Pokemon Winner Prediction

Ana Maria Sandoval Jimenez, Arthur Gogohia, Shuaib Yunus, Viktor Treu

13/01/2019

Contents

1 Introduction

"Pokemon are fantasy creatures in the series of video games of the same name. The games were originally developed by Satoshi Tajiri and the Japanese game software company GAME FREAK Inc. and represent one of the most important franchises of the publisher Nintendo. The Pokémon can be captured, collected and trained by the player. The success of the game, which was first released in 1996, was followed by an anime television series, a trading card game, a large number of merchandising products and, since 1998, 21 feature films. The video games of the Pokémon franchise have sold more than 200 million copies worldwide" (Wikipedia 2019)

1.1 Task

Our goal is to predict the probability of winning for a Pokemon using supervised learning methods. We will use two regression models to do that. First, we will use Linear regression (Ridge and Lasso), and second we will use Decision Tree. The preprocessed data will be split into 60% train, 20% validation and 20% test data.

1.2 Tools

In the process of the group work, we have been using Github and R Studio. With Github we were able to share our R scripts. R Studio has been the choice of our local working environment due to the fact that lab sessions are carried out in R Studio. Hence the environment was known to each team member.

1.3 Data

The two dataset used in the project were obtained from Nintendo's famous original Pokemon video game.

The pokemon dataset contains a full set of in-game statistics for 800 pokemon in the 6 generations of video games that form the Pokemon world. It also includes data about each pokemon's description, power, type and generation it belongs to. The data is stored in `pokemon.csv`.

The combats dataset is a collection of 50,000 combats between two pokemons and the pokemon that won. This data is stored in combats.csv.

1.3.1 Pokemon Dataset

Column	Description
X.	ID for each pokemon
Name	Name of each pokemon
Type 1	Each pokemon has a type, this determines weakness/resistance to attacks
Type 2	Some pokemon are dual type and have 2
HP	hit points, or health, defines how much damage a pokemon can withstand before fainting
Attack	the base modifier for normal attacks (eg. Scratch, Punch)
Defense	the base damage resistance against normal attacks
SP Atk	special attack, the base modifier for special attacks (e.g. fire blast, bubble beam)
SP Def	the base damage resistance against special attacks
Speed	determines which pokemon attacks first each round
Generation	Video game version
Legendary	if the Pokemon is legendary or not

Pokemon generation refers to a chronological division by release. The first generation of the video game came up in 1996 "Pokemon Red and Green" for Game Boy (Wikipedia 2019), each new generation brings to life new Pokemon, characters and features. In the dataset there are Pokemon from 6 generations.

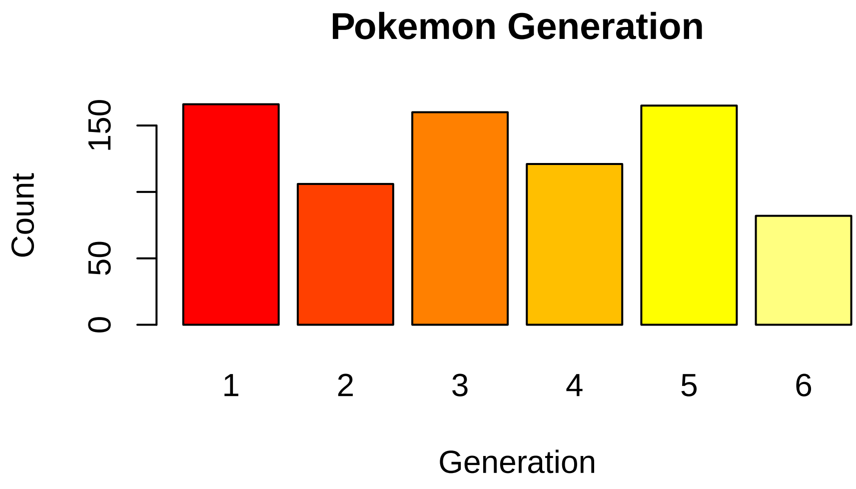
```
print('Pokemon Generation')
```

```
## [1] "Pokemon Generation"
```

```
table(pokemon$Generation)
```

```
##
##   1   2   3   4   5   6
## 166 106 160 121 165  82
```

```
barplot(table(pokemon$Generation), col=heat.colors(6),
        main = 'Pokemon Generation', xlab = 'Generation',
        ylab = 'Count')
```



Type 1 and type 2 refers to the main trait or element that the Pokemon possesses. Some pokemon have two main traits, while others only have one. Having several types of traits can present an advantage for a Pokemon during a fight.

```
print('Pokemon Type 1')
```

