

# DataScience\_FinalProject

Jan-Philipp Steinwender & Paul Pavlis

2020-10-28

## Clean the environment variables (*Temporary*)

*Remove this whole header before handing the project in. This is just so that working with the document is easier*

```
rm(list = ls())
```

## Load needed libraries

```
library(tidyverse)
```

```
## -- Attaching packages ---

## v ggplot2 3.3.2     v purrr    0.3.4
## v tibble   3.0.3     v dplyr    1.0.2
## v tidyverse 1.1.2     v stringr  1.4.0
## v readr    1.3.1     vforcats  0.5.0

## -- Conflicts ---
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
```

```
library(ggplot2)
library(caret)
```

```
## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
## 
##      lift
```

```

library(randomForest)

## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':
##       combine

## The following object is masked from 'package:ggplot2':
##       margin

library(e1071)
library(nnet)
library(rpart)
library(keras)

## Warning: package 'keras' was built under R version 4.0.3

```

## Data wrangling

### Data import from csv file

Die Daten werden mit `read.csv` eingelesen, da dadurch alle Spalten richtig eingelesen werden, was aufgrund der Spalte **Cuisine.Style** anders nicht so leicht geht, da diese Spalte ein Python List object enthält, was als Trennzeichen das Gleiche Trennzeichen verwendet wie bei der Trennung der verschiedenen Spalten.

Außerdem werden leere Einträge durch NA Werte ersetzt, da Funktionen mit diesen besonders umgehen können.

```

restaurant_data = read.csv(file = "restaurants_data.csv", encoding = "UTF-8")
# Set empty data entries to NA
restaurant_data = restaurant_data %>% na_if("")
```

## Clean the data

### Rename and remove columns

Die Spaltennamen werden auf angenehmerer Art dargestellt und die Spalten welche keine benötigten Infos liefern e.gg **X**, **URL\_TA** & **ID\_TA** werden entfernt.

```

restaurant_data = as_tibble(restaurant_data)

restaurant_data = restaurant_data %>%
  rename("Cuisine_Style" = Cuisine.Style) %>%
  rename("Price_Range" = Price.Range) %>%
  rename("Review_Count" = Number.of.Reviews)

# Remove useless columns
restaurant_data = restaurant_data %>%
  mutate(X = NULL, URL_TA = NULL, ID_TA = NULL)

```

### Correct the column types

Die Spalte **City** wird von einer Textvariable zu einer kategorialen Faktorvariable geändert, da die Städte öfter vorkommen und dadurch mehr Infos aus der Spalte geholt werden können. Diese Variable besitzt 31 Ausprägungen

Die Spalten **Price\_Range** und **Rating** werden aus den selben Gründen ebenso in einen Faktor umgewandelt. Die Variable **Price\_Range** besitzt 3 Ausprägungen, die für leichtere Verständlichkeit umbenannt werden in: *low*, *medium* & *high*. Die Variable **Rating** besitzt 9 Ausprägungen.

```

# Change from character to factor
restaurant_data = restaurant_data %>% mutate(City = as_factor(City))

# Change from character to factor
restaurant_data = restaurant_data %>% mutate(Price_Range = as_factor(Price_Range))
# Rename the levels
restaurant_data = restaurant_data %>% mutate(Price_Range = fct_recode(
  Price_Range,
  "high" = "$$$$",
  "medium" = "$$ - $$",
  "low" = "$"
))

# Set invalid entries with a rating of -1 to NA
restaurant_data$Rating = restaurant_data$Rating %>% na_if("-1")
# Change from character to factor
restaurant_data = restaurant_data %>% mutate(Rating = as_factor(Rating))

```

### Check duplicated and NULL values

Ursprünglich enthält der Datensatz 286 Datensätze die doppelt vorkommen. Diese werden entfernt.

Insgesamt enthält der Datensatz 123992 NA Werte. Diese sind wie folgt auf die einzelnen Spalten aufgeteilt:

```

duplicated(restaurant_data) %>% sum()

## [1] 289

# Remove duplicates
restaurant_data = restaurant_data %>% distinct()

```

```

is.na(restaurant_data) %>% sum()

## [1] 124022

sapply(restaurant_data, function(x) sum(is.na(x)))

##          Name        City Cuisine_Style      Ranking      Rating
## 0            0         0       31222        9370       9389
## Price_Range Review_Count     Reviews
## 47642        17062        9337

```

## Explorative Datenanalyse

Der Datensatz handelt von TripAdvisor Bewertungen vieler Restaurants von 31 europäischen Städten.

Der endgültige Datensatz mit dem wir in diesem Projekt arbeiten werden, besteht aus *acht* Variablen (Spalten) mit *125.238* Observationen (Zeilen). Diese beinhalten:

**Name:** Name des Restaurants - Textvariable (unique)

**City:** Stadt in der sich das Restaurant befindet - Kategoriale Faktorvariable mit 31 Ausprägungen (London mit 18113 Einträgen, Paris mit 14867 Einträgen, ...)

**Cuisine\_Style:** Essensrichtungen des Restaurants - Textvariable (grundsätzlich besteht diese aus mehreren Faktoren innerhalb eines Python list Objektes / 31222 NA Werte)

**Ranking:** Rang des Restaurants im Vergleich zu allen anderen Restaurants in der Stadt - Diskrete Variable (Min: 1, Mean: 3658, Median: 2256, Max: 16444 / 9370 NA Werte)

**Rating:** Bewertung des Restaurants von 1-5 in 0.5 Schritten - Kategoriale Faktorvariable mit 9 Ausprägungen (Bewertung 4 mit 39841 Einträgen, 4.5 mit 31325 Einträgen, ... / 9389 NA Werte)

**Price\_Range:** Preisbewertung - Kategoriale Faktorvariable mit 3 Ausprägungen (medium mit 54302 Einträgen, high mit 4306 Einträgen und low mit 18988 Einträgen / 47642 NA Werte)

**Review\_Count:** Anzahl der Reviews - Diskrete Variable (Min: 2, Mean: 125.2, Median: 32, Max: 16478 / 17062 NA Werte)

**Reviews:** Zwei Reviews des Restaurants und die Daten, an dem die Reviews geschrieben wurden - Textvariable (als Python list Objekt abgespeichert)

```
restaurant_data
```

```

## # A tibble: 125,238 x 8
##   Name    City Cuisine_Style  Ranking Rating Price_Range Review_Count Reviews
##   <chr>   <fct>  <chr>        <dbl> <fct>   <fct>      <dbl> <chr>
## 1 Marti~ Amste~ ['French', 'D~      1 5       medium        136 [['Just~
## 2 De Si~ Amste~ ['Dutch', 'Eu~     2 4.5      high         812 [['Grea~
## 3 La Ri~ Amste~ ['Mediterrane~  3 4.5      high         567 [['Sati~
## 4 Vinke~ Amste~ ['French', 'E~     4 5       high         564 [['True~
## 5 Libri~ Amste~ ['Dutch', 'Eu~     5 4.5      high         316 [['Best~
## 6 Ciel ~ Amste~ ['Contemporar~  6 4.5      high         745 [['A tr~
## 7 Zaza's Amste~ ['French', 'I~     7 4.5      medium       1455 [['40th~
## 8 Blue ~ Amste~ ['Asian', 'In~    8 4.5      high         675 [['Grea~

```

```

## 9 Teppa~ Amste~ ['Japanese', ~ 9 4.5 high 923 [['Grea~
## 10 Rob W~ Amste~ ['Dutch', 'Se~ 10 4.5 low 450 [['Exce~
## # ... with 125,228 more rows

str(restaurant_data)

## tibble [125,238 x 8] (S3: tbl_df/tbl/data.frame)
## $ Name : chr [1:125238] "Martine of Martine's Table" "De Silveren Spiegel" "La Rive" "Vinke...
## $ City : Factor w/ 31 levels "Amsterdam","Athens",...: 1 1 1 1 1 1 1 1 1 ...
## $ Cuisine_Style: chr [1:125238] "[French", 'Dutch', 'European']" "[Dutch", 'European', 'Vegetarian...
## $ Ranking : num [1:125238] 1 2 3 4 5 6 7 8 9 10 ...
## $ Rating : Factor w/ 9 levels "1","1.5","2",...: 9 8 8 9 8 8 8 8 8 ...
## $ Price_Range : Factor w/ 3 levels "medium","high",...: 1 2 2 2 2 2 1 2 2 3 ...
## $ Review_Count : num [1:125238] 136 812 567 564 316 ...
## $ Reviews : chr [1:125238] "[Just like home", 'A Warm Welcome to Wintry Amsterdam'], ['01/03/2018']

summary(restaurant_data)

##      Name           City      Cuisine_Style      Ranking
## Length:125238    London     :18113  Length:125238   Min.   : 1
## Class :character  Paris      :14867  Class :character  1st Qu.: 965
## Mode  :character  Madrid     : 9524  Mode  :character  Median : 2256
##                   Barcelona: 8390               Mean   : 3658
##                   Berlin    : 7073               3rd Qu.: 5237
##                   Milan     : 6680               Max.   :16444
##                   (Other)   :60591              NA's   :9370
##      Rating         Price_Range      Review_Count      Reviews
## 4       :39841  medium:54302  Min.   : 2.0  Length:125238
## 4.5     :31325  high   : 4306  1st Qu.: 9.0  Class  :character
## 3.5     :19744  low    :18988  Median : 32.0  Mode   :character
## 5       :11257  NA's   :47642  Mean   : 125.2
## 3       : 8522   NA's   :47642  3rd Qu.: 114.0
## (Other): 5160   NA's   :47642  Max.   :16478.0
## NA's   : 9389   NA's   :47642  NA's   :17062

```

## Anzahl der Restaurants pro Stadt

Hier zu sehen ist die Verteilung der Anzahl der Restaurants pro Stadt absteigend geordnet. Zusätzlich ist noch die Verteilung der Preis Einteilung pro Stadt zu sehen.

