FPS Multiplayer project report

June 4, 2020

1 FPS Multiplayer project report

Autohor: Gabrielmaria Scozzarro

Date:25/05/2020

1.1 Executive Summary

This project born from a suggestion from my company partner who is a teacher at the Italian VideoGames Accademy as I need a project for my exam during HardvardX Data Science: Capstone.

The data set origin is provided by a game house which developed a well known online multiplayer First Person Shooter. This game has a Battle Royale-style where 100 players are dropped onto an island empty-handed and must explore, scavenge, and eliminate other players until only one is left standing, all while the play zone continues to shrink. The data set is divided into training and testing sets and the final aim of the project is to provide a prediction algorithm able to determine the final winner starting from player stats recorded during previous games.

To better understand the data we will at first proceed with a an exploration and data analysis (EDA). In the second part of the report we will build the prediction algorithm based on what we discovered in the EDA.

As a quality parameter and reference for other approach we will use the Mean Square Error compute on our model.

Data fields DBNOs - Number of enemy players knocked.
assists - Number of enemy players this player damaged that were killed by teammates.
boosts - Number of boost items used.
damageDealt - Total damage dealt. Note: Self inflicted damage is subtracted.
headshotKills - Number of enemy players killed with headshots.
heals - Number of healing items used.
Id - Player's Id
killPlace - Ranking in match of number of enemy players killed.
killPoints - Kills-based external ranking of player. (Think of this as an Elo ranking where only kills matter.) If there is a value other than -1 in rankPoints, then any 0 in killPoints should be treated as a "None".
killStreaks - Max number of enemy players killed in a short amount of time.
kills - Number of enemy players killed.
longestKill - Longest distance between player and player killed at time of death. This may be misleading, as downing a player and driving away may lead to a large longestKill stat.

matchDuration - Duration of match in seconds.
matchId - ID to identify match. There are no matches that are in both the training and testing set.
matchType - String identifying the game mode that the data comes from. The standard modes are “solo”, “duo”, “squad”, “solo-fpp”, “duo-fpp”, and “squad-fpp”; other modes are from events or custom matches.
rankPoints - Elo-like ranking of player. This ranking is inconsistent and is being deprecated in the API’s next version, so use with caution. Value of -1 takes place of “None”.
revives - Number of times this player revived teammates.
rideDistance - Total distance traveled in vehicles measured in meters.
roadKills - Number of kills while in a vehicle.
swimDistance - Total distance traveled by swimming measured in meters.
teamKills - Number of times this player killed a teammate.
vehicleDestroys - Number of vehicles destroyed.
walkDistance - Total distance traveled on foot measured in meters.
weaponsAcquired - Number of weapons picked up.
winPoints - Win-based external ranking of player. (Think of this as an Elo ranking where only winning matters.) If there is a value other than -1 in rankPoints, then any 0 in winPoints should be treated as a “None”.
groupId - ID to identify a group within a match. If the same group of players plays in different matches, they will have a different groupId each time.
numGroups - Number of groups we have data for in the match.
maxPlace - Worst placement we have data for in the match. This may not match with numGroups, as sometimes the data skips over placements.
winPlacePerc - The target of prediction. This is a percentile winning placement. It is calculated off of maxPlace, not numGroups, so it is possible to have missing chunks in a match.

1.2 Methods

1.2.1 EDA

For our analysis we need the following packages:

[ ]:
```
library(tidyverse)
library(data.table)
library(corrplot)
```

And options:

[2]:
```
options(repr.plot.width=18, repr.plot.height=9)
```

Loading the data set:

[3]:
```
train_set<- fread("C:/Users/elekt/OneDrive/Documenti/datascience_capstone/FPS␣
 ↪Machine Learning project/Data input/train_V2.csv")
test_set<- fread("C:/Users/elekt/OneDrive/Documenti/datascience_capstone/FPS␣
 ↪Machine Learning project/Data input/test_V2.csv")
```

Now we can explore the acquired data:

[3]:
```
str(train_set)
```

```
Classes 'data.table' and 'data.frame':  4446966 obs. of  29 variables:
 $ Id             : chr  "7f96b2f878858a" "eef90569b9d03c" "1eaf90ac73de72"
"4616d365dd2853" …
 $ groupId        : chr  "4d4b580de459be" "684d5656442f9e" "6a4a42c3245a74"
"a930a9c79cd721" …
 $ matchId        : chr  "a10357fd1a4a91" "aeb375fc57110c" "110163d8bb94ae"
"f1f1f4ef412d7e" …
 $ assists        : int  0 0 1 0 0 0 0 0 0 0 …
 $ boosts         : int  0 0 0 0 0 0 0 0 0 0 …
 $ damageDealt    : num  0 91.5 68 32.9 100 …
 $ DBNOs          : int  0 0 0 0 0 1 0 0 0 0 …
 $ headshotKills  : int  0 0 0 0 0 1 0 0 0 0 …
 $ heals          : int  0 0 0 0 0 0 0 0 0 0 …
 $ killPlace      : int  60 57 47 75 45 44 96 48 64 74 …
 $ killPoints     : int  1241 0 0 0 0 0 1262 1000 0 0 …
 $ kills          : int  0 0 0 0 1 1 0 0 0 0 …
 $ killStreaks    : int  0 0 0 0 1 1 0 0 0 0 …
 $ longestKill    : num  0 0 0 0 58.5 …
 $ matchDuration  : int  1306 1777 1318 1436 1424 1395 1316 1967 1375 1930 …
 $ matchType      : chr  "squad-fpp" "squad-fpp" "duo" "squad-fpp" …
 $ maxPlace       : int  28 26 50 31 97 28 28 96 28 29 …
 $ numGroups      : int  26 25 47 30 95 28 28 92 27 27 …
 $ rankPoints     : int  -1 1484 1491 1408 1560 1418 -1 -1 1493 1349 …
 $ revives        : int  0 0 0 0 0 0 0 0 0 0 …
 $ rideDistance   : num  0 0.0045 0 0 0 …
 $ roadKills      : int  0 0 0 0 0 0 0 0 0 0 …
 $ swimDistance   : num  0 11 0 0 0 …
 $ teamKills      : int  0 0 0 0 0 0 0 0 0 0 …
 $ vehicleDestroys: int  0 0 0 0 0 0 0 0 0 0 …
 $ walkDistance   : num  244.8 1434 161.8 202.7 49.8 …
 $ weaponsAcquired: int  1 5 2 3 2 1 1 6 4 1 …
 $ winPoints      : int  1466 0 0 0 0 0 1497 1500 0 0 …
 $ winPlacePerc   : num  0.444 0.64 0.775 0.167 0.188 …
 - attr(*, ".internal.selfref")=<externalptr>
```

We can see that the data is composed by our target value **winPlacePerc** and **24** different feature.