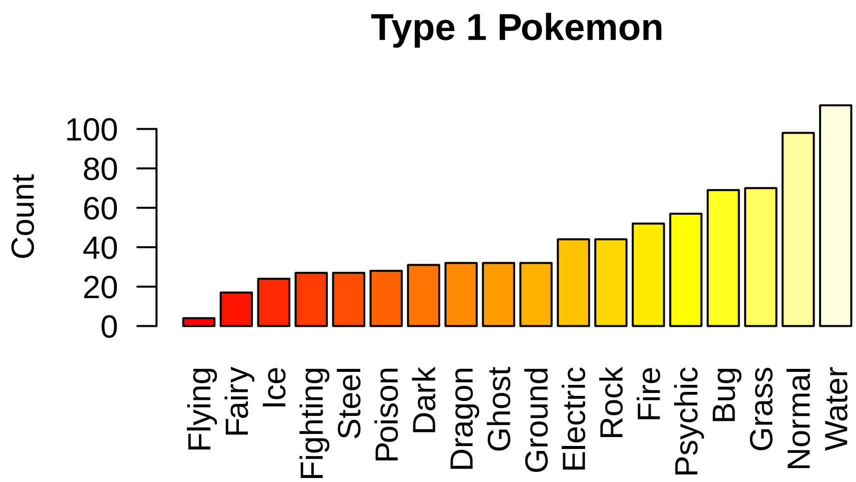
```
## [1] "Pokemon Type 1"
```

```r
sort(table(pokemon$Type.1))
```

```
##
##   Flying    Fairy      Ice Fighting    Steel   Poison     Dark   Dragon
##        4       17       24       27       27       28       31       32
##    Ghost   Ground Electric     Rock     Fire  Psychic      Bug    Grass
##       32       32       44       44       52       57       69       70
##   Normal    Water
##       98      112
```

```r
par(las=2)

barplot(sort(table(pokemon$Type.1)),
        col = heat.colors(length(unique(pokemon$Type.1))),
        main = 'Type 1 Pokemon',
        ylab = 'Count')
```



The most common type of pokemon is the Water type with 112 pokemon, followed by Normal type with 98 pokemon, and Grass type with 70 pokemon. The least common type 1 among the 800 pokémon is the Flying type with only 4 pokémon.

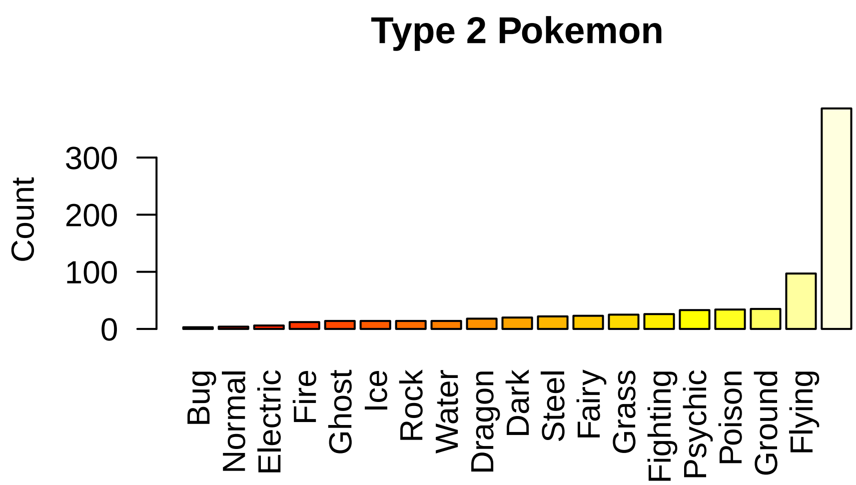
```r
par(las=2)
print('Pokemon Type 2')
```

```
## [1] "Pokemon Type 2"
```

```r
sort(table(pokemon$Type.2))
```

```
##
##      Bug   Normal Electric     Fire    Ghost      Ice     Rock    Water
##        3        4        6       12       14       14       14       14
##   Dragon     Dark    Steel    Fairy    Grass Fighting  Psychic   Poison
##       18       20       22       23       25       26       33       34
##   Ground   Flying
##       35       97      386
```

```r
barplot(sort(table(pokemon$Type.2)),
        col = heat.colors(length(unique(pokemon$Type.2))),
        main = 'Type 2 Pokemon', ylab = 'Count')
```



For 386 pokemon, there is no type 2 recorded. This simply indicates that for these pokemones there is only one type of element in which they are strong. Another interesting fact is that the type Flying as type 1 is very rare, but as type 2 it's one of the most common.

The Pokemon table has these dimensions:

```r
dim(pokemon)
```

```
## [1] 800  12
```

The column names as described before are:

```r
names(pokemon)
```

```
##  [1] "X."         "Name"       "Type.1"     "Type.2"     "HP"
##  [6] "Attack"     "Defense"    "Sp..Atk"    "Sp..Def"    "Speed"
## [11] "Generation" "Legendary"
```

This is what the data looks like:

```r
head(pokemon)
```

```
##   X.          Name Type.1 Type.2 HP Attack Defense Sp..Atk Sp..Def Speed
## 1  1     Bulbasaur  Grass Poison 45     49      49      65      65    45
## 2  2       Ivysaur  Grass Poison 60     62      63      80      80    60
## 3  3      Venusaur  Grass Poison 80     82      83     100     100    80
## 4  4 Mega Venusaur  Grass Poison 80    100     123     122     120    80
## 5  5    Charmander   Fire        39     52      43      60      50    65
## 6  6    Charmeleon   Fire        58     64      58      80      65    80
##   Generation Legendary
## 1          1     False
## 2          1     False
## 3          1     False
## 4          1     False
## 5          1     False
## 6          1     False
```