Gut zu sehen ist, dass in dem Datensatz Städte wie *London*, *Paris* sehr viele Restaurants besitzen. (Bereich ~15.000+) Danach ist ein stärkerer Abfall an Vorkommnissen zu sehen und die Städte *Luxenburg* und *Ljubljana* besitzen am wenigsten Restaurants hier. (<1000)

Auch zu erkennen ist, dass die meisten Restaurants in so gut wie allen Städten in die Preis Klasse *medium* fallen. Am wenigsten Restaurants sind in der Klasse *high*. Der Anteil an nicht klassifizierter Restaurants (NA) ist sehr hoch und ist ein beträchtlicher Anteil.

```

city_price_graph <-
  mutate(restaurant_data, City = fct_infreq(City)) %>%
  mutate(Price_Range = fct_relevel(Price_Range,
                                    "high", "medium", "low")) %>%
  ggplot(aes(x = City)) + geom_bar(aes(fill = Price_Range))

```

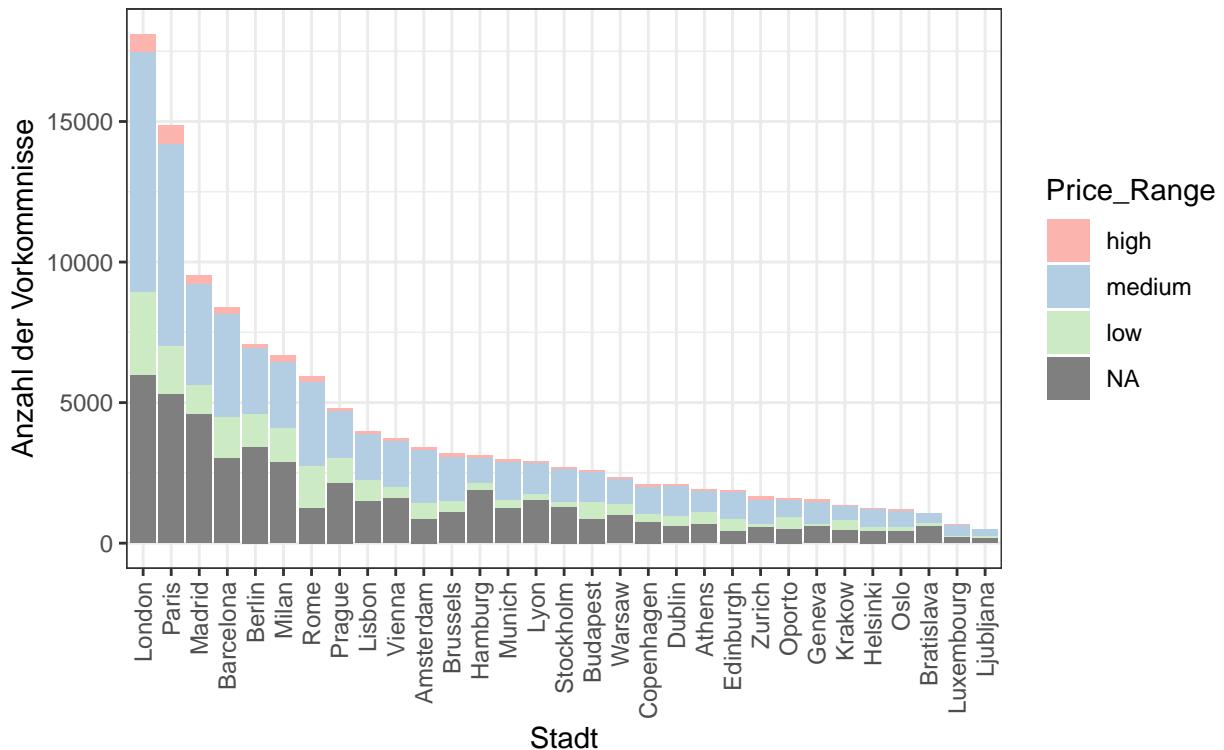
```

city_price_graph + ggtitle("Anzahl der Restaurants pro Stadt",
                           subtitle = paste0("Anzahl der Datensätze: ",
                                             length(which(
                                               !is.na(restaurant_data$City
                                             ))))) +
  xlab("Stadt") +
  ylab("Anzahl der Vorkommnisse") +
  theme_bw() +
  theme(axis.text.x = element_text(
    angle = 90,
    vjust = 0.5,
    hjust = 1
  )) +
  scale_fill_brewer(type = "qual", palette = 4, na.value = "grey50")

```

## Anzahl der Restaurants pro Stadt

Anzahl der Datensätze: 125238



## Anzahl der Restaurants Ratings

Hier zu sehen ist die Verteilung der Anzahl der Rating Stufen.

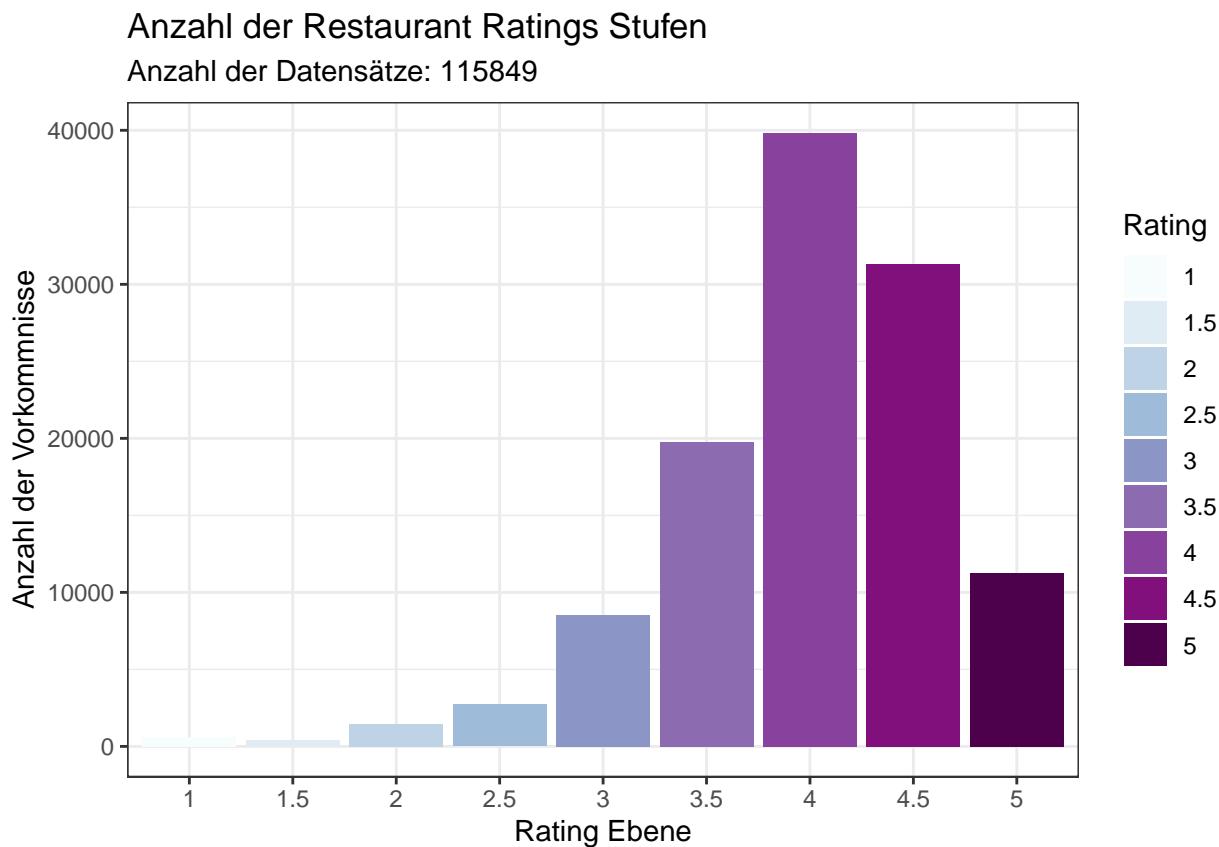
Gut zu erkennen ist, dass die Ratings 4 und 4.5 am häufigsten gegeben werden und Ratings wie 3 oder 5 viel seltener. Ratings  $\leq 2.5$  kommen am seltensten vor. Dies könnte sich dadurch erklären lassen, dass diese Restaurants sich nicht besonders gut halten werden und wahrscheinlich öfters schließen müssen als Restaurants mit besseren Bewertungen.

```

ratings_graph = restaurant_data %>%
  filter(!is.na(Rating)) %>%
  ggplot(aes(x = Rating)) + geom_bar(aes(fill = Rating))

ratings_graph + ggtitle("Anzahl der Restaurant Ratings Stufen",
                        subtitle = paste0("Anzahl der Datensätze: ", length(which(
                          !is.na(restaurant_data$Rating)
                        )))) +
  xlab("Rating Ebene") +
  ylab("Anzahl der Vorkommnisse") +
  theme_bw() +
  scale_fill_brewer(type = "seq", palette = 3)

```



## Anzahl der Restaurant Ratings pro Preisklasse

Hier zu sehen ist die Verteilung der Anzahl der Rating Stufen pro Preisklasse.