We can discover the number of unique player and games:

[5]:
```
length(unique(train_set$Id))
length(unique(train_set$matchId))
```

4446966

47965

Our train_set contains around 48K different games played by around 444K different players. Now we can have a deeper observation on the main event an multiplayer FPS can have:

[6]:
```
summary(train_set$assists)
summary(train_set$damageDealt)
summary(train_set$headshotKills)
summary(train_set$kills)
summary(train_set$matchDuration)
summary(train_set$walkDistance)
summary(train_set$winPlacePerc)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.0000  0.0000  0.2338  0.0000 22.0000
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    0.00   84.24  130.72  186.00 6616.00
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.0000  0.0000  0.2268  0.0000 64.0000
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.0000  0.0000  0.9248  1.0000 72.0000
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      9    1367    1438    1580    1851    2237
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0.0   155.1   685.6  1154.2  1976.0 25780.0
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 0.0000  0.2000  0.4583  0.4728  0.7407  1.0000       1
```

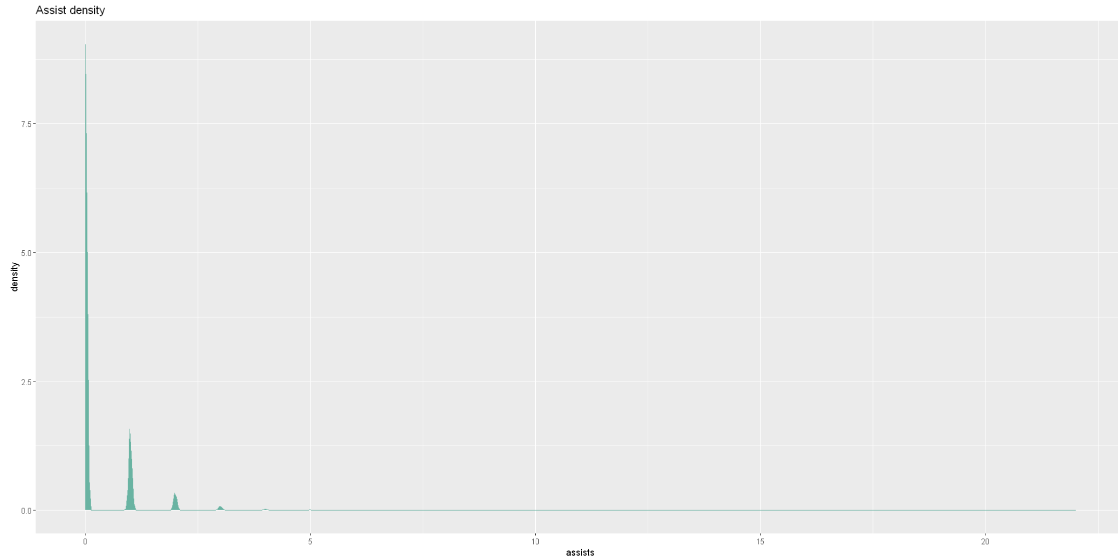
We discover that assist in killing and head shots are quite rare, the avg game duration is around 26 minutes, the avg distance walked is 1.5 Km and the avg player placement is around in the middle of the ranking. We also discover that our target value can be any number from 0 to 1.

We remove NAs from winPlacePerc:

[4]:
```
train_set$winPlacePerc[is.na(train_set$winPlacePerc)]<- 0
```

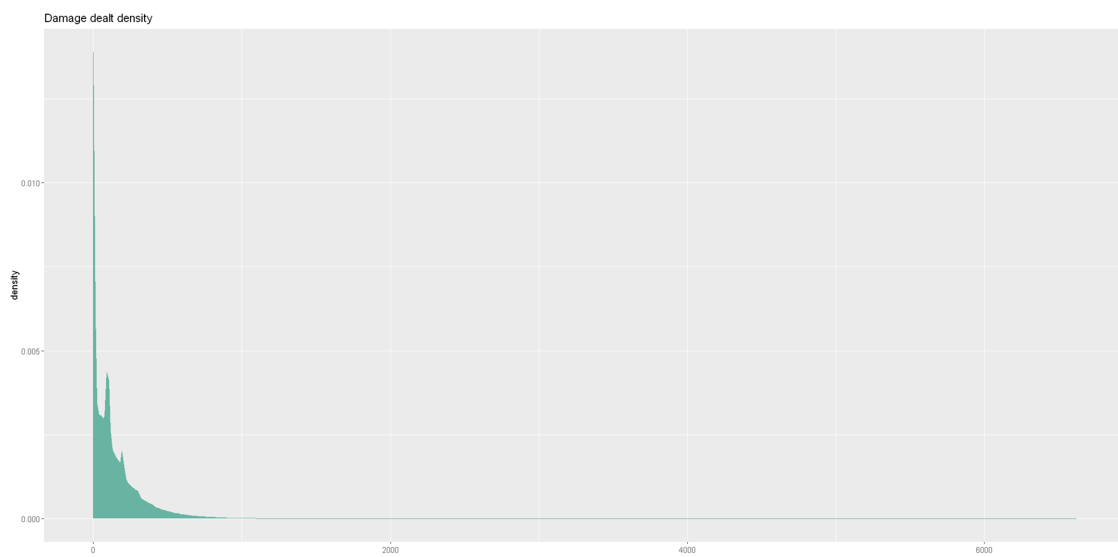
To understand better the main feature in train_set we can visualize their distribution:

[11]:
```
train_set %>%
  ggplot(aes(assists))+
  geom_density(color="#69b3a2", fill="#69b3a2")+
  ggtitle("Assist density")
```



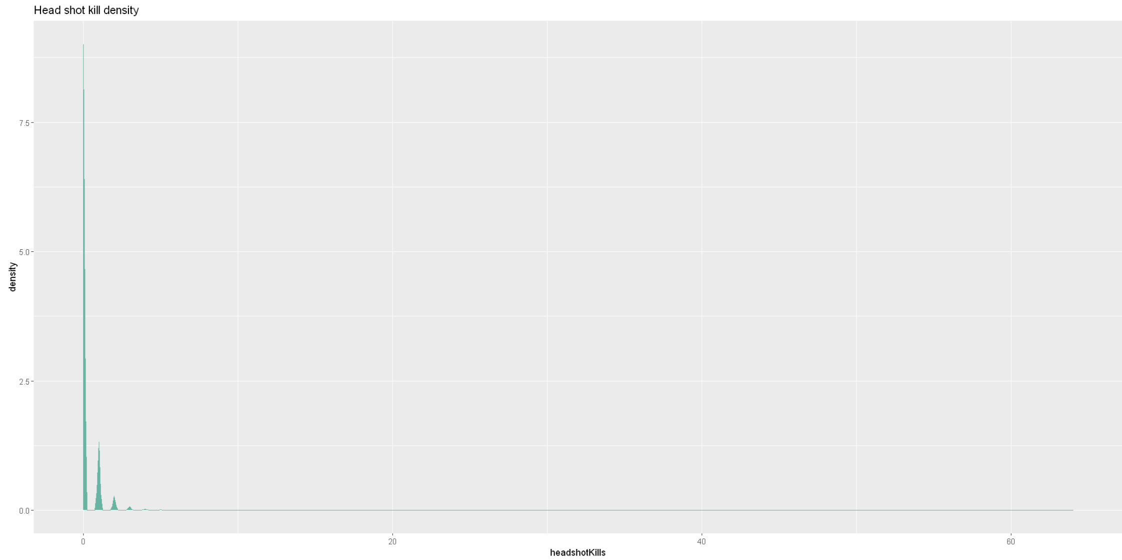
As we aspected assists are very few.