There are no missing values in the Pokemon dataset:

```r
colSums(is.na(pokemon))
```

```
##         X.       Name     Type.1     Type.2         HP     Attack
##          0          0          0          0          0          0
##    Defense    Sp..Atk    Sp..Def      Speed Generation  Legendary
```

```
##            0            0            0            0            0            0
```

1.3.2 Combat Dataset

Column	Description
First_pokemon	Pokemon that attack first
Second_pokemon	Pokemon that attack second
Winner	Winner of the combat

The entries in `combat.csv` are the ids of the Pokemon found in `pokemon.csv`. In a later step, we will convert the ids in this dataset to Pokemon names for clarity.

The Pokemon table has dimensions of:

```r
dim(fights)
```

```
## [1] 50000     3
```

The column names as described before are:

```r
names(fights)
```

```
## [1] "First_pokemon"  "Second_pokemon" "Winner"
```

This is what the data look like:

```r
head(fights)
```

```
##   First_pokemon Second_pokemon Winner
## 1           266            298    298
## 2           702            701    701
## 3           191            668    668
## 4           237            683    683
## 5           151            231    151
## 6           657            752    657
```

There are no missing values in the Pokemon dataset:

```r
colSums(is.na(fights))
```

```
##  First_pokemon Second_pokemon         Winner
##              0              0              0
```

This combat dataset contains fifty thousand battles between pokemon and the corresponding winner. Here we have grouped the fights by winner.

```r
print('Winner Table')
```

```
## [1] "Winner Table"
```

```r
wins <- group_by(fights, Winner)
summarise(wins, count = n())
```

```
## # A tibble: 783 x 2
##    Winner count
##     <int> <int>
##  1      1    37
##  2      2    46
##  3      3    89
```

```
## 4        4    70
## 5        5    55
## 6        6    64
## 7        7   115
## 8        8   119
## 9        9   114
## 10      10    19
## # ... with 773 more rows
```

Up to this point, we've presented the data; we've indicated the purpose of this report will be to predict the probability of winning for a pokemon using two regression models (Linear regression and decision trees); and we will ultimately choose the model that is more suitable for our data.

2 Exploratory Data Analysis

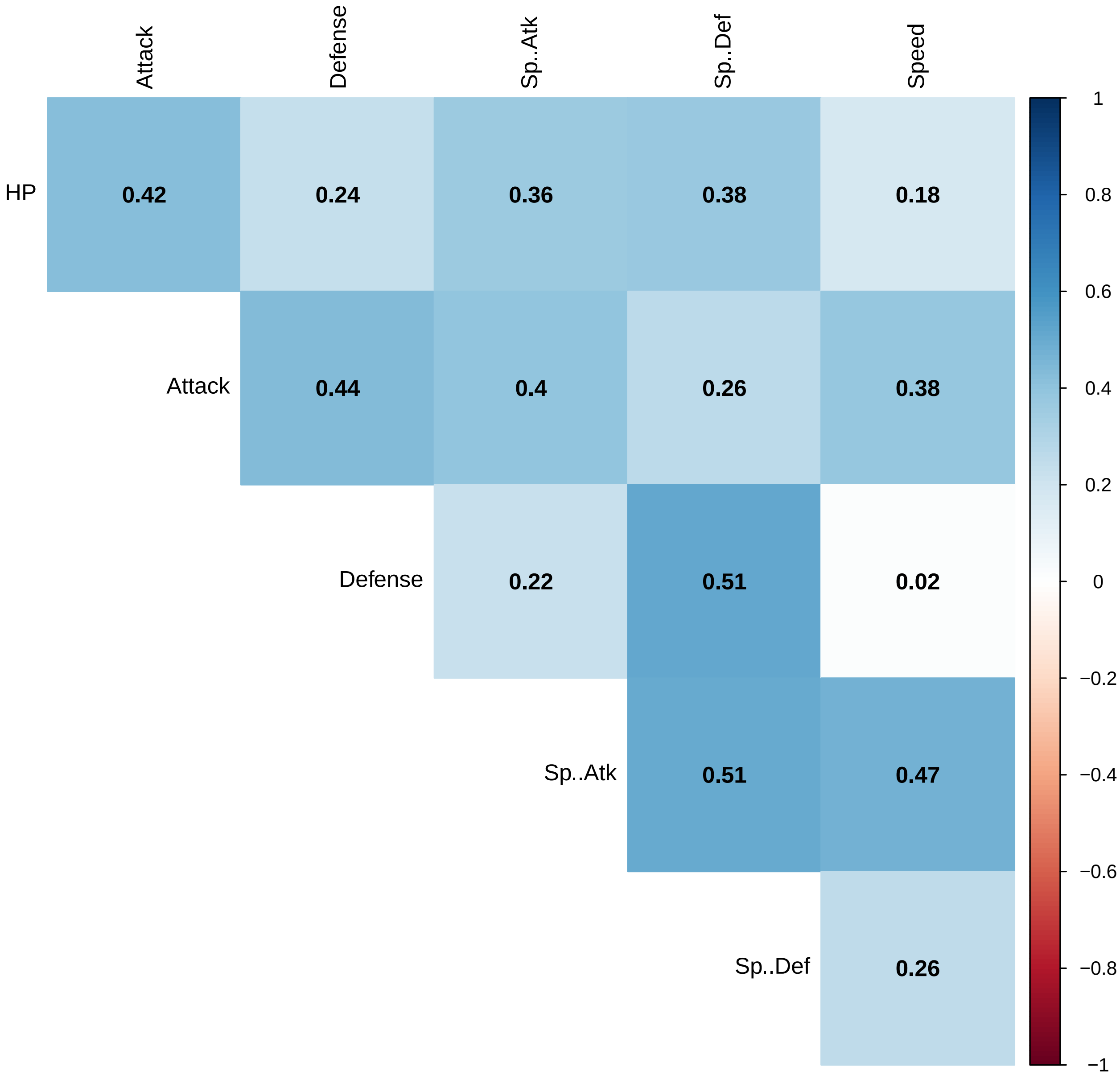
Before modeling our data, the first step is to perform a descriptive analysis of the dataset.