Es ist ersichtlich, dass Restaurants mit einer niedrigen Preisklasse tendenziell eher weniger hohe Bewertungen haben (im Vergleich zu den anderen Preisklassen) Restaurants in einer hohen Preisklasse hingegen besitzen höhere Ratings welche im Schnitt 4 oder höher sind.

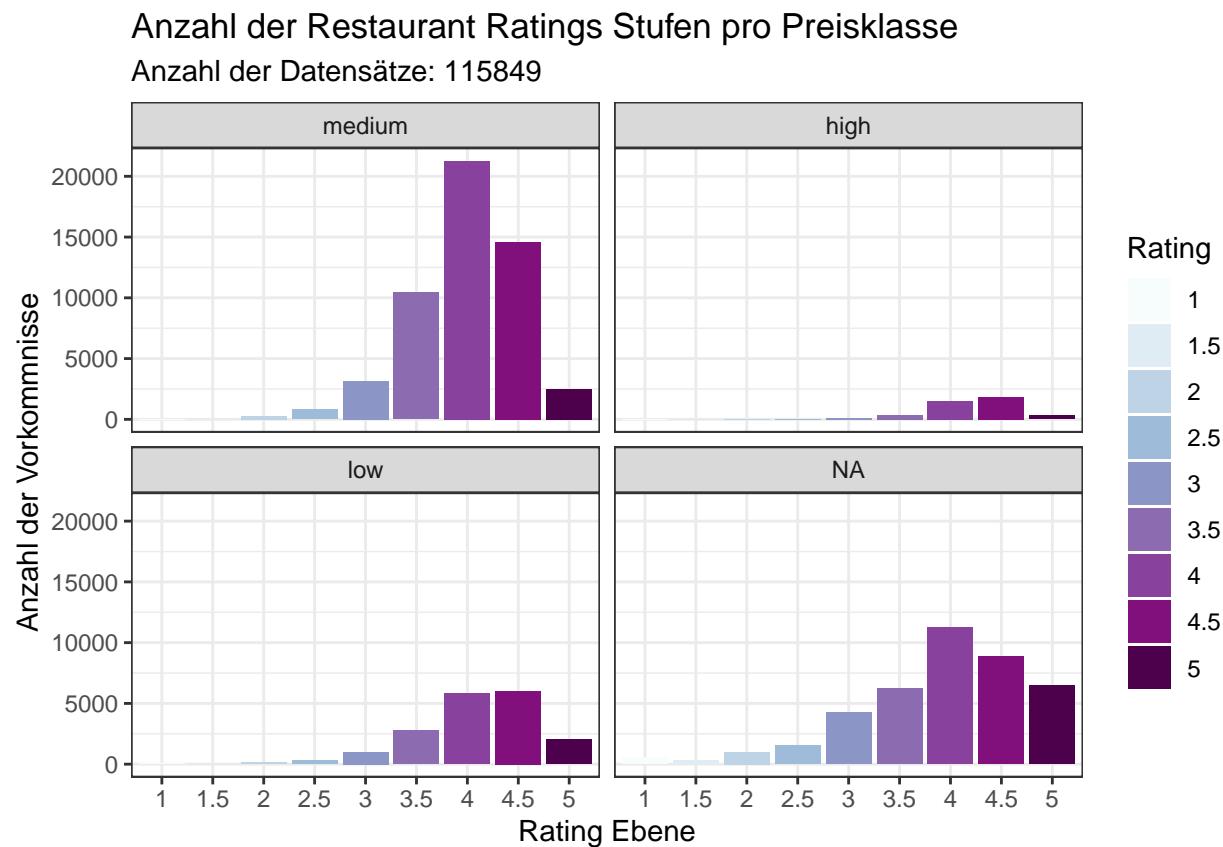
Ein weiteres interessantes Merkmal ist, dass Restaurants ohne Preisklasseneinstufung tendenziell viel mehr 5 Sterne Bewertungen haben. Dies kann entweder ein statistischer Zufall sein oder es könnte daran liegen, dass diese Restaurants nicht in traditionelle Preisklassen eingeteilt werden können und dadurch Leute besser ansprechen (was aber eher weit hergeholt ist).

```

ratings_price_graph = restaurant_data %>%
  filter(!is.na(Rating)) %>%
  ggplot(aes(x = Rating)) + geom_bar(aes(fill = Rating)) + facet_wrap(~Price_Range)

ratings_price_graph + ggtitle("Anzahl der Restaurant Ratings Stufen pro Preisklasse",
                             subtitle = paste0("Anzahl der Datensätze: ", length(which(
                               !is.na(restaurant_data$Rating)
                             )))) +
  xlab("Rating Ebene") +
  ylab("Anzahl der Vorkommnisse") +
  theme_bw() +
  scale_fill_brewer(type = "seq", palette = 3)

```



## Review Anzahl

Hier zu sehen ist die Verteilung der Anzahl der Review Bewertungen.

Dieser Graph gruppiert auf der x-Achse die Anzahl der Reviews in 50er Blöcke und stellt davon die Anzahl der Vorkommnisse auf der y-Achse dar.

Hier kann man stark betrachten, dass die Verteilung stark rechts fallend ist und die meisten Vorkommnisse in die Kategorie von 0-50 Reviews fallen. Dies bedeutet, dass die meisten Restaurants wenig Reviews haben und das nur noch ein sehr kleiner Anteil an Restaurants mehr als 500 Reviews haben.

```

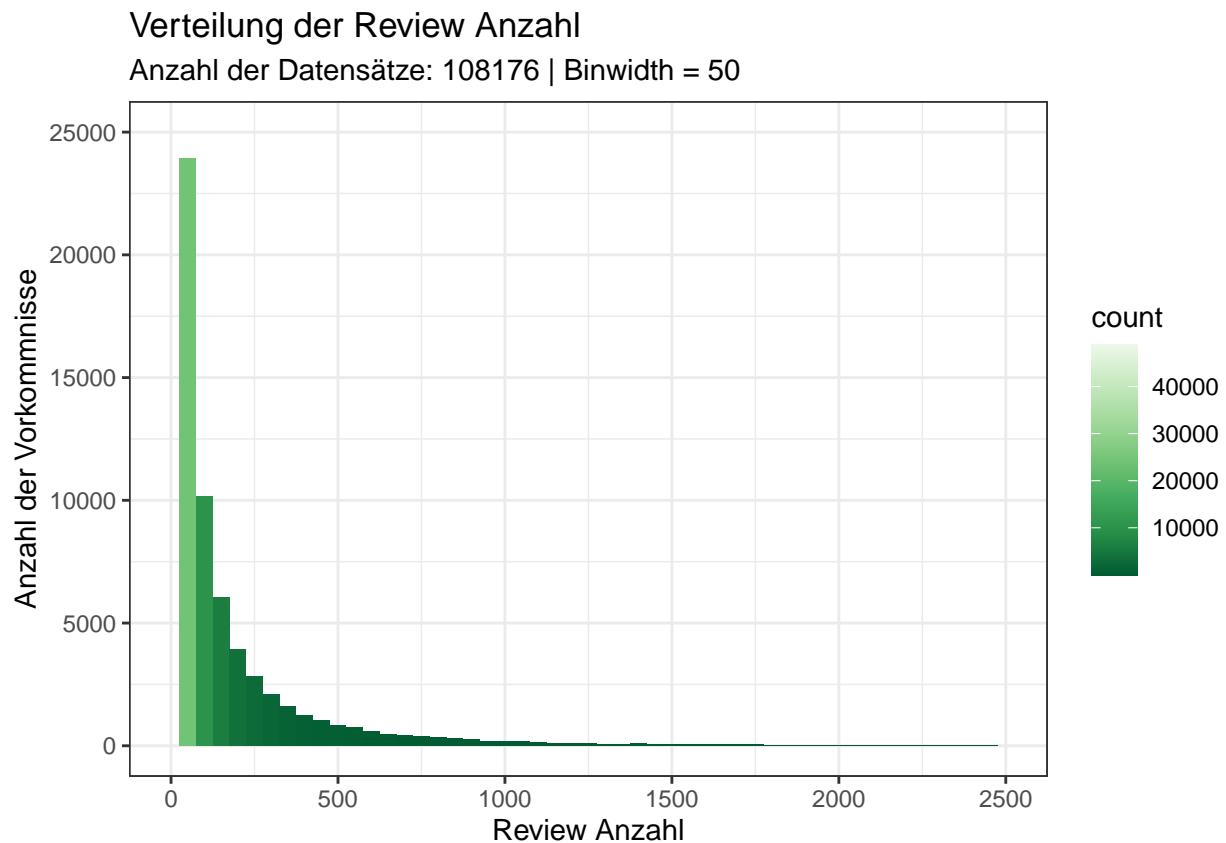
reviewCount_graph = restaurant_data %>% filter(!is.na(Review_Count)) %>% ggplot(aes(x = Review_Count))

reviewCount_graph + ggtitle("Verteilung der Review Anzahl",
                            subtitle = paste0("Anzahl der Datensätze: ", length(which(
                                !is.na(restaurant_data$Review_Count)
                            )), " | Binwidth = 50")) +
  xlab("Review Anzahl") +
  ylab("Anzahl der Vorkommnisse") +
  theme_bw() +
  xlim(0, 2500) + ylim(0, 25000) +
  scale_fill_distiller(type = "seq", palette = 5)

## Warning: Removed 250 rows containing non-finite values (stat_bin).

## Warning: Removed 2 rows containing missing values (geom_bar).