[12]:
```
train_set%>%
  ggplot(aes(damageDealt))+
  geom_density(color="#69b3a2", fill="#69b3a2")+
  ggtitle("Damage dealt density")
```

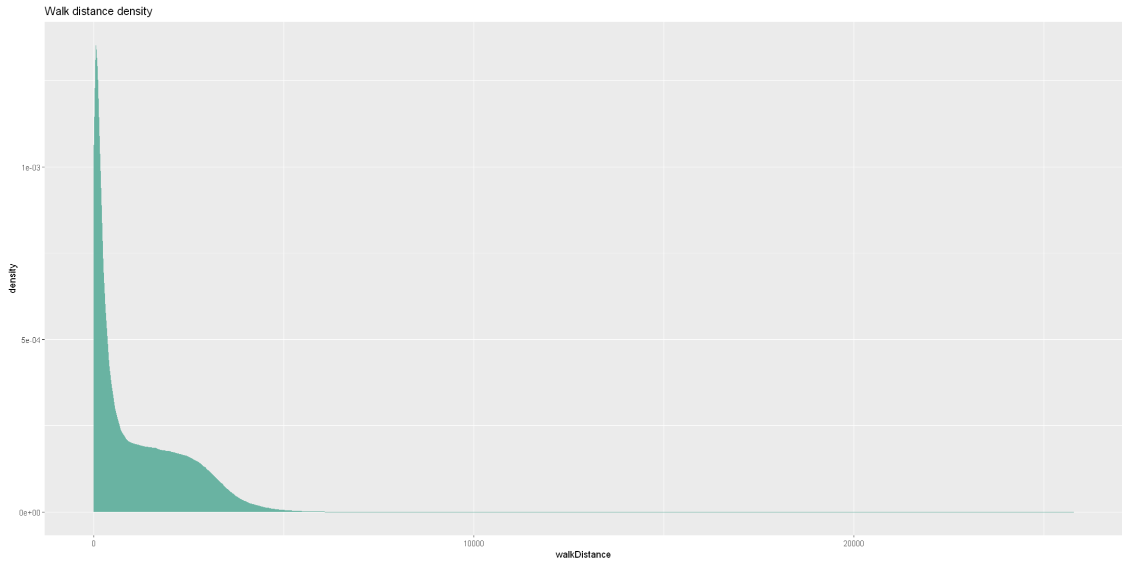


[13]:
```
train_set%>%
  ggplot(aes(headshotKills))+
  geom_density(color="#69b3a2", fill="#69b3a2")+
```

```
  ggtitle("Head shot kill density")
```

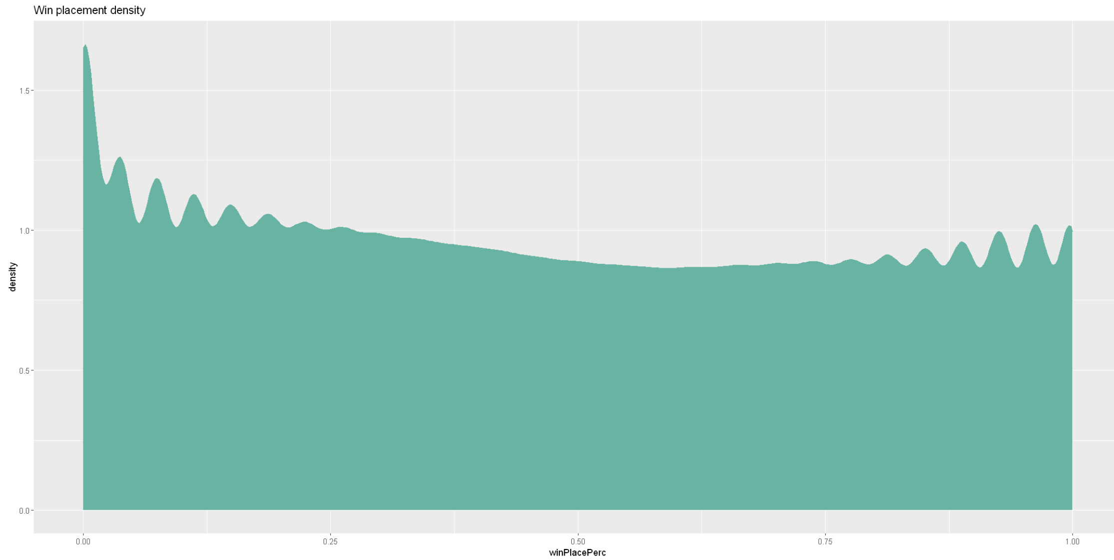


[16]:
```
train_set%>%
  ggplot(aes(walkDistance)) +
  geom_density(color="#69b3a2", fill="#69b3a2") +
  ggtitle("Walk distance density")
```



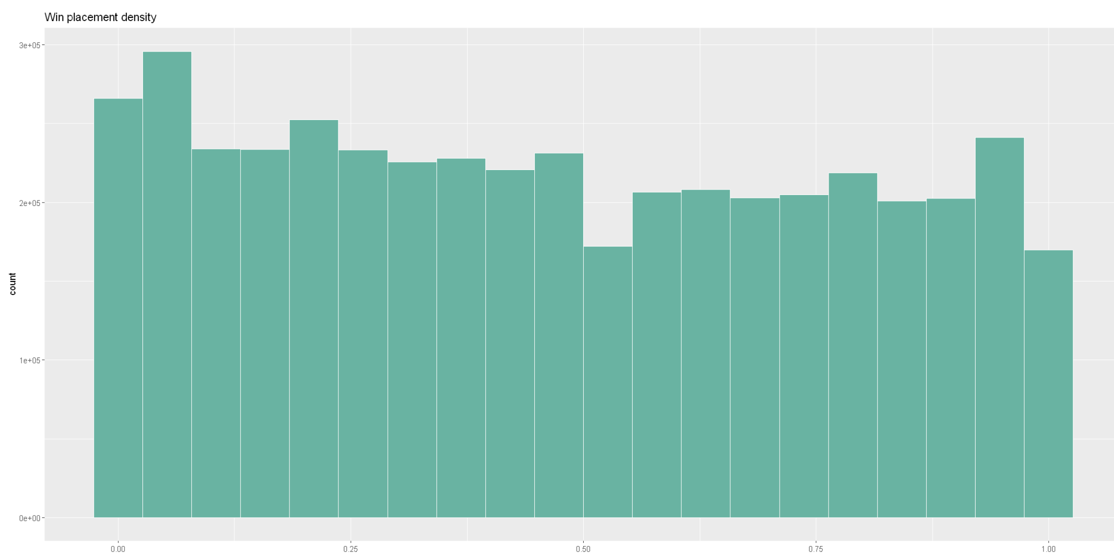
[14]:
```
train_set%>%
  ggplot(aes(winPlacePerc)) +
  geom_density(color="#69b3a2", fill="#69b3a2")+
```

```
  ggtitle("Win placement density")
```



This visualization of the ranking density doesn't allow a good understanding of the distribution. We can remedy with:

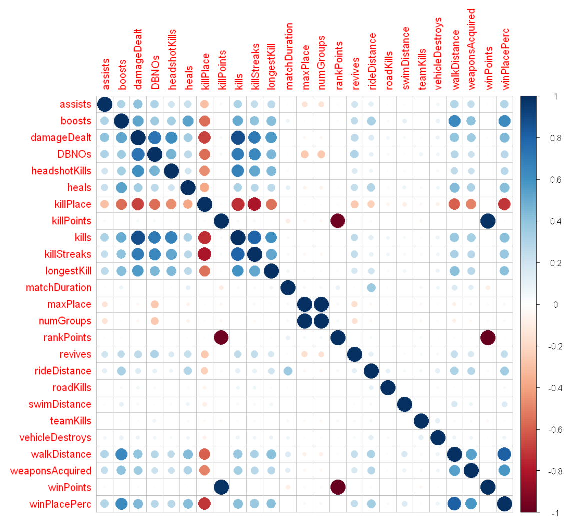
[15]:
```
train_set%>%
  ggplot(aes(winPlacePerc)) +
  geom_histogram(bins = 20, color="white", fill="#69b3a2") +
  ggtitle("Win placement density")
```



The analysis of distribution tell us that the majority of players tend to low performance but are equally distributed in the the ranking.