2.1 Data Visualization

2.1.1 Data Correlation

To check for any statistical association between the pokemon features, we create a correlation plot.

```r
# visualization
# feature correlation
# correlation table for features
featCorr <- cor(select(pokemon, HP, Attack, Defense, Sp..Atk, Sp..Def, Speed ))
corrplot(featCorr, method = "color", type = "upper",
         addCoef.col = "black",tl.col = "black", diag = FALSE)
```



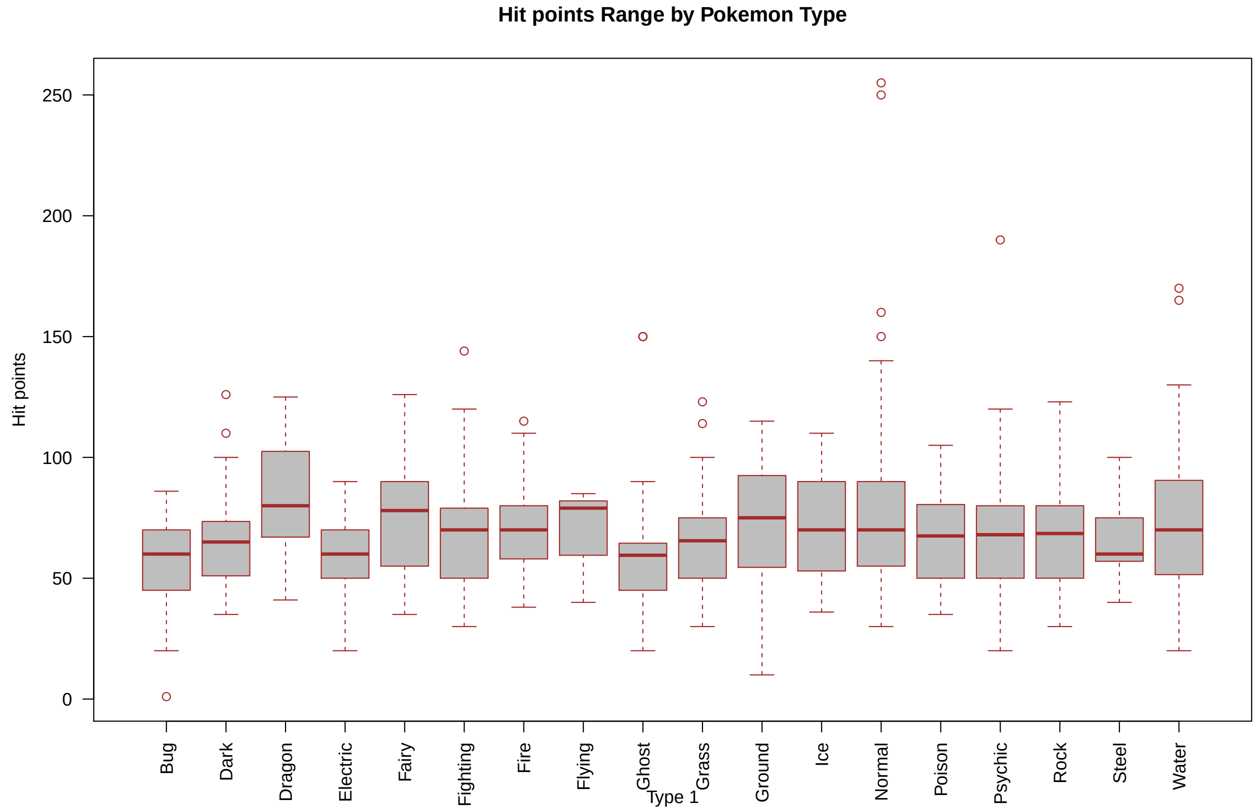
When looking at the correlation plot, there is no strong correlation (greater than 0.80) between any two features. The highest correlation is between Defense vs. Sp..Def and Sp..Atk vs..Def, which in both cases is a positive correlation of 0.51. The cor values indicates how close two variables are to having a linear relationship with each other (Wikipedia 2019).