```



## Restaurant Ranking erklärt durch Review Anzahl

Hier zu sehen ist die Verteilung des Rankings von Restaurants erklärt durch die Review Anzahl.

Es ist eindeutig ersichtlich, dass Restaurants welche ein hohes Ranking in ihrer Stadt haben, eine dementsprechend höhere Anzahl an Reviews hat. Dies kann sich dadurch erklären lassen, dass Top-Restaurants öfters besucht werden und dadurch mehr Leute auch eine Bewertung schreiben.

```

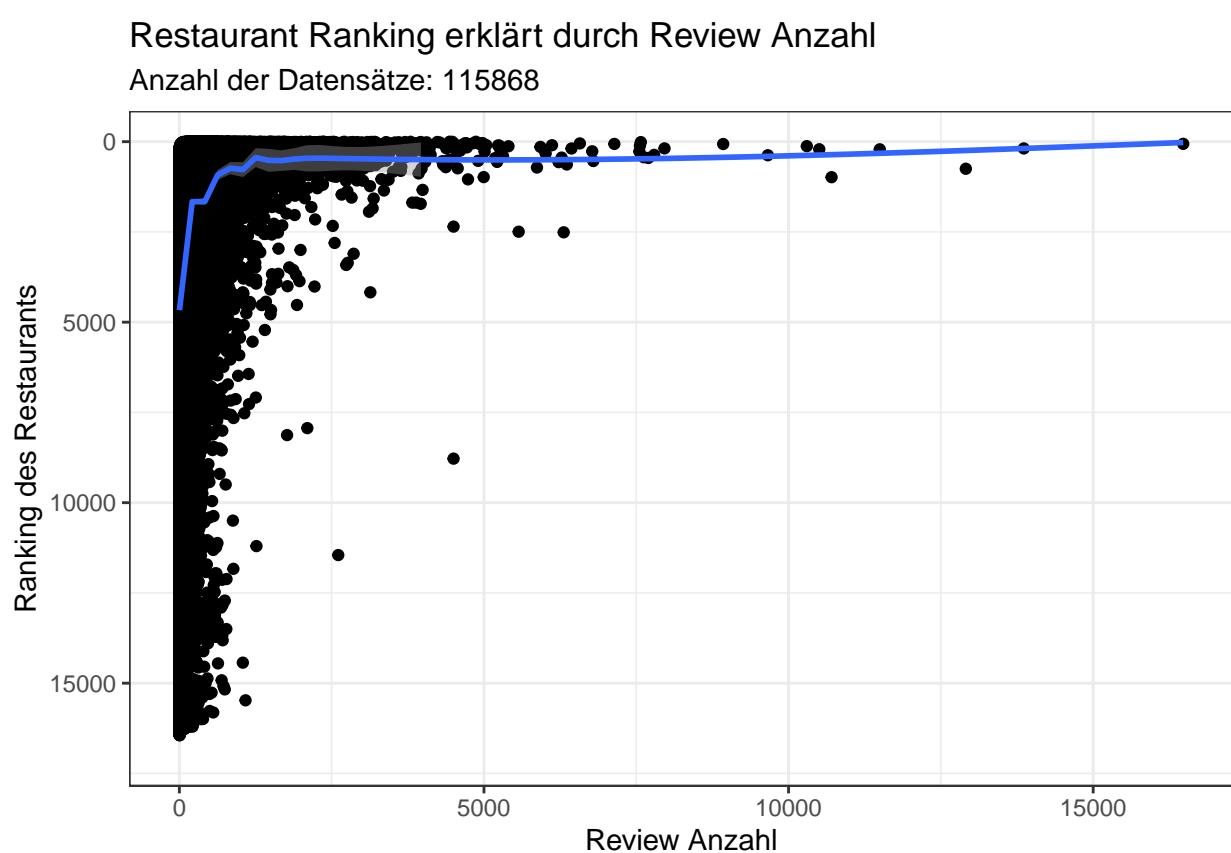
ranking_reviewCount_graph = restaurant_data %>% filter(!is.na(Review_Count)) %>% filter(!is.na(Ranking))
  geom_point() + geom_smooth(method = "gam")

ranking_reviewCount_graph + ggtitle("Restaurant Ranking erklärt durch Review Anzahl",
                                     subtitle = paste0("Anzahl der Datensätze: ", length(which(
                                         !is.na(restaurant_data$Ranking)
                                         )))) +
  xlab("Review Anzahl") +
  ylab("Ranking des Restaurants") +
  theme_bw() +
  scale_y_continuous(trans = "reverse", limits = c(17000, 1))

## 'geom_smooth()' using formula 'y ~ s(x, bs = "cs")'

```

## 'geom\_smooth()' using formula 'y ~ s(x, bs = "cs")'



## Modellierung

Auswahl des Merkmals zur Vorhersage:

Auswahl: Ranking

Das Ranking sollte ein gutes Merkmal für die Vorhersage sein, da dieser gut durch die anderen Parameter vorhersagbar sein sollte (in gewissen Grenzen) und da er recht interessant für neue Restaurants sein könnte.

Dadurch könnte man dann schauen, in welcher Stadt es sich auszahlen würde, Restaurants zu öffnen, anhand des jeweiligen Rankings im Vergleich zu den anderen Restaurants in der Stadt.

Außerdem hatten wir in der LV und bei den Übungen sehr viel mit Klassifikation zu tun und wollten dies auch einmal mit einem Regressionsbeispiel durchprobieren.

## Spezielle Aufbereitung

Name und Reviewtext bringen keinen Mehrwert beim Vorhersagen und werden deswegen im ersten Schritt entfernt. Cuisine Style bringt in dem vorhandenen Format ebenfalls nichts. Zusätzlich werden, da genügend viele Daten vorhanden sind, im nächsten Schritt alle Daten rausgehaut, bei denen Werte fehlen.

```
data_not_null = mutate(restaurant_data, Name = NULL, Reviews = NULL, Cuisine_Style = NULL) %>% drop_na()
```

## Trainings- und Testdaten kreieren

```
N = nrow(data_not_null)
train_ind = sample(1 : N, size = N * 2/3)
trainData = data_not_null[train_ind,]
testData = data_not_null[-train_ind,]

pp = preProcess(trainData, method = c("center", "scale"))
trainData_scaled = predict(pp, trainData)
testData_scaled = predict(pp, testData)
```

Leider wurde hier nicht anfänglich berücksichtigt, dass nur die Prediktoren preProcessed werden sollen. Dadurch sind die Ergebnisse der Modelle auch centered und skaliert. Um dies auszugleichen wurde eine Funktion geschrieben, die den Prozess der predicted Variablen umwandelt um dadurch menschlich lesbare Ergebnisse zu erhalten.

```
unPreProc <- function(preProc, data){
  data_digits <- data[, 1] * preProc$std[[1]] + preProc$mean[[1]]
  data_round <- round(data_digits, digits = 0)
  return(data_round)
}
```

## Linear Regression

Nicht eine von den drei erforderlichen Methoden. Wurde aus Interesse gemacht. Sie ist (wie erwartet) nicht so gut für diesen Anwendungsfall. Eventuell könnte sie noch mehr getuned werden, allerdings sind andere Methoden für diesen Fall attraktiver und (für uns) spannender.

```
m_linearRegression = lm(Ranking ~ ., data = trainData_scaled)

# summary(m_linearRegression)
```

## Berechnung Training Error

```
fitted_linearRegression = predict(m_linearRegression, trainData_scaled)
train_error_linearRegression = mean((fitted_linearRegression - trainData_scaled$Ranking)^2)

train_error_linearRegression
```

```
## [1] 0.3392307
```

### Berechnung Gerneralization Error

```
pred_linearRegression = predict(m_linearRegression, testData_scaled)
test_error_linearRegression = mean((pred_linearRegression - testData_scaled$Ranking)^2)

test_error_linearRegression
```

```
## [1] 0.3383621
```

### Tree Modell (rpart)

```
# m_rpart = rpart(Ranking ~ ., trainData_scaled)
```

### Parameter Tuning

Getestet zwischen: Minimale Anzahl in Node bevor Split: 5 - 100 Minimale Anzahl in Node: 5 - 100

Beste Parameter beim Erstelldruckgang: Keine Verbesserung -> Nehmen von 50 in einer Node (minbucket)

```
# tune_rpart = tune.rpart(Ranking ~ ., data = trainData_scaled,
#                           # # Dauert recht lange, man sieht aber schön
#                           # # wo die Grenzen sind
#                           # minsplit = seq(1, 10000, by = 100),
#                           # minbucket = seq(1, 10000, by = 100),
#                           minsplit = seq(5, 100, by = 5),
#                           minbucket = seq(5, 100, by = 2),
#                           tunecontrol = tune.control(sampling = "fix",
#                                                       fix = 2/3))
#
#
# tune_rpart
# plot(tune_rpart, color.palette = topo.colors)
```