We can perform a correlation analysis to understand broadly wich are the most important features:

[17]:
```
train_sample<- train_set[,c(4:15,17:29)]
train_corr<- as.data.frame(lapply(train_sample, as.numeric))

corrplot(cor(train_corr), method = "circle")
```

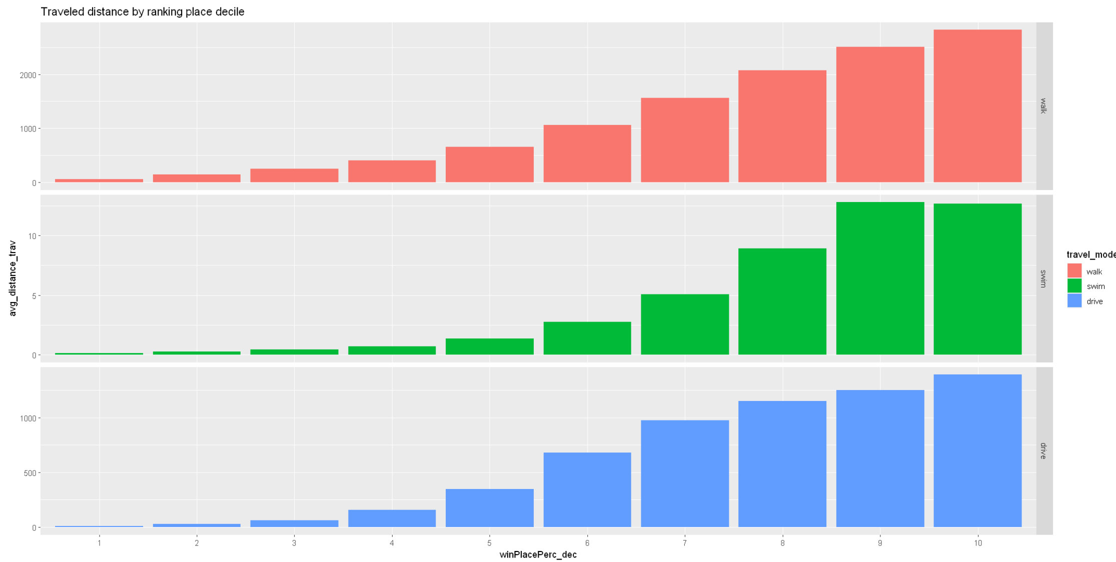


We can appreciate the strong correlation between winPlacePer, our target, and boosts, walkDistance, weaponsAcquired. A medium intensity correlation between winPlacePer and swimDistance, rideDistance, Kills, headshotKills. There is also very strong negative correlation between winPlacePer and klillPlace.

To dig deeper we can perform a match beteween ranking (win place) with travelled distance (walk, swim, drive) using the following code:

[6]:
```
trav<- train_set %>%
  mutate(winPlacePerc_dec = ntile(winPlacePerc, 10)) %>%
  group_by(winPlacePerc_dec) %>%
  summarize(walk = mean(walkDistance),
            swim = mean(swimDistance),
            drive = mean(rideDistance)) %>%
  ungroup() %>%
  reshape2::melt(., measure.vars= c("walk", "swim", "drive"), variable.name =␣
 ↪"travel_mode", value.name = 'avg_distance_trav' ) %>%
        as.data.table()

levels<- trav$winPlacePerc_dec %>%
```

```
  unique() %>%
  sort()

trav$winPlacePerc_dec<- factor(trav$winPlacePerc_dec, levels = levels)

trav %>% ggplot(aes(winPlacePerc_dec, avg_distance_trav, fill = travel_mode)) +
  geom_bar(stat = "identity", position = "dodge") +
  facet_grid(travel_mode ~ . , scales = "free") +
  ggtitle("Traveled distance by ranking place decile")
```

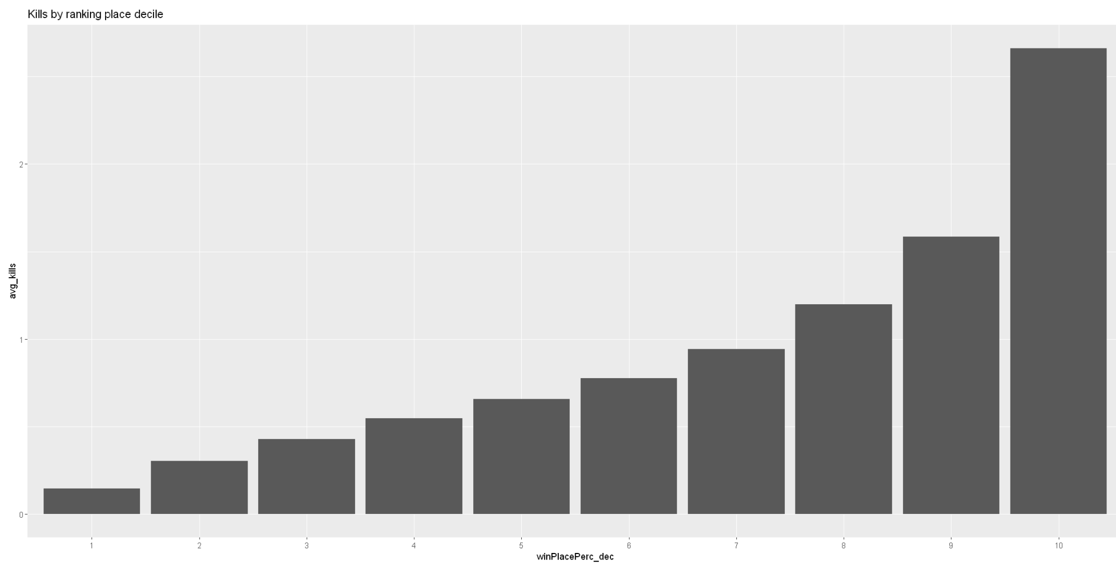


The average distance traveled regardless the type (walk,swim,drive) steadily increase with the player's placement decile. We can deduce that the better player you are, the better is your exploration and knoledge of the map.

We can perform the same type of analysis olso on kills wich in logic should be an important aspect for winning the game:

[7]:
```
kills<- train_set %>%
  mutate(winPlacePerc_dec = ntile(winPlacePerc, 10)) %>%
  group_by(winPlacePerc_dec) %>%
  summarize(kills = mean(kills)) %>%
  ungroup() %>%
  reshape2::melt(., measure.vars= c("kills"), value.name = 'avg_kills' ) %>%
  as.data.table()

levels2<- kills$winPlacePerc_dec %>%
  unique() %>%
  sort()
```

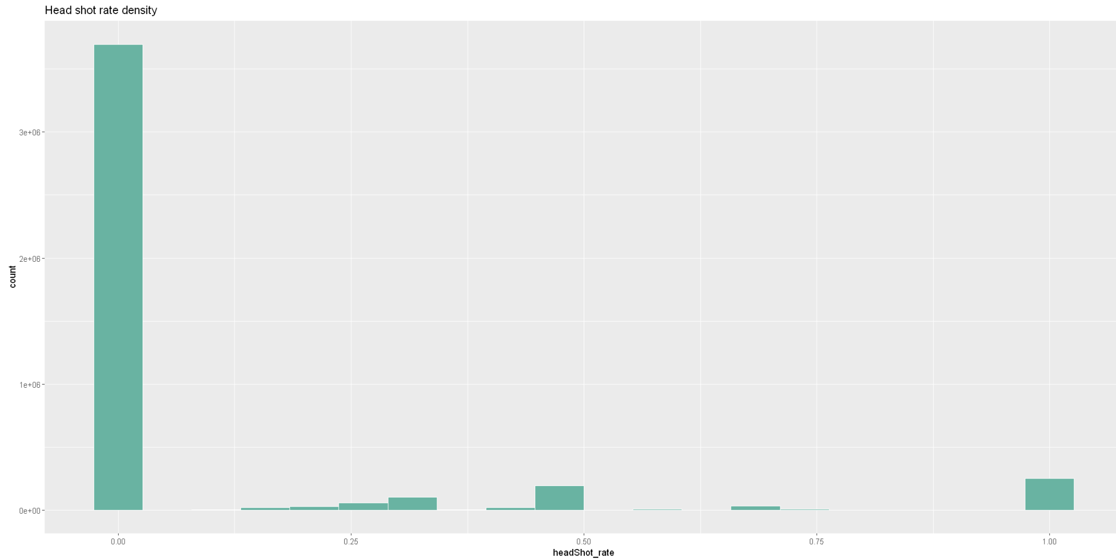
```
kills$winPlacePerc_dec<- factor(kills$winPlacePerc_dec, levels = levels2)

kills %>% ggplot(aes(winPlacePerc_dec, avg_kills)) +
  geom_bar(stat = "identity", position = "dodge") +
  ggtitle("Kills by ranking place decile")
```



We can see the same effect as in the previous plot.