Testing for correlations is useful because it can indicate dependency, and highly correlated variables are not desirable for modeling.

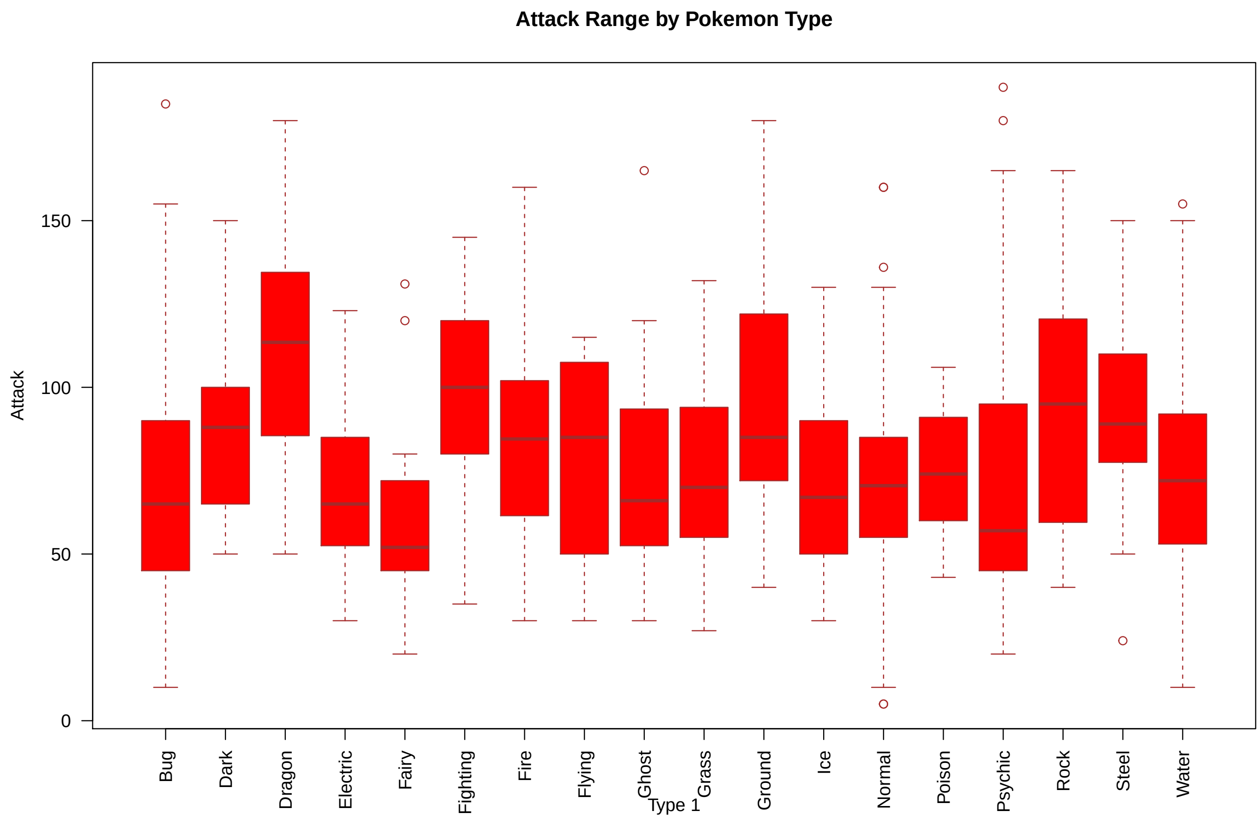
2.1.2 Feature Distribution by Pokemon Type

```
# plot specific feature distribution by Pokemon type
par(las=2)
boxplot(HP ~ Type.1,
        data=pokemon,
        main="Hit points Range by Pokemon Type",
        xlab="Type 1",
```