### Erstellung des getunten Modells

```
m_rpart = rpart(Ranking ~ ., data = trainData_scaled, minbucket = 50)
```

### Berechnung Training Error

```
fitted_rpart = predict(m_rpart, trainData_scaled)
train_error_rpart = mean((fitted_rpart - trainData_scaled$Ranking)^2)

train_error_rpart
```

```
## [1] 0.1989144
```

## Berechnung Gerneralization Error

```
pred_rpart = predict(m_rpart, testData_scaled)
test_error_rpart = mean((pred_rpart - testData_scaled$Ranking)^2)

test_error_rpart

## [1] 0.198018
```

## RandomForest

```
# m_randomForest = randomForest(Ranking ~ ., data = trainData_scaled, importance = TRUE, ntree = 100)
```

### Parameter Tuning

Getestet zwischen: Anzahl der Bäume: 50 - 500 Minimale Größe der einzelnen Nodes: 10 - 1000

Beste Parameter beim Erstelldruckgang: nodesize: 40 | ntree: 100

```
# tune_randomForest = tune.randomForest(Ranking ~ ., data = trainData_scaled,
#                                         ntree = seq(50, 500, by = 20),
#                                         nodesize = seq(10, 100, by = 10),
#                                         tunecontrol = tune.control(sampling = "fix",
#                                         fix = 2/3))
#
#
# tune_randomForest
# plot(tune_randomForest, color.palette = topo.colors)
```

## Erstellung des getunten Modells

```
m_randomForest = randomForest(Ranking ~ ., data = trainData_scaled, importance = TRUE, ntree = 100, nodesize = 40)
```

### Berechnung Training Error

```
fitted_randomForest = predict(m_randomForest, trainData_scaled)
train_error_randomForest = mean((fitted_randomForest - trainData_scaled$Ranking)^2)

train_error_randomForest

## [1] 0.180433
```

### Berechnung Gerneralization Error

```

pred_randomForest = predict(m_randomForest, testData_scaled)
test_error_randomForest = mean((pred_randomForest - testData_scaled$Ranking)^2)

test_error_randomForest

## [1] 0.1795474

```

## NNNet

Testen des nnet Modells um den Unterschied nochmals zu verdeutlichen zwischen der Qualität von nnet und keras. (Abgesehen von dem enormen Zeitunterschied)

**Wurde komplett auskommentiert, da es sehr lange dauert**

```
# m_nnet = nnet(Ranking ~ ., data = trainData_scaled, size = 10, lineout = TRUE, decay = 0.2, MaxNWts =
```

Getestet zwischen: Size: 10 - 500 decay: 0.05 - 0.3

Beste Parameter beim Erstelldruckgang: size: 90 | decay: 0.1

```

# tune_nnet = tune.nnet(
#   Ranking ~ .,
#   data = trainData_scaled,
#   MaxNWts = 30000,
#   lineout = TRUE,
#   size = seq(10, 500, by = 80),
#   decay = seq(0.05, 0.3, by = 0.05),
#   tunecontrol = tune.control(sampling = "fix",
#                             fix = 2 / 3)
# )
#
# tune_nnet
# plot(tune_nnet, color.palette = topo.colors)

```

## Erstellung des getunten Modells

*Auskommentiert, da es recht lange braucht*

```
# m_nnet = nnet(Ranking ~ ., data = trainData_scaled, size = 90, lineout = TRUE, decay = 0.1, MaxNWts =
```

## Berechnung Training Error

```

# fitted_nnet = predict(m_nnet, trainData_scaled)
# train_error_nnet = mean((fitted_nnet - trainData_scaled$Ranking)^2)
#
# train_error_nnet

```

## Berechnung Gerneralization Error

```
# pred_nnet = predict(m_nnet, testData_scaled)
# test_error_nnet = mean((pred_nnet - testData_scaled$Ranking)^2)
#
# test_error_nnet
```

## Neural Network (keras -> tensorflow)

### Binary Coding the factors

Eventuell ist es nicht so besonders gut so etwas wie die Stadt hier einzugeben, da diese unglaublich viele Ausprägungen hat. Generell sind drei Faktoren recht viel.

```
trainData_bcoded_City = to_categorical(as.integer(trainData_scaled$City) - 1)
trainData_bcoded_Rating = to_categorical(as.integer(trainData_scaled$Rating) - 1)
trainData_bcoded_Price_Range = to_categorical(as.integer(trainData_scaled$Price_Range) - 1)

trainData_bound = cbind(
  trainData_bcoded_City,
  trainData_bcoded_Rating,
  trainData_bcoded_Price_Range,
  trainData_scaled$Review_Count
)
trainData_labels = trainData_scaled$Ranking

# str(trainData_scaled)
# 31 + 9 + 3 + 1 = 44 --> input Shape

testData_bcoded_City = to_categorical(as.integer(testData_scaled$City) - 1)
testData_bcoded_Rating = to_categorical(as.integer(testData_scaled$Rating) - 1)
testData_bcoded_Price_Range = to_categorical(as.integer(testData_scaled$Price_Range) - 1)

testData_bound = cbind(
  testData_bcoded_City,
  testData_bcoded_Rating,
  testData_bcoded_Price_Range,
  testData_scaled$Review_Count
)
testData_labels = testData_scaled$Ranking
```

### Basic model

```
library(keras) # Weniger oft eine Fehlermeldung wenn man sie nochmal aufruft
# m_nn_keras <- keras_model_sequential()
# m_nn_keras %>%
#   layer_dense(units = 100, activation = "sigmoid", input_shape = c(44)) %>%
#   layer_dense(units = 1, activation = "sigmoid")
```

## Compile the model

```
# m_nn_keras %>% compile(  
#   loss = "mse",  
#   optimizer = optimizer_rmsprop(),  
#   metrics = c("mean_absolute_error")  
# )  
# summary(m_nn_keras)
```

## Fit train data

```
# history_nn_keras <- m_nn_keras %>% fit(  
#   trainData_bound, trainData_labels,  
#   epochs = 10, batch_size = 100,  
#   validation_split = 0.2  
# )  
# plot(history_nn_keras)
```

## Parameter Tuning

```
# m_nn_keras <- keras_model_sequential()  
# m_nn_keras %>%  
#   # layer_dense(units = 100, activation = "relu", input_shape = c(44)) %>%  
#   # layer_dense(units = 1, activation = "relu")  
#   # layer_dense(units = 100, activation = "softplus", input_shape = c(44)) %>%  
#   # layer_dense(units = 1, activation = "softplus")  
#   # layer_dense(units = 100, activation = "exponential", input_shape = c(44)) %>%  
#   # layer_dense(units = 1, activation = "exponential")  
#   # layer_dense(units = 100, activation = "softsign", input_shape = c(44)) %>%  
#   # layer_dense(units = 1, activation = "softsign")  
#   layer_dense(units = 100, activation = "selu", input_shape = c(44)) %>%  
#   layer_dense(units = 1, activation = "selu")  
#   # layer_dense(units = 100, activation = "elu", input_shape = c(44)) %>%  
#   # layer_dense(units = 1, activation = "elu")  
#   # layer_dense(units = 100, activation = "exponential", input_shape = c(44)) %>%  
#   # layer_dense(units = 1, activation = "exponential")  
#  
# m_nn_keras %>% compile(  
#   loss = "mse",  
#   optimizer = optimizer_rmsprop(),  
#   metrics = c("mean_absolute_error")  
# )  
#  
# history_nn_keras <- m_nn_keras %>% fit(  
#   trainData_bound, trainData_labels,  
#   epochs = 10, batch_size = 100,  
#   validation_split = 0.2  
# )  
# plot(history_nn_keras)
```

**Testen von anderen activation functions** Elu oder Selu sollten als Activation Parameter weitverwendet und optimiert werden

**Testen von Input Units** Input Units getestet zwischen: 10 - 10000

Ergebnis: Bereich unter 300 sollte weiter getestet werden

Erkenntnis: Selu braucht bei uns niedrigere Start Units (Unter 200), damit es nicht auswendig lernt.

```
# m_nn_keras <- keras_model_sequential()
# m_nn_keras %>%
#   layer_dense(units = 10000, activation = "selu", input_shape = c(44)) %>%
#   # layer_dense(units = 1000, activation = "selu", input_shape = c(44)) %>%
#   # layer_dense(units = 500, activation = "selu", input_shape = c(44)) %>%
#   # layer_dense(units = 300, activation = "selu", input_shape = c(44)) %>%
#   # layer_dense(units = 200, activation = "selu", input_shape = c(44)) %>%
#   # layer_dense(units = 100, activation = "selu", input_shape = c(44)) %>%
#   # layer_dense(units = 50, activation = "selu", input_shape = c(44)) %>%
#   # layer_dense(units = 1, activation = "selu")
#
# m_nn_keras %>% compile(
#   loss = "mse",
#   optimizer = optimizer_rmsprop(),
#   metrics = c("mean_absolute_error")
# )
# summary(m_nn_keras)
#
# history_nn_keras <- m_nn_keras %>% fit(
#   trainData_bound, trainData_labels,
#   epochs = 10, batch_size = 100,
#   validation_split = 0.2
# )
# plot(history_nn_keras)
```