Also head shot kills should be a prove of how good your are in the game so it's worth to perform a deeper analysis on it using the following code:

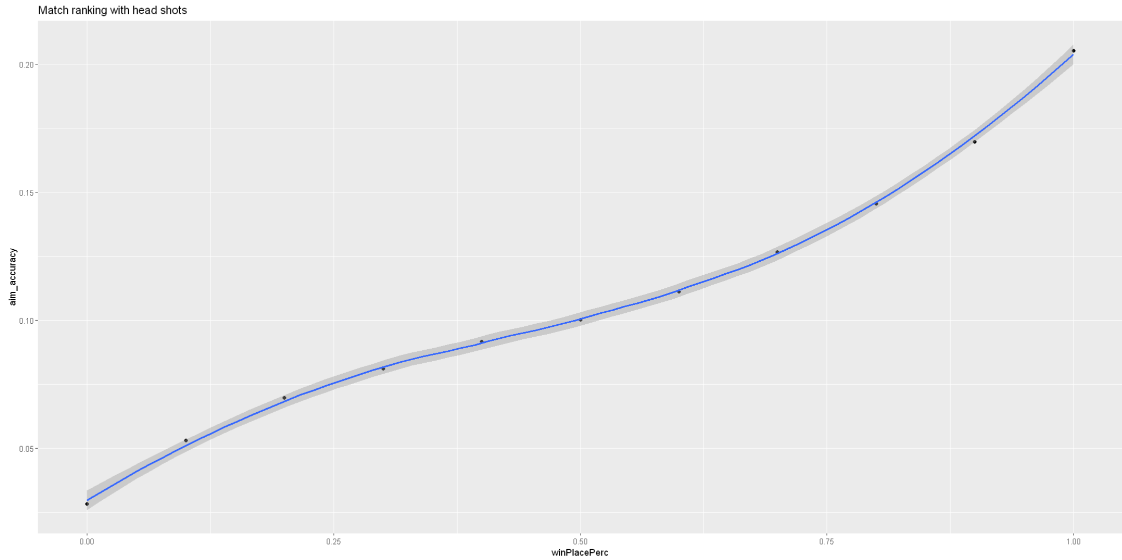
[8]:
```
train_set %>%
  mutate(headShot_rate = ifelse(kills == 0, 0, headshotKills / kills)) %>%
  ggplot(aes(headShot_rate)) +
  geom_histogram(bins = 20, color="white", fill="#69b3a2") +
  ggtitle("Head shot rate density")
```



Head shots are very rare and maybe it has only a marginal effect on the final result of the game. We can perform a match analysis between head shoots and final ranking:

[9]:
```
train_set %>%
  select(Id, headshotKills, kills, winPlacePerc) %>%
  mutate(winPlacePerc = round(winPlacePerc,1)) %>%
  group_by(winPlacePerc) %>%
  summarize(aim_accuracy = mean(ifelse(kills == 0, 0, headshotKills / kills)))
 ↪%>%
  as.data.table() %>%
  ggplot(aes(winPlacePerc, aim_accuracy)) +
  geom_point()+
  geom_smooth(method = "loess") +
  ggtitle("Match ranking with head shots")
```

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



This confirm that there is a link between the ranking and head shoot but is less strong than others.

From my experience on online multiplayer FPS the number of teams partecipating in the game can influence the outcome. We can explore how using the following code:

[10]:
```
summary(train_set$numGroups)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00   27.00   30.00   43.01   47.00  100.00
```

What happen when there is only one group in the game ?

[13]:
```
train_set %>%
  filter(numGroups == 1) %>%
  mutate(groupId = order(groupId)) %>%
  group_by(matchId) %>%
  select(matchId, groupId, winPlacePerc, numGroups) %>%
  head(20)
```

A grouped_df: 20 × 4

matchId	groupId	winPlacePerc	numGroups
<chr>	<int>	<dbl>	<int>
f3a64f99badeca	60	0	1
36ba8957ba552a	238	0	1
2c7ee565a600c6	387	0	1
0867d8854b101b	487	0	1
f3a64f99badeca	495	0	1
9929e0c83364c7	534	0	1
2798696f6875d7	535	0	1
fb9f622215e151	593	0	1
2c7ee565a600c6	642	0	1
40b8e09ef5575e	673	0	1
2e13b88e3f46ab	721	0	1
f0a26d38d1078c	761	0	1
fe6875c748e67d	983	0	1
25e2ac7d9040f0	1013	0	1
0d23db0c80a7ad	1091	0	1
6bc84f9ade1041	1101	0	1
3f4fe53ee073b1	1118	0	1
a697baffd089d6	1122	0	1
56837f98325977	155	0	1
f9f3a4cf607eca	386	0	1

When there is only 1 team the game is automatically lost. For the prediction model we can confidently filter all observations with numGroup = 1

1.2.2 Prediction algorithm

The goal of the prediction algorithm is to return the winPlacePerc given the player stats from previous games. WinPlacePerc can be any number from 0 to 1, this mean it is a continuous variable therefore it is a regression problem.

Linear regression is used to predict the value of an outcome variable Y based on one or more input predictor (feature) variables X. The aim is to establish a linear relationship between the predictor variable(s) and the target variable, so that, we can use this formula to estimate the value of the response Y, when only the predictors (Xs) values are known. The general mathematical formula of linear model is:

$Y = \beta_0 + \beta_1 x + \epsilon$

where, β_0 is the intercept and β_1 is the slope. Collectively, they are called regression coefficients. ϵ is the error term, the part of Y the regression model is unable to explain.

To measure the quality of all the models we are going to design we are going to use the Mean Absulute Error which measure the errors between paired observations expressing the same phenomenon. The MAE formula is:

$MAE = \frac{\sum_{i=1}^{N} |y_i - x_i|}{N}$

We start loading all the needed tools and options:

[5]:
```r
#Load packages.

library(tidyverse)
library(data.table)
library(caret)
library(h2o)

#options and set up
conn_h2o<- h2o.init(nthreads = 4)
```

```
Loading required package: lattice


Attaching package: 'caret'


The following object is masked from 'package:purrr':

    lift




----------------------------------------------------------------------

Your next step is to start H2O:
    > h2o.init()

For H2O package documentation, ask for help:
    > ??h2o

After starting H2O, you can use the Web UI at http://localhost:54321
For more information visit http://docs.h2o.ai

----------------------------------------------------------------------



Attaching package: 'h2o'


The following objects are masked from 'package:data.table':

    hour, month, week, year


The following objects are masked from 'package:stats':
```

```
    cor, sd, var


The following objects are masked from 'package:base':

    %*%, %in%, &&, ||, apply, as.factor, as.numeric, colnames,
    colnames<-, ifelse, is.character, is.factor, is.numeric, log,
    log10, log1p, log2, round, signif, trunc




H2O is not running yet, starting it now…

Warning message in .h2o.startJar(ip = ip, port = port, name = name, nthreads =
nthreads, :
"You have a 32-bit version of Java. H2O works best with 64-bit Java.
Please download the latest Java SE JDK from the following URL:
https://www.oracle.com/technetwork/java/javase/downloads/index.html"


Note:  In case of errors look at the following log files:
    C:\Users\elekt\AppData\Local\Temp\RtmpcL9bH5\file5a145d4663/h2o_elekt_starte
d_from_r.out
    C:\Users\elekt\AppData\Local\Temp\RtmpcL9bH5\file5a1422f3333a/h2o_elekt_star
ted_from_r.err


Starting H2O JVM and connecting:  Connection successful!

R is connected to the H2O cluster:
    H2O cluster uptime:         2 seconds 548 milliseconds
    H2O cluster timezone:       Europe/Berlin
    H2O data parsing timezone:  UTC
    H2O cluster version:        3.30.0.3
    H2O cluster version age:    13 days
    H2O cluster name:           H2O_started_from_R_elekt_fkq405
    H2O cluster total nodes:    1
    H2O cluster total memory:   0.97 GB
    H2O cluster total cores:    12
    H2O cluster allowed cores:  4
    H2O cluster healthy:        TRUE
    H2O Connection ip:          localhost
    H2O Connection port:        54321
    H2O Connection proxy:       NA
    H2O Internal Security:      FALSE
    H2O API Extensions:         Amazon S3, Algos, AutoML, Core V3,
TargetEncoder, Core V4
    R Version:                  R version 4.0.0 (2020-04-24)
```

We have seen in the EDA that team work, the exploration of the map and the boosts and perks found in the game have a big impact on the target value. To fully take them in consideration we can create 3 more different features like this:

[6]:
```
FPS <- function(x){
  x <- x %>%
    mutate(teamwork = (assists*0.2)+(revives*0.8),
           avgDistPerMinute = (walkDistance+rideDistance+swimDistance)/
 ↪(matchDuration/60),
           itemsFound = (weaponsAcquired+heals+boosts))
  return(x)
}

train_set <- FPS(train_set)
```

To simplify the train_set e speed up the training procedure we can drop useless features:

[7]:
```
less_imp <- c("vehicleDestroys","roadKills ","teamKills", "maxPlace")

train_set <- train_set[, -which(names(train_set) %in% less_imp)]
```

As we have seen in the EDA when there is only 1 team the game is automatically lost, therefore wecan remove games with numGroup == 1

[8]:
```
train_set <- train_set %>%
  filter(numGroups > 1)

test_set <- test_set %>%
  filter(numGroups > 1)
```

Now we can split the train_set in train and validation, since the test_set provided cames without the target value winPlacePerc:

[9]:
```
val_index<- createDataPartition(train_set$winPlacePerc, p = 0.2, list = FALSE)
validation<- train_set[val_index,]
train<- train_set[-val_index,]
```

We can start building the first model using a Generalized Linear Models and the faetures with a hight correlation:

[13]:
```
glm_fit <- train(winPlacePerc ~ walkDistance + rideDistance +
                               swimDistance + avgDistPerMinute +
                               kills + matchDuration +
                               assists + boosts +
                               killPoints + winPoints +
                               numGroups + headshotKills +
                               teamwork + itemsFound ,
```

```
                                    method = "glm" , data = train)
```

now we can examine the computed model:

[14]:
```
glm_fit
```

```
Generalized Linear Model

3556654 samples
     14 predictor

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 3556654, 3556654, 3556654, 3556654, 3556654, 3556654, ...
Resampling results:

  RMSE       Rsquared   MAE
  0.1537612  0.7497442  0.1166283
```

[15]:
```
summary(glm_fit)
```

```
Call:
NULL

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-4.4867  -0.1061  -0.0089   0.0877   0.9211

Coefficients:
                   Estimate Std. Error  t value Pr(>|t|)
(Intercept)       2.933e-01  7.056e-04  415.697  < 2e-16 ***
walkDistance      6.209e-05  3.152e-07  196.951  < 2e-16 ***
rideDistance     -6.216e-05  2.624e-07 -236.853  < 2e-16 ***
swimDistance      8.114e-06  2.726e-06    2.976 0.002918 **
avgDistPerMinute  2.762e-03  8.018e-06  344.509  < 2e-16 ***
kills             1.028e-02  8.027e-05  128.078  < 2e-16 ***
matchDuration    -1.184e-04  4.301e-07 -275.285  < 2e-16 ***
assists           9.907e-03  1.621e-04   61.106  < 2e-16 ***
boosts           -3.088e-04  8.097e-05   -3.814 0.000137 ***
killPoints       -3.698e-05  7.320e-07  -50.524  < 2e-16 ***
winPoints         3.183e-05  6.196e-07   51.382  < 2e-16 ***
numGroups         1.655e-03  3.618e-06  457.540  < 2e-16 ***
headshotKills    -1.586e-03  1.836e-04   -8.641  < 2e-16 ***
teamwork          1.981e-02  2.308e-04   85.854  < 2e-16 ***
itemsFound        1.107e-02  2.687e-05  412.011  < 2e-16 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 0.02365459)

    Null deviance: 336004  on 3556653  degrees of freedom
Residual deviance:  84131  on 3556639  degrees of freedom
AIC: -3223453

Number of Fisher Scoring iterations: 2
```

[16]:
```
varImp(glm_fit$finalModel)
```

A data.frame: 14×1

	Overall <dbl>
walkDistance	196.951106
rideDistance	236.853435
swimDistance	2.976283
avgDistPerMinute	344.508904
kills	128.078155
matchDuration	275.284949
assists	61.106450
boosts	3.813544
killPoints	50.524154
winPoints	51.381961
numGroups	457.540380
headshotKills	8.641428
teamwork	85.853804
itemsFound	412.011176

We can use the model to predict the target value from of the validation set:

[17]:
```
winPerc_glm_yhat <- predict(glm_fit, validation)
```

and calculate the MAE:

[18]:
```
postResample(pred = winPerc_glm_yhat, obs = validation$winPlacePerc)
```

RMSE 0.153634502226424 **Rsquared** 0.750053743744969 **MAE** 0.116538090911578

To reach a better MAE since we have a very large and complicated data set we could try a more complex and deep approack. Using the H2o package we can build a neural network and compute a deep learning model.

Neural Network (or Artificial Neural Network) has the ability to learn by examples. ANN is an information processing model inspired by the biological neuron system. It is composed of a large number of highly interconnected processing elements known as the neuron to solve problems. It follows the non-linear path and process information in parallel throughout the nodes. A neural network is a complex adaptive system. Adaptive means it has the ability to change its internal structure by adjusting weights of inputs. The neural network was designed to solve problems which are easy for humans and difficult for machines which are often referred to as pattern recognition.

We can start building our NN preparing the data as requested from the H2o package. H2o is an open source, in-memory, distributed, fast, and scalable machine learning and predictive analytics platform that allows you to build machine learning models on big data.