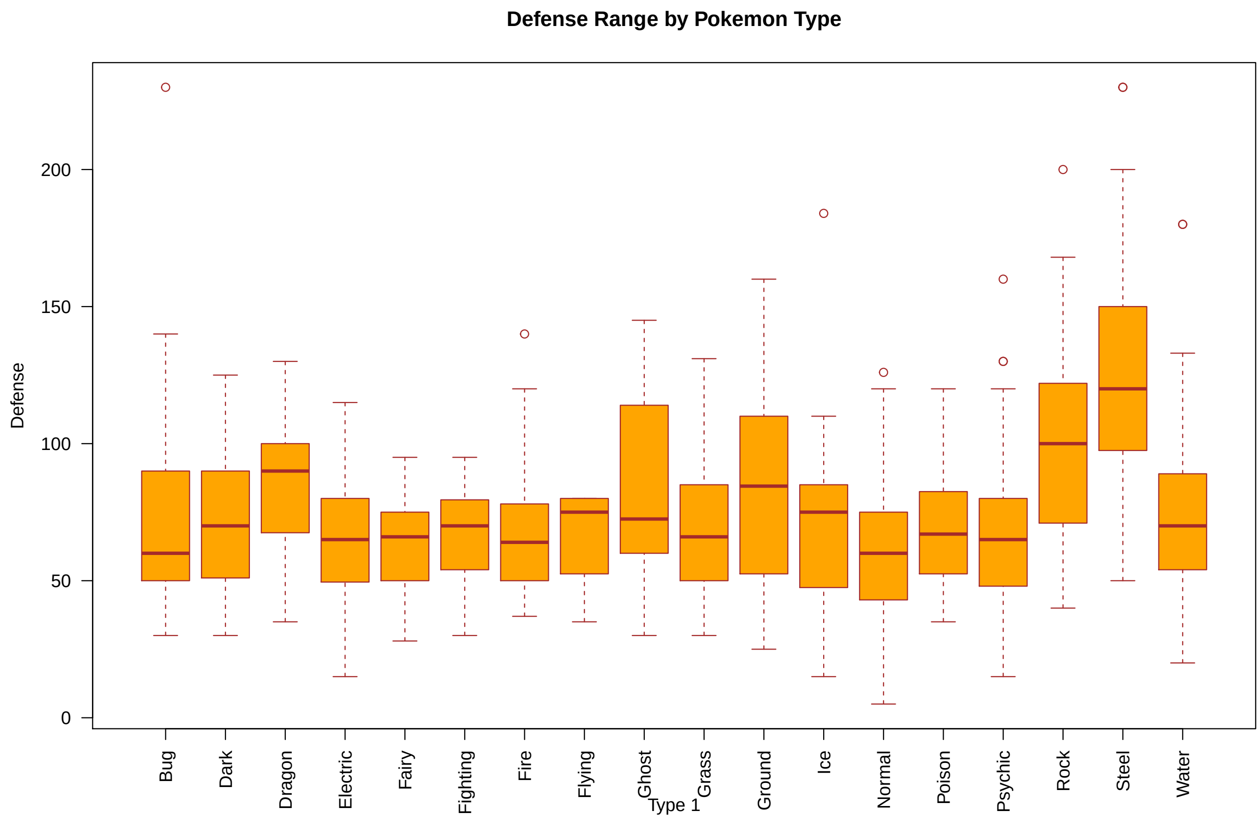
```
        ylab="Hit points",
        col="grey",
        border="brown"
)
```




```
par(las=2)
boxplot(Attack ~ Type.1,
        data=pokemon,
        main="Attack Range by Pokemon Type",
        xlab="Type 1",
        ylab="Attack",
        col="red",
        border="brown"
)
```




```r
par(las=2)
boxplot(Defense ~ Type.1,
        data=pokemon,
        main="Defense Range by Pokemon Type",
        xlab="Type 1",
        ylab="Defense",
        col="orange",
        border="brown"
)
```



2.1.3

The sum of all the numerical features is a good indication of how strong a pokemon is. Next, there is a plot that brings together all the numerical characteristics of the Pokemon dataset. It shows which type of Pokemon is the strongest and therefore most likely to win a battle.

```
# set color for each pokemon type for plotting
# hex codes from http://www.epidemicjohto.com/t882-type-colors-hex-colors
color<-c("#6F35FC","#B7B7CE","#A98FF3","#F95587","#B6A136","#EE8130","#F7D02C",
         "#705746","#735797","#E2BF65","#96D9D6","#6390F0","#7AC74C","#C22E28",
         "#D685AD","#A33EA1","#A8A77A","#A6B91A")

# pokemon characteristics
# select particular features
res <- select(pokemon, Type.1, HP, Attack, Defense, Sp..Atk, Sp..Def, Speed)
# group by pokemon type
res <- group_by(res, Type.1)
# get mean values for the types
res <- summarise_all(res, funs(mean))
# sum up all mean values
res <- mutate(res, sumChars = HP + Attack + Defense + Sp..Atk + Sp..Def + Speed)
```

```
## Warning: `as_dictionary()` is soft-deprecated as of rlang 0.3.0.
## Please use `as_data_pronoun()` instead
## This warning is displayed once per session.
```