**Testen von batch-size** Ergebnis: Batch-size sollte sehr niedrig sein (Dafür braucht es auch länger): Unter 20 am Besten

```
# m_nn_keras <- keras_model_sequential()
# m_nn_keras %>%
#   layer_dense(units = 100, activation = "elu", input_shape = c(44)) %>%
#   layer_dense(units = 1, activation = "elu")
#
# m_nn_keras %>% compile(
#   loss = "mse",
#   optimizer = optimizer_rmsprop(),
#   metrics = c("mean_absolute_error")
# )
# summary(m_nn_keras)
#
# history_nn_keras <- m_nn_keras %>% fit(
#   trainData_bound, trainData_labels,
#   # epochs = 10, batch_size = 10,
#   # epochs = 10, batch_size = 50,
```

```

#   epochs = 10, batch_size = 100,
#   # epochs = 10, batch_size = 1000,
#   validation_split = 0.2
# )
# plot(history_nn_keras)

```

**Test: Deep Learning** Erkanntes Merkmal: Auch nach über 100 Epochen entsteht kein Overfitting. Allerdings gibt es meistens ab der 30 Epoche keine Verbesserung mehr.

```

# m_nn_keras <- keras_model_sequential()
# m_nn_keras %>%
#   # Versuch 1 - Ist auch nach über 50 Epochen nicht unter 15% gegangen -> nicht so gut
#   # layer_dense(units = 100, activation = "elu", input_shape = c(44)) %>%
#   # layer_dropout(rate = 0.3) %>%
#   # layer_dense(units = 10, activation = "elu") %>%
#   # layer_dropout(rate = 0.2) %>%
#   # layer_dense(units = 1, activation = "elu")
#
#   # # Versuch 2 - Lernt schon recht früh auswendig -> 13%
#   # layer_dense(units = 1000, activation = "elu", input_shape = c(44)) %>%
#   # layer_dropout(rate = 0.3) %>%
#   # layer_dense(units = 100, activation = "elu") %>%
#   # layer_dropout(rate = 0.2) %>%
#   # layer_dense(units = 1, activation = "elu")
#
#   # # Versuch 3 - Nach 10 Epochen schon relativ am Minimum -> 11,8%
#   # # (Pendelt sich nach 30 Epochen bei 11,3% ein)
#   # layer_dense(units = 100, activation = "elu", input_shape = c(44)) %>%
#   # layer_dropout(rate = 0.3) %>%
#   # layer_dense(units = 50, activation = "elu") %>%
#   # layer_dropout(rate = 0.2) %>%
#   # layer_dense(units = 1, activation = "elu")
#
#   # # Versuch 4 - Pendelt sich um die 12% ein
#   # layer_dense(units = 100, activation = "elu", input_shape = c(44)) %>%
#   # layer_dropout(rate = 0.3) %>%
#   # layer_dense(units = 30, activation = "elu") %>%
#   # layer_dropout(rate = 0.2) %>%
#   # layer_dense(units = 1, activation = "elu")
#
#   # # Versuch 5 -Pendelt sich um die 13% ein
#   # layer_dense(units = 50, activation = "elu", input_shape = c(44)) %>%
#   # layer_dropout(rate = 0.3) %>%
#   # layer_dense(units = 20, activation = "elu") %>%
#   # layer_dropout(rate = 0.2) %>%
#   # layer_dense(units = 1, activation = "elu")
#
#   # # Versuch 6 - Mehrere hidden Layer machen es nicht besser -> 14%
#   layer_dense(units = 100, activation = "elu", input_shape = c(44)) %>%
#   layer_dropout(rate = 0.3) %>%
#   layer_dense(units = 50, activation = "elu") %>%
#   layer_dropout(rate = 0.2) %>%
#   layer_dense(units = 10, activation = "elu") %>%

```

```

#   layer_dropout(rate = 0.1) %>%
#   layer_dense(units = 1, activation = "elu")
#
# m_nn_keras %>% compile(
#   loss = "mse",
#   optimizer = optimizer_rmsprop(),
#   metrics = c("mean_absolute_error")
# )
# summary(m_nn_keras)
#
# history_nn_keras <- m_nn_keras %>% fit(
#   trainData_bound, trainData_labels,
#   epochs = 10, batch_size = 10,
#   # epochs = 30, batch_size = 10,
#   validation_split = 0.2
# )
# plot(history_nn_keras)

```

## Erstellung des getunten Modells

Es gibt mehrere akzeptable Modelle. Eine davon ist Elu mit 20 Input Units und 7 Epochen (Es Overfitted meist ab 7-10 Epochen im Schnitt) -> Kommt auf in etwa 11,4% Error

Trotzdem haben wir uns nach vielen Durchläufen dazu entschieden, ein Modell mit mehreren hidden layers zu nehmen, da diese mit über zehn verschiedenen Zuteilungen auf Trainings- und Testdaten, konsistent waren. Die mit nur einem hidden layer hatten in 3/10 starke Inkonsistenz und haben zum Beispiel schon in der zweiten Epoche overfitted, was wir vermeiden wollten. Seeden wäre auch ein Option gewesen, hätte aber eher zum selektiven Datenaussuchen geführt, was wir vermeiden wollten damit ein gutes Generalisationsmodell entsteht.

Anmerkung: Beim Testen von den Varianten mit nur einem hidden layer kommt es hin und wieder vor, dass Unglückliche Werte oder Gewichte anfänglich genommen werden und dieses Ergebnis dadurch anders aussieht (zb Früheres Overfitting oder andere weirde Ergebnisse)

```

m_nn_keras <- keras_model_sequential()
m_nn_keras %>%
  layer_dense(units = 100, activation = "elu", input_shape = c(44)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 50, activation = "elu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 1, activation = "elu")

m_nn_keras %>% compile(
  loss = "mse",
  optimizer = optimizer_rmsprop(),
  metrics = c("mean_absolute_error")
)
summary(m_nn_keras)

## Model: "sequential"
##
## Layer (type)                  Output Shape           Param #
## =====
## 
```

```

## dense (Dense)           (None, 100)          4500
##
## dropout (Dropout)       (None, 100)          0
##
## dense_1 (Dense)         (None, 50)           5050
##
## dropout_1 (Dropout)     (None, 50)           0
##
## dense_2 (Dense)         (None, 1)            51
## =====

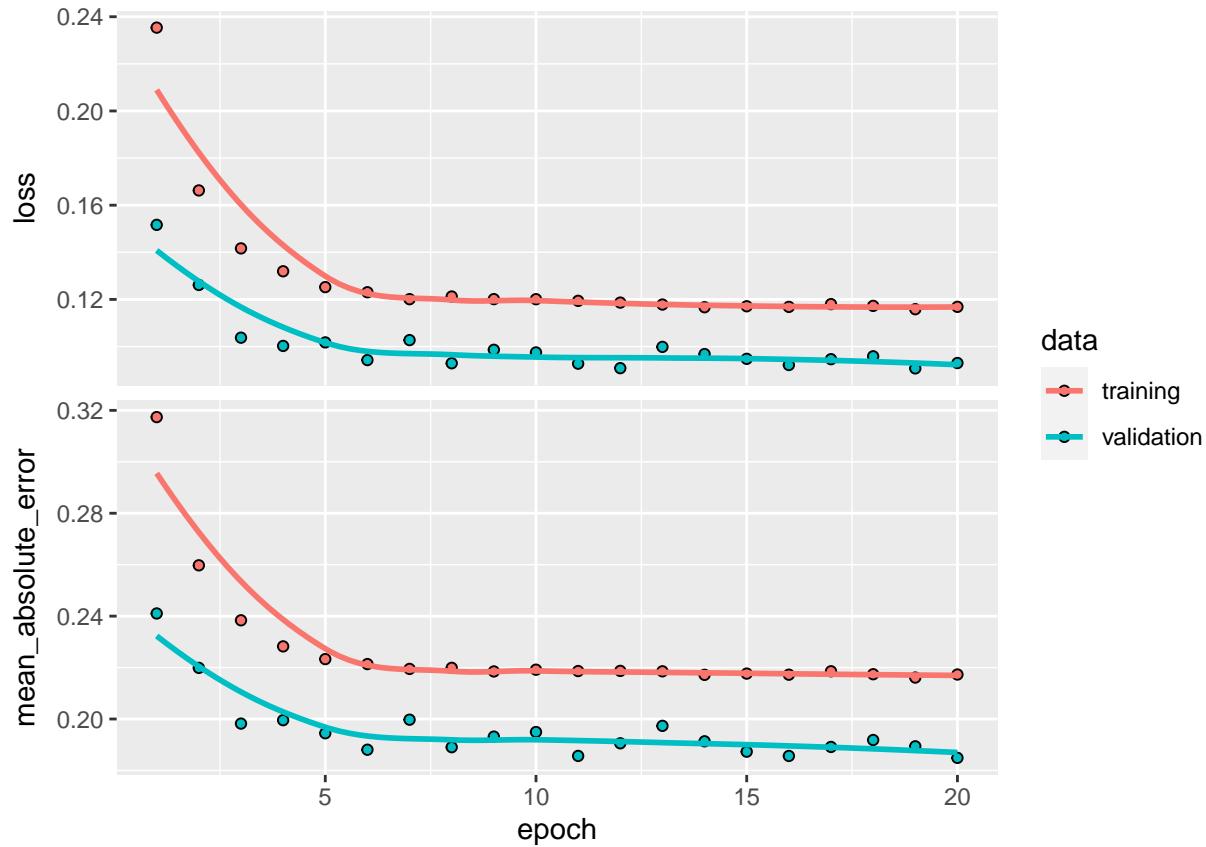
## Total params: 9,601
## Trainable params: 9,601
## Non-trainable params: 0
## -----


save_model_tf(m_nn_keras, "model/") # speicherung des Modells für REST

history_nn_keras <- m_nn_keras %>% fit(
  trainData_bound, trainData_labels,
  # 20, da es sonst sehr lange dauert und es nicht wirklich besser wird
  # Es wurde bis 100 epochen getestet und die Unterschiede waren marginal.
  epochs = 20, batch_size = 10,
  # epochs = 100, batch_size = 10,
  validation_split = 0.2
)
plot(history_nn_keras)

## `geom_smooth()` using formula 'y ~ x'