[10]:
```
train.h2o <- as.h2o(train)
sampled_train <- as.h2o(train[1:10000,]) #For speed we create a 10K rows train␣
↪data set for model tuning
sampled_valid <- as.h2o(train[10001:20000,])
validation.h2o <- as.h2o(validation)
test.h2o <- as.h2o(test_set)
```

```
|======================================================================| 100%
|======================================================================| 100%
|======================================================================| 100%
|======================================================================| 100%
|======================================================================| 100%
```

We have also created a smaller train set just for the purpose of tuning parameters for the deep learning training. Now we identify the position of the target value and the features:

[11]:
```
colnames(train.h2o)

target_val <- 26
features <- c(4:25,27:29)
```

1. 'Id' 2. 'groupId' 3. 'matchId' 4. 'assists' 5. 'boosts' 6. 'damageDealt' 7. 'DBNOs' 8. 'headshotKills' 9. 'heals' 10. 'killPlace' 11. 'killPoints' 12. 'kills' 13. 'killStreaks' 14. 'longestKill' 15. 'matchDuration' 16. 'matchType' 17. 'numGroups' 18. 'rankPoints' 19. 'revives' 20. 'rideDistance' 21. 'roadKills' 22. 'swimDistance' 23. 'walkDistance' 24. 'weaponsAcquired' 25. 'winPoints' 26. 'winPlacePerc' 27. 'teamwork' 28. 'avgDistPerMinute' 29. 'itemsFound'

Before launching an entire deep learning training we should find the best paramenters accordig to aour data set:

[12]:
```
hyper_params <- list(activation = c("Rectifier","Tanh"),
                     hidden = list(c(50, 50, 50), c(200, 200), c(100, 100),␣
↪c(20,20)),
                     input_dropout_ratio = c(0, 0.05),
                     l1=seq(0,1e-4,1e-6),
                     l2=seq(0,1e-4,1e-6))
```

Selecting optimal model search criteria. Search will stop once top 5 models are within 1% of each other:

[13]:
```
search_criteria <- list(strategy = "RandomDiscrete",
                       max_runtime_secs = 600,
                       max_models = 200,
```

```
                        seed=1234567,
                        stopping_rounds=10,
                        stopping_tolerance=1e-2)
```

Now we are ready to perform a random hyper parameters search having as a main parameter MAE:

[14]:
```
dl_random_grid <- h2o.grid(
  algorithm="deeplearning",
  grid_id = "dl_grid_random",
  training_frame=sampled_train,
  validation_frame=sampled_valid,
  x=features,
  y=target_val,
  epochs=5,
  stopping_metric="MAE",
  stopping_tolerance=1e-2,        ## stop when MAE does not improve by >=1% for
 ↪2 scoring events
  stopping_rounds=3,
  max_w2=10,                      ## can help improve stability for Rectifier
  hyper_params = hyper_params,
  search_criteria = search_criteria
)
```

```
  |======================================================================| 100%
```

Extrapolate the tuning search results:

[16]:
```
grid <- h2o.getGrid("dl_grid_random",sort_by="MAE", decreasing=FALSE);grid
```

```
H2O Grid Details
================

Grid ID: dl_grid_random
Used hyper parameters:
  -  activation
  -  hidden
  -  input_dropout_ratio
  -  l1
  -  l2
Number of models: 57
Number of failed models: 0

Hyper-Parameter Search Summary: ordered by increasing MAE
  activation     hidden input_dropout_ratio      l1      l2
1  Rectifier [200, 200]                 0.0 2.4E-5 2.3E-5
2  Rectifier [200, 200]                 0.0 7.4E-5 5.8E-5
3       Tanh [200, 200]                 0.0 1.3E-5 1.0E-4
4       Tanh [100, 100]                 0.0 8.9E-5 1.9E-5
```

```
5  Rectifier [200, 200]                0.05 9.2E-5    0.0
                model_ids                 mae
1 dl_grid_random_model_54 0.07087981892896002
2 dl_grid_random_model_21 0.07159506162527687
3 dl_grid_random_model_35 0.07349126243968858
4 dl_grid_random_model_48 0.07353105008718545
5 dl_grid_random_model_38 0.07416571789004218

---
   activation     hidden input_dropout_ratio     l1     l2
52       Tanh   [20, 20]                0.05 5.5E-5 6.9E-5
53  Rectifier [100, 100]                0.05 2.5E-5 8.8E-5
54       Tanh   [20, 20]                0.05 1.6E-5 3.2E-5
55       Tanh   [20, 20]                0.05 7.1E-5 5.4E-5
56       Tanh [200, 200]                0.05 2.3E-5 1.2E-5
57  Rectifier   [20, 20]                0.05 2.9E-5    0.0
                 model_ids                 mae
52 dl_grid_random_model_51 0.08108724170206656
53 dl_grid_random_model_40  0.0813677124296093
54 dl_grid_random_model_11 0.08158950299993807
55 dl_grid_random_model_37  0.0833742441174554
56 dl_grid_random_model_57 0.08441922166779707
57  dl_grid_random_model_8  0.0949530963326015
```

We already can see witch is the best model according to MAE parameter. With the next code we can see more details about it:

[17]:
```r
best_model <- h2o.getModel(grid@model_ids[[1]]);best_model
```

```
Model Details:
==============

H2ORegressionModel: deeplearning
Model ID:  dl_grid_random_model_54
Status of Neuron Layers: predicting winPlacePerc, regression, gaussian distribution, Quadratic
  layer units      type dropout       l1       l2 mean_rate rate_rms momentum
1     1    24     Input  0.00 %       NA       NA        NA       NA       NA
2     2   200 Rectifier  0.00 % 0.000024 0.000023  0.019735 0.026301 0.000000
3     3   200 Rectifier  0.00 % 0.000024 0.000023  0.188049 0.209986 0.000000
4     4     1    Linear      NA 0.000024 0.000023  0.001799 0.001220 0.000000
  mean_weight weight_rms mean_bias bias_rms
1          NA         NA        NA       NA
2    0.004932   0.098633  0.369844 0.062396
3   -0.019728   0.064233  0.921812 0.037394
4   -0.009642   0.051063  0.047039 0.000000
```

```
H2ORegressionMetrics: deeplearning
** Reported on training data. **
** Metrics reported on full training frame **

MSE:  0.008778115
RMSE:  0.09369159
MAE:  0.06702661
RMSLE:  0.06368166
Mean Residual Deviance :  0.008778115


H2ORegressionMetrics: deeplearning
** Reported on validation data. **
** Metrics reported on full validation frame **

MSE:  0.01011592
RMSE:  0.1005779
MAE:  0.07087982
RMSLE:  0.06830146
Mean Residual Deviance :  0.01011592
```

the best model according the tuning procedure has a Rectifier activation, 2 hidden layers with 200 neurons each and an optimization for L1/L2 parameters.

L1 regularization can add stability and improve generalization, causes many weights to become 0). Defaults is 0. L2 regularization can add stability and improve generalization, causes many weights to be small. Defaults to 0. Meaning, L1 lets only strong weights survive (constant pulling force towards zero), while L2 prevents any single weight from getting too big.

Now we can see all the other optimized parameters:

[18]:
```
best_params <- best_model@allparameters;best_params
```

$model_id 'dl_grid_random_model_54'

$training_frame 'data.frame_sid_9801_3'

$validation_frame 'data.frame_sid_9801_5'

$nfolds 0

$keep_cross_validation_models TRUE

$keep_cross_validation_predictions FALSE

$keep_cross_validation_fold_assignment FALSE

$fold_assignment 'AUTO'

$ignore_const_cols TRUE

$score_each_iteration FALSE

$balance_classes FALSE

$max_after_balance_size 5

$max_confusion_matrix_size 20

$max_hit_ratio_k 0

$overwrite_with_best_model TRUE

$use_all_factor_levels TRUE

$standardize TRUE

$activation 'Rectifier'

$hidden 1. 200 2. 200

$epochs 5

$train_samples_per_iteration -2

$target_ratio_comm_to_comp 0.05

$seed 1234820

$adaptive_rate TRUE

$rho 0.99

$epsilon 1e-08

$rate 0.005

$rate_annealing 1e-06

$rate_decay 1

$momentum_start 0

$momentum_ramp 1e+06

$momentum_stable 0

$nesterov_accelerated_gradient TRUE

$input_dropout_ratio 0

$l1 2.4e-05

$l2 2.3e-05

$max_w2 10

$initial_weight_distribution 'UniformAdaptive'