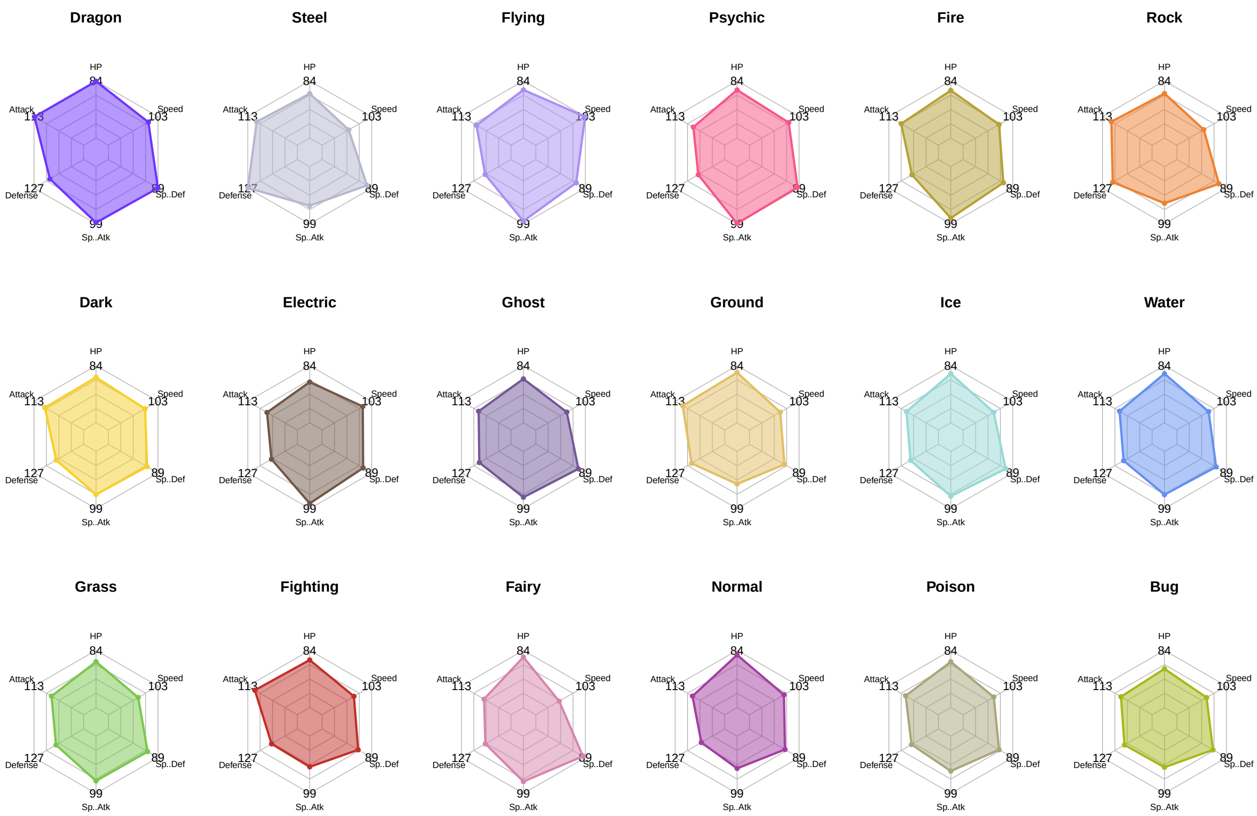
```
## Warning: `new_overscope()` is soft-deprecated as of rlang 0.2.0.
## Please use `new_data_mask()` instead
## This warning is displayed once per session.
```

```
## Warning: The `parent` argument of `new_data_mask()` is deprecated.
## The parent of the data mask is determined from either:
##
##   * The `env` argument of `eval_tidy()`
##   * Quosure environments when applicable
## This warning is displayed once per session.

## Warning: `overscope_clean()` is soft-deprecated as of rlang 0.2.0.
## This warning is displayed once per session.
```

```
# sort for values, descending
res <- arrange(res, -sumChars)
res$color<-color
# apply color scheme
max<- ceiling(apply(res[,2:7], 2, function(x)
  max(x, na.rm = TRUE)) %>% sapply(as.double)) %>% as.vector
min<-rep.int(0,6)

par(mfrow=c(3,6))
par(mar=c(1,1,1,1))
for(i in 1:nrow(res)){
  curCol<-(col2rgb(as.character(res$color[i]))%>% as.integer())/255 # convert to rgb
  radarchart(rbind(max,min,res[i,2:7]),
             axistype=2 ,
             pcol=rgb(curCol[1],curCol[2],curCol[3], alpha = 1) ,
             pfcol=rgb(curCol[1],curCol[2],curCol[3],.5) ,
             plwd=2 , cglcol="grey", cglty=1,
             axislabcol="black", caxislabels=seq(0,2000,5), cglwd=0.8, vlcex=0.8,
             title=as.character(res$Type.1[i]))
}
```



The plots are sorted from strongest to weakest in relation to the sum of their characteristics. The Dragon Type pokémon are the strongest and the Bug Type pokémon are the weakest.

2.2 Data Preprocessing for Modeling

At the start of the data preprocessing step, we found a pokemon without a name. It turned out to be the Primeape pokemon and we added it to the pokemon table.

```
levels(pokemon$Name)[levels(pokemon$Name)==""] <- "Primeape" # pokemon name was not given
```

Create an object called `names` that contains only pokemon id and pokemon name.

```
names <- pokemon[,c(1,2)]; head(names) # join id with pokemon name
```

```
##   X.          Name
## 1  1     Bulbasaur
## 2  2       Ivysaur
## 3  3      Venusaur
## 4  4 Mega Venusaur
## 5  5    Charmander
## 6  6    Charmeleon
```

```
colnames(names)
```

```
## [1] "X."    "Name"
```