```



### Berechnung Training Error

```
fitted_nn_keras = predict(m_nn_keras, trainData_bound)
train_error_nn_keras = mean((fitted_nn_keras - trainData_labels)^2)

train_error_nn_keras

## [1] 0.09885997
```

### Berechnung Gerneralization Error

```
pred_nn_keras = predict(m_nn_keras, testData_bound)
test_error_nn_keras = mean((pred_nn_keras - testData_labels)^2)

test_error_nn_keras

## [1] 0.09467244
```

### Endgültige Entscheidung

Für unseren Fall der Reggression würden wir uns für das getunte Model von keras/tensorflow entscheiden, da dieses den geringsten Error aufweist. Baummodelle waren ebenfalls akzeptabel mit um die 20% Error,

allerdings war das Neuronale Netz um zirka 8-10% besser.

## REST Service Vorbeireitung

Für den REST Server werden einige Variablen benötigt. Diese werden hier exportiert.

```
levels_city = levels(trainData_scaled$City)
levels_price_range = levels(trainData_scaled$Price_Range)
levels_rating = levels(trainData_scaled$Rating)

save(levels_city, levels_price_range, levels_rating, pp, unPreProc, file="variables.rda")
```

## Text mining

```
library(tm)

## Warning: package 'tm' was built under R version 4.0.3

## Loading required package: NLP

## Warning: package 'NLP' was built under R version 4.0.3

##
## Attaching package: 'NLP'

## The following object is masked from 'package:ggplot2':
## 
##     annotate

library(wordcloud)

## Warning: package 'wordcloud' was built under R version 4.0.3

## Loading required package: RColorBrewer

library(tidytext) # für unnest_tokens

## Warning: package 'tidytext' was built under R version 4.0.3
```

Zuerst werden die Review Texte extrahiert und in eine neue Spalte eingefügt.

```
restaurant_tm <- restaurant_data

# Review Text ausschneiden und in neue Spalte einfügen (Daten entfernen)
restaurant_tm$ReviewText = gsub('.{31}$', '', restaurant_tm$Reviews)
# Sonderzeichen entfernen
restaurant_tm$ReviewText = gsub("[[:punct:]]+", "", restaurant_tm$ReviewText)
```

## Sentiment analysis

Wir analyseren den Text um die Stimmung/Empfinden der Person zu erkennen, waren ihre Bewertungen positiv oder negativ. Dafür schauen wir uns 6 Städte an: London, Paris, Berlin, Madrid, Rom und Wien.

```
London = restaurant_tm %>% filter(City=="London") %>% select(ReviewText)
Paris = restaurant_tm %>% filter(City=="Paris") %>% select(ReviewText)
Berlin = restaurant_tm %>% filter(City=="Berlin") %>% select(ReviewText)
Madrid = restaurant_tm %>% filter(City=="Madrid") %>% select(ReviewText)
Rome = restaurant_tm %>% filter(City=="Rome") %>% select(ReviewText)
Vienna = restaurant_tm %>% filter(City=="Vienna") %>% select(ReviewText)
```

Um leichter Tokens zu erstellen und um die Tokens in Positiv und Negativ einzufügen, haben wir zwei Hilfsfunktionen geschrieben.

```
 tokenize = function (review){
  # Alle Zeilen zu einem Text zusammenfügen
  words = paste(unlist(review), collapse=" ")
  # Aus dem Text wird ein tibble
  # und dieses wird dann in Tokens bzw Keywords umgewandelt
  tokens = tibble(text=words) %>%
    unnest_tokens(word,text)
  # Ergebnis ein tibble mit einer Spalte 'word' das alle Tokens beinhaltet
  return(tokens)
}

restaurant_sentiments = function(tokens){
  sentiments= tokens %>%
    # Das Empfinden, ob der Tok / das Wort positiv oder negativ ist,
    # wird mit einem inner join hinzugefügt.
    # So bleiben nur die relevanten Token über
    inner_join(get_sentiments("bing")) %>%
    # positive und negative werden zusammengezählt
    count(sentiment) %>%
    # Tabelle spiegeln, neue Spalten: negative, positive
    pivot_wider(names_from = sentiment, values_from = n, values_fill = 0) %>%
    # Spalten deutsch
    rename("Negativ" = negative) %>%
    rename("Positiv" = positive) %>%
    # zusätzliche Spalte 'Unterschied'
    mutate(Unterschied = Positiv - Negativ)
  return(sentiments)
}
```

Nun werden die Tokens der Reviews je Stadt erzeugt und dann analysiert nach dem Empfinden und zusammengezählt.

```
# Token generierung je Stadt
londonTokens = tokenize(London$ReviewText)
parisTokens = tokenize(Paris$ReviewText)
berlinTokens = tokenize(Berlin$ReviewText)
madridTokens = tokenize(Madrid$ReviewText)
romeTokens = tokenize(Rome$ReviewText)
```

```

viennaTokens = tokenize(Vienna$ReviewText)

# Tokens nach Empfindung zählen
londonSentiment = restaurant_sentiments(londonTokens)

## Joining, by = "word"

parisSentiment = restaurant_sentiments(parisTokens)

## Joining, by = "word"

berlinSentiment = restaurant_sentiments(berlinTokens)

## Joining, by = "word"

madridSentiment = restaurant_sentiments(madridTokens)

## Joining, by = "word"

romeSentiment = restaurant_sentiments(romeTokens)

## Joining, by = "word"

viennaSentiment = restaurant_sentiments(viennaTokens)

## Joining, by = "word"

sentiments= rbind(londonSentiment,parisSentiment, berlineSentiment,
                   madridSentiment, romeSentiment, viennaSentiment)
cities=data.frame(Name=c("London", "Paris", "Berlin", "Madrid", "Rom", "Wien"))

# Städte mit den Empfindungen der Bewertung zusammenführen
city_sentiments=cbind(cities, sentiments)

```

Eine grafische Darstellung der Empfindungen der Reviews je Stadt.

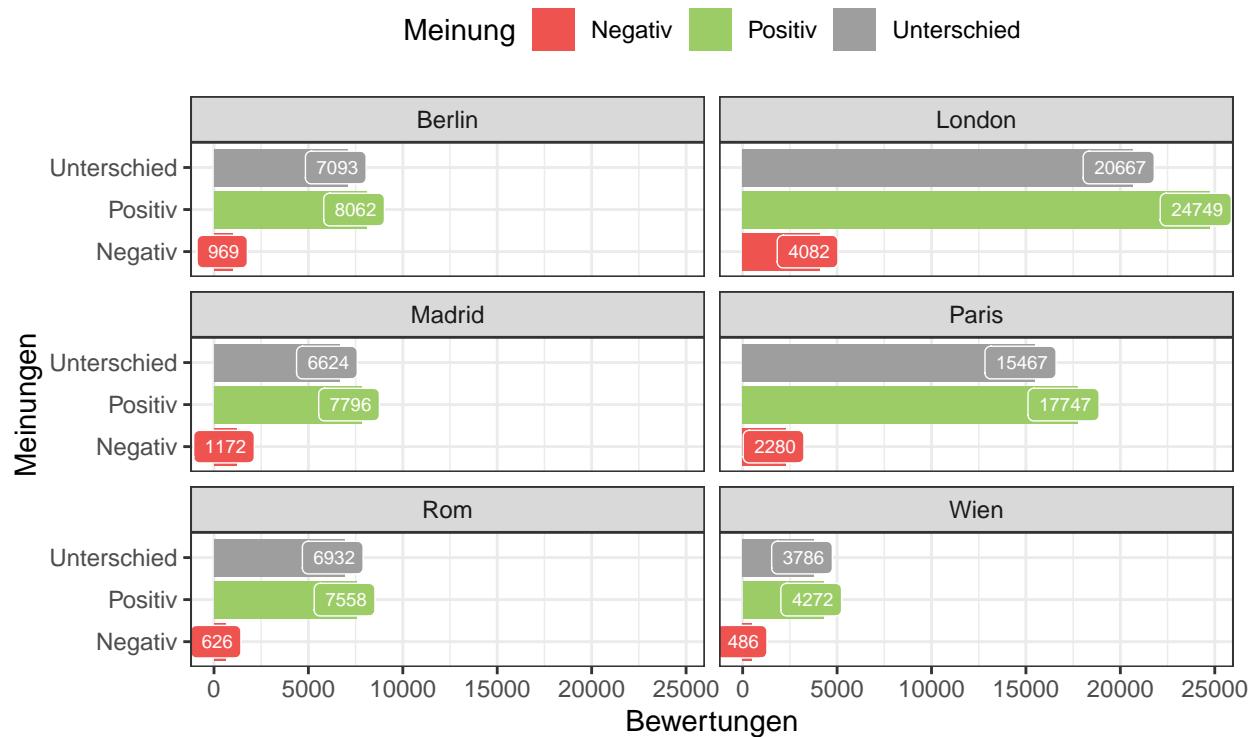
```

sents = city_sentiments %>%
  pivot_longer(-Name, names_to="Meinung", values_to="Value") %>%
  arrange(Name)

ggplot(sents, aes(x = Meinung, y = Value, fill=Meinung)) +
  geom_bar(stat = "identity", position = "dodge") +
  labs(title="Positive und negative Bewertungen der Reviewer",
       subtitle="Und deren Differenz",
       x="Meinungen",
       y="Bewertungen")+
  facet_wrap(~Name, ncol=2) +
  theme_bw() +
  scale_fill_manual(values=c("#ef5350","#9ccc65","#9e9e9e","#9e9e9e"))+
  geom_label(aes(label=Value), hjust=0.7, color="white", size=2.5, show.legend=F)+
  theme(legend.position="top")+
  coord_flip()

