm1_pred = predict(lm_fit1, newdata = test_df),
         rf1_pred = predict(rf_fit2, newdata = test_df),
         rf2_pred = predict(rf_fit3, newdata = test_df),
         nn1_pred = predict(nn_fit1, newdata = test_df),
         nn2_pred = predict(nn_fit2, newdata = test_df))
```

Jetzt lassen wir uns typische Kennzahlen der Modellgüte ausgeben:

```
postResample(pred = test_preds$lm1_pred, obs = test_preds$arr_delay)
#>      RMSE  Rsquared       MAE
#> 0.8895674 0.1924120 0.5383087
```

Der Vektor der Modellnamen lautet:

```
model_names <- names(test_preds)
```

Das wiederholen wir in einer Schleife für jedes Modell:

```
test_pred_df <- test_preds %>%
  map_df(~ postResample(pred = ., obs = test_preds$arr_delay)) %>%
  mutate(statistic = c("RMSE", "Rsquared", "MAE")) %>%
  select(statistic, everything())

test_pred_df
#> # A tibble: 3 x 7
#>   statistic arr_delay lm1_pred rf1_pred rf2_pred nn1_pred nn2_pred
#>   <chr>         <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
#> 1 RMSE              0    0.890    0.630    0.675    0.611    0.518
#> 2 Rsquared          1    0.192    0.633    0.567    0.644    0.727
#> 3 MAE               0    0.538    0.397    0.424    0.296    0.313
```

Formen wir diese Tabelle in Langform (Normalform um):

```
test_pred_df_t <- test_pred_df %>%
  gather(key = "model_name", value = "value", -c(statistic))
```

Eine andere Form wäre:

```
test_pred_df %>%
  gather(key = "model_name", value = "value", -c(statistic)) %>%
  spread(key = statistic, value = value)
#> # A tibble: 6 x 4
#>   model_name   MAE  RMSE Rsquared
#>   <chr>      <dbl> <dbl>    <dbl>
#> 1 arr_delay  0     0        1
#> 2 lm1_pred   0.538 0.890    0.192
#> 3 nn1_pred   0.296 0.611    0.644
#> 4 nn2_pred   0.313 0.518    0.727
#> 5 rf1_pred   0.397 0.630    0.633
#> 6 rf2_pred   0.424 0.675    0.567
```

Bestes Modell identifizieren

Der kleinste RMSE-Wert (nach dem Modell, dass als vorhergesagten Werte die beobachteten nimmt, also einen RSME von Null hat):

```
test_pred_df_t %>%
  filter(statistic == "RMSE") %>%
  top_n(2, wt = -value)
#> # A tibble: 2 x 3
#>   statistic model_name value
#>   <chr>     <chr>      <dbl>
#> 1 RMSE      arr_delay  0
#> 2 RMSE      nn2_pred   0.518
```

Der größte R^2-Wert:

```
test_pred_df_t %>%
  filter(statistic == "Rsquared") %>%
  top_n(2, wt = value)
#> # A tibble: 2 x 3
#>   statistic model_name value
#>   <chr>     <chr>      <dbl>
#> 1 Rsquared  arr_delay  1
#> 2 Rsquared  nn2_pred   0.727
```

Visualisieren

```
test_pred_df_t %>%
  group_by(statistic) %>%
  mutate(is_max = value == max(value),
         is_min = value == min(value)) %>%
  ggplot(aes(y = model_name, x = value, color = is_max, shape = is_min)) +
  geom_point(size = 5) +
  facet_wrap(~statistic, scales = "free_x") +
  theme(legend.position = "bottom")
```

Damit hat das neuronale Netz “gewonnen”.

Fazit

Es darf nicht vergessen werden, dass wir nur einen Teil des Datensatzes verwendet haben - schlicht aus Gründen der komputationalen Kostensparung. Insofern sind die Modellgüten nur bedingt für bare Münze zu nehmen. Diese Fallstudie hat nur einen Teil der Möglichkeiten einer ernsthaften Modellierung aufgenommen, so dass die Ergebnisse schon aus diesem Grund mit einem großen Gramm Salz zu betrachten sind. Außerdem fanden nur relativ weniger Modell Eingang; es bleibt also offen, ob nicht andere Modelle "besser" sind. Beim Wort "besser" muss man immer im Kopf behalten, dass "besser" eine *bedingte* Aussage ist: *besser* vor dem Hintergrund gewisser anderer Modelle, gewisser Transformationen, gewisser Implementierungen, gewisser Stichprobenmerkmale, gewisser Implementierungsspezifika und so weiter.