$initial_weight_scale 1

$loss 'Automatic'

$distribution 'AUTO'

$quantile_alpha 0.5

$tweedie_power 1.5

$huber_alpha 0.9

$score_interval 5

$score_training_samples 10000

$score_validation_samples 0

$score_duty_cycle 0.1

$classification_stop 0

$regression_stop 1e-06

$stopping_rounds 3

$stopping_metric 'MAE'

$stopping_tolerance 0.01

$max_runtime_secs 92.041

$score_validation_sampling 'Uniform'

$diagnostics TRUE

$fast_mode TRUE

$force_load_balance TRUE

$variable_importances TRUE

$replicate_training_data TRUE

$single_node_mode FALSE

$shuffle_training_data FALSE

$missing_values_handling 'MeanImputation'

$quiet_mode FALSE

$autoencoder FALSE

$sparse FALSE

$col_major FALSE

$average_activation 0

$sparsity_beta 0

$max_categorical_features 2147483647

$reproducible FALSE

$export_weights_and_biases FALSE

$mini__batch__size 1

$categorical__encoding 'AUTO'

$elastic__averaging FALSE

$elastic__averaging__moving__rate 0.9

$elastic__averaging__regularization 0.001

$x 1. 'assists' 2. 'boosts' 3. 'damageDealt' 4. 'DBNOs' 5. 'headshotKills' 6. 'heals' 7. 'killPlace' 8. 'killPoints' 9. 'kills' 10. 'killStreaks' 11. 'longestKill' 12. 'matchDuration' 13. 'numGroups' 14. 'rankPoints' 15. 'revives' 16. 'rideDistance' 17. 'roadKills' 18. 'swimDistance' 19. 'walkDistance' 20. 'weaponsAcquired' 21. 'winPoints' 22. 'teamwork' 23. 'avgDistPerMinute' 24. 'itemsFound'

$y 'winPlacePerc'

We are ready to launch the final deep learning model with optimized parameters:

[19]:
```
dl_model <- h2o.deeplearning(model_id = "dl_all_data_1",
                             training_frame=train.h2o,
                             x=features,
                             y=target_val,
                             hidden = c(200,200),
                             activation = "Rectifier",
                             stopping_metric="MAE",
                             stopping_tolerance=1e-2,
                             score_validation_samples=10000, # downsample␣
 ↪validation set for faster scoring
                             score_duty_cycle=0.025,         # don't score more␣
 ↪than 2.5% of the wall time
                             seed = 1234567,
                             epochs = 10,
                             l1 = 2.4e-05,
                             l2 = 2.3e-05,
                             max_w2 = 10,)
```

```
Warning message in .h2o.processResponseWarnings(res):
"Dropping bad and constant columns: [matchType].
"

  |======================================================================| 100%
```

Examine the deep learning model:

[20]:
```
dl_model
```

```
Model Details:
==============

H2ORegressionModel: deeplearning
```

```
Model ID:  dl_all_data_1
Status of Neuron Layers: predicting winPlacePerc, regression, gaussian distribution, Quadratic
  layer units       type dropout       l1       l2 mean_rate rate_rms momentum
1     1    24      Input  0.00 %       NA       NA        NA       NA       NA
2     2   200 Rectifier  0.00 % 0.000024 0.000023  0.073637 0.184437 0.000000
3     3   200 Rectifier  0.00 % 0.000024 0.000023  0.360333 0.317859 0.000000
4     4     1     Linear      NA 0.000024 0.000023  0.007816 0.007345 0.000000
  mean_weight weight_rms mean_bias bias_rms
1          NA         NA        NA       NA
2    0.000405   0.118615 -0.074951 0.246706
3   -0.039029   0.092543  0.489384 0.274598
4   -0.008985   0.088698  0.521139 0.000000


H2ORegressionMetrics: deeplearning
** Reported on training data. **
** Metrics reported on temporary training frame with 9985 samples **

MSE:  0.00702743
RMSE:  0.08382977
MAE:  0.06014526
RMSLE:  0.05672961
Mean Residual Deviance :  0.00702743
```

We can immediately see a better MAE result **0.06014** Let's explore which feature the model judge important:

[21]:
```r
head(as.data.frame(h2o.varimp(dl_model)))
```

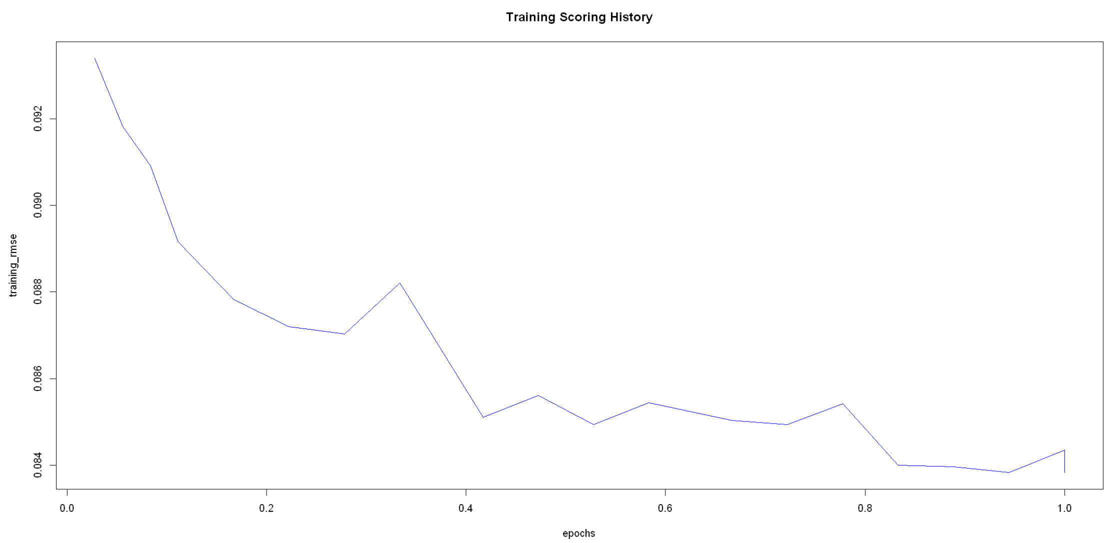
A data.frame: 6×4

	variable <chr>	relative_importance <dbl>	scaled_importance <dbl>	percentage <dbl>
1	numGroups	1.0000000	1.0000000	0.16436870
2	killPlace	0.8141788	0.8141788	0.13382552
3	kills	0.4760079	0.4760079	0.07824080
4	walkDistance	0.4249777	0.4249777	0.06985303
5	matchDuration	0.4067536	0.4067536	0.06685757
6	killStreaks	0.3701204	0.3701204	0.06083621

Strangely the most important one is numGroups differently from the glm model where the most important one is walkDistance which makes a lot more sense.

We can also plot the training history of the model:

[22]:
```r
plot(dl_model)
```



Now let's generate the prediction using this model and test them with the rest of the data:

[23]:
```
dl_result <-as.data.frame(h2o.predict(dl_model, validation.h2o))
```

```
|======================================================================| 100%
```

[24]:
```
MAE(dl_result$predict,validation$winPlacePerc)
```

0.0606931647663832

The final MAE on the test data is **0.0606** which seems correct according the previous result.

1.3 Conclusion

This was a very challenging case where the knowledge of the game dynamics is essential to find the correct path. The EDA part was fundamental to extrapolate the most important features and understand the data.

Both approach gives good results and the glm model mimic the logic behind the importance of map exploration during the game. Never the less according to the initial KPI, the Mean Absolute Error the best model seems to be the deep learning created using an artificial neural network.

Much more improvement can be done on the deep learning approach exploring all the other tunable parameters.

[ ]:
```

```