```

## Positive und negative Bewertungen der Reviewer Und deren Differenz



Wie man erkennen kann, haben alle Städte eine positive Differenz, was darauf schließen lässt, dass alle hauptsächlich positiv Bewertet wurden. London hat den größten Unterschied von positiven zu negativen Bewertungen, gefolgt von Paris. Wien hat die kleinste Differenz, was bedeutet, dass die Anzahl an positiven und negativen Bewertungen sehr ähnlich ist.

## Corpus erstellen und Document Term Matrix

### Preprocessing

Aus den einzelnen Reviewtexten wird ein Corpus erstellt und dieser für die Analyse vorbereitet. Dabei werden numbers, capitalization, common words, punctuation bearbeitet bzw. entfernt.

```
corpus=VCorpus(VectorSource(restaurant_tm$ReviewText))

corpus=tm_map(corpus, tolower)
corpus=tm_map(corpus, removePunctuation)
corpus=tm_map(corpus, removeWords, stopwords("english"))
corpus=tm_map(corpus, stripWhitespace)
corpus=tm_map(corpus, PlainTextDocument)
```

### Document Term Matrix erstellen

```

dtm=DocumentTermMatrix(corpus)
dtm

## <<DocumentTermMatrix (documents: 125238, terms: 26404)>>
## Non-/sparse entries: 495000/3306289152
## Sparsity           : 100%
## Maximal term length: 31
## Weighting          : term frequency (tf)

length(findFreqTerms(dtm, lowfreq=400))

## [1] 189

3306289152/100

## [1] 33062892

```

Man kann erkennen, dass viele Wörter nur einmal vorkommen: 26404. Die Sparsity ist 100%, genauer 99.985%. Daraus schließen wir, sehr viele Spalten in der Document Term Matrix sind leer(null). Ein weiterer interessanter Fakt, nur 189 Wörter kommen mindestens 400 mal oder öfter vor. Darum werden wir die weniger relevanten Wörter im nächsten Schritt entfernen.

```

dtms = removeSparseTerms(dtm, 0.998)
dtms

## <<DocumentTermMatrix (documents: 125238, terms: 267)>>
## Non-/sparse entries: 339689/33098857
## Sparsity           : 99%
## Maximal term length: 13
## Weighting          : term frequency (tf)

most_words = as.data.frame(as.matrix(dtms))

```

Die Sparsity ist zwar noch sehr hoch, obwohl es schon 1/100 von den ursprünglichen sparse Wörtern ist. Nun verbleiben noch 267 Wörter. Reichlich für eine Visualisierung.

```

freq <- sort(colSums(most_words), decreasing=TRUE)
head(freq, 15)

##      food      good     great      nice     place    service     best
##    29315    26442    24128    12828    11499     9636    7769
## restaurant  excellent   lunch delicious friendly amazing  lovely
##      7662     7211     5859     4796     4149     3981    3980
##      pizza
##      3929

```

Bei Betrachtung der 15 häufigsten Wörter erkennt man viele positive Wörter und Pizza dürfte wohl sehr beliebt sein. Wer mag den keine Pizza?

# Wordcloud

Eine Wordcloud ist ein gutes Diagramm, da es für viele sehr einfach zu verstehen ist und schon bunt ist.

```
set.seed(100)
wordcloud(names(freq),
          freq,
          max.words=150,
          random.order = FALSE,
          rot.per=0.3,
          scale=c(5.5, .5),
          colors = brewer.pal(8, "Dark2"))
```



## Die meist verwendeten Wörter in den Reviews

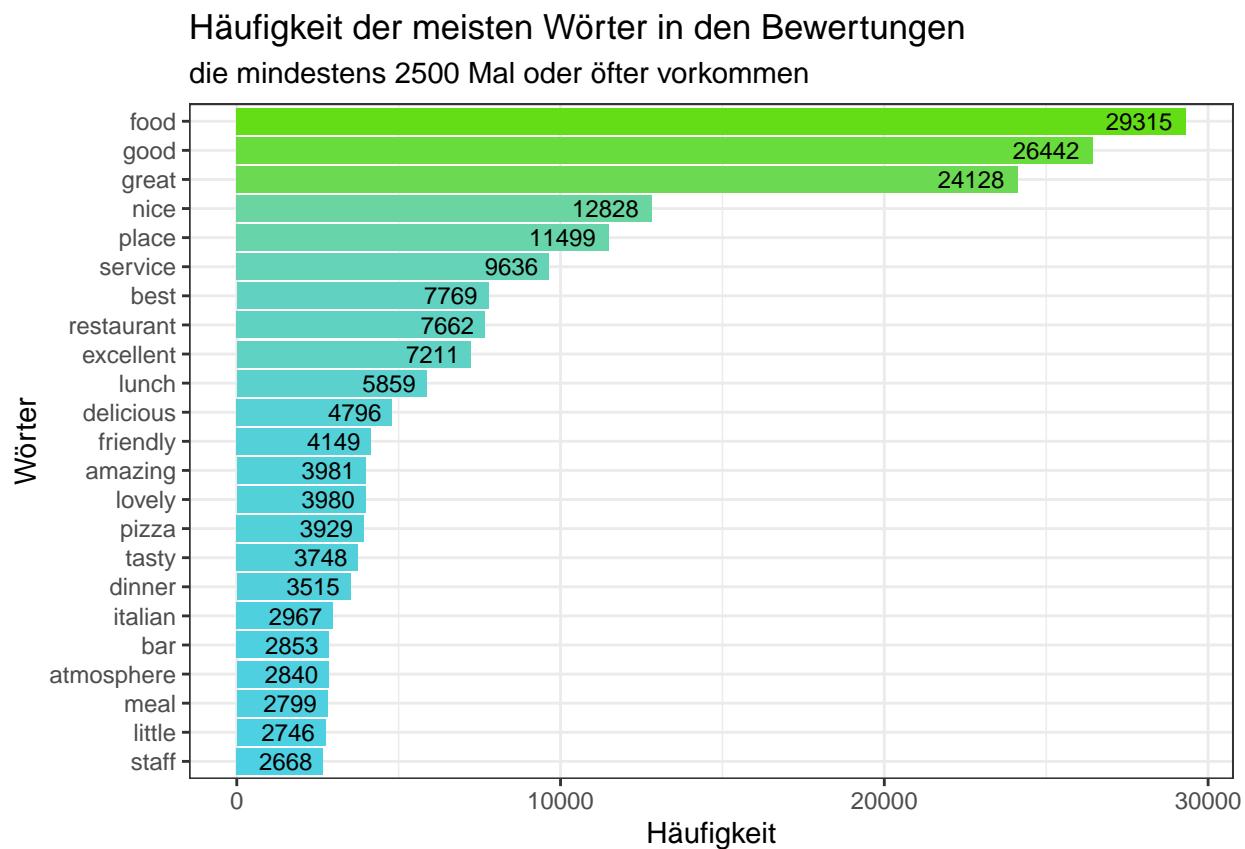
```
words=colnames(most_words)
frequency=colSums(most_words)
top_words=data.frame(words, frequency)

ggplot(top_words [top_words$frequency>=2500,],
       aes(x= reorder(words,frequency),
            y=frequency, fill=frequency)) +
  geom_bar(stat="identity")+
  coord_flip()
```

```

theme_bw() +
theme(legend.position="none")+
scale_fill_continuous(low="#4dd0e1", high="#64dd17")+
labs(title="Häufigkeit der meisten Wörter in den Bewertungen",
    subtitle="die mindestens 2500 Mal oder öfter vorkommen",
    x= "Wörter",
    y="Häufigkeit") +
geom_text(aes(label=frequency), hjust=1.2,color="black", size=3)

```



Wie schon oben erwähnt, sind die meist verwendeten Wörter positiv. Neben Pizza ist auch italienisches Essen sehr beliebt.