We use the new object `names` to map ids to their corresponding names in the combat dataset. After the mapping, we observe that only 784 out of 800 Pokemon fought (in our dataset).

```
#Map the figths table from id to pokemon name
fights.name <- data.frame(lapply(fights, function(x) names$Name[match(x,names$X.)]))
head(fights.name)
```

```
##   First_pokemon         Second_pokemon    Winner
## 1      Larvitar                Nuzleaf   Nuzleaf
## 2      Virizion              Terrakion Terrakion
## 3       Togetic               Beheeyem  Beheeyem
## 4        Slugma              Druddigon Druddigon
## 5       Omastar                Shuckle   Omastar
## 6        Joltik Aegislash Shield Forme    Joltik
```

Of the 784 that fought in battle, one of them never won.

```
sapply(fights.name, function(x) length(unique(x))) # only 784 of 800 pokemon fought
```

```
##  First_pokemon Second_pokemon         Winner
##            784            784            783
```

next step was to write a function to get the number of times a pokemon won, a pokemon attak first and a pokemon attak second from the combat dataset.

```
get_win_table <- function() {
  counts <- group_by(fights.name, Winner)
  count_table <- summarise(counts, count = n())
  return(count_table)
}

get_firsts_table <- function() {
  counts <- group_by(fights.name, First_pokemon)
  count_table <- summarise(counts, count = n())
  return(count_table)
}

get_seconds_table <- function() {
  counts <- group_by(fights.name, Second_pokemon)
  count_table <- summarise(counts, count = n())
  return(count_table)
}

win_table <- get_win_table()
firsts_table <- get_firsts_table()
seconds_table <- get_seconds_table()
```

The next step was to add the combat table information into the pokemon table. To do this, we counted the number of times a pokemon fought, won, attacked first or second and added it to the pokemon table.

```
win_counts <- sapply(pokemon$Name, function(x)
  win_table$count[match(x,win_table$Winner)])
first_counts <- sapply(pokemon$Name, function(x)
  firsts_table$count[match(x,firsts_table$First_pokemon)])
second_counts <- sapply(pokemon$Name, function(x)
  seconds_table$count[match(x,seconds_table$Second_pokemon)])

pokemon_feats <- cbind(pokemon, win_counts, first_counts, second_counts)
pokemon_feats$losses <-
  pokemon_feats$first_counts + pokemon_feats$second_counts - pokemon_feats$win_counts
```

```r
pokemon_feats$win_ratio <-
  pokemon_feats$win_counts / (pokemon_feats$second_counts + pokemon_feats$first_counts)
head(pokemon_feats)
```

```
##   X.          Name Type.1 Type.2 HP Attack Defense Sp..Atk Sp..Def Speed
## 1  1     Bulbasaur  Grass Poison 45     49      49      65      65    45
## 2  2       Ivysaur  Grass Poison 60     62      63      80      80    60
## 3  3      Venusaur  Grass Poison 80     82      83     100     100    80
## 4  4 Mega Venusaur  Grass Poison 80    100     123     122     120    80
## 5  5    Charmander   Fire        39     52      43      60      50    65
## 6  6    Charmeleon   Fire        58     64      58      80      65    80
##   Generation Legendary win_counts first_counts second_counts losses
## 1          1     False         37           70            63     96
## 2          1     False         46           55            66     75
## 3          1     False         89           68            64     43
## 4          1     False         70           62            63     55
## 5          1     False         55           50            62     57
## 6          1     False         64           66            52     54
##   win_ratio
## 1 0.2781955
## 2 0.3801653
## 3 0.6742424
## 4 0.5600000
## 5 0.4910714
## 6 0.5423729
```

And finally we save this file as "features.csv"

```r
write.csv(pokemon_feats, file="./features.csv")
```

2.3 One Hot Encoding Categorical Data

Continuing with the dataset we created in section 2.2, we'll one hot encode categorical variables. We load the file and check again for any missing values.

```r
feats <- read.csv('./data/features.csv')
colSums(is.na(feats))
```

```
##             X            X.          Name        Type.1        Type.2
##             0             0             0             0             0
##            HP        Attack       Defense       Sp..Atk       Sp..Def
##             0             0             0             0             0
##         Speed    Generation     Legendary    win_counts  first_counts
##             0             0             0            17            16
## second_counts        losses     win_ratio
##            16            17            17
```

We know that not all the pokémon fought and that's why we have missing values. To proceed with modelling, we'll eliminate all the rows with missing values.

```r
feats2 <- feats[-(which(is.na(feats$win_ratio))),]
dim(feats2)
```

```
## [1] 783  18
```

```r
colnames(feats2)
```

```
##  [1] "X"               "X."              "Name"            "Type.1"
##  [5] "Type.2"          "HP"              "Attack"          "Defense"
##  [9] "Sp..Atk"         "Sp..Def"         "Speed"           "Generation"
## [13] "Legendary"       "win_counts"      "first_counts"    "second_counts"
## [17] "losses"          "win_ratio"
```

We also found a duplicate in the id column `X.`, but since we don't need the name of the pokemon for one hot encoding, we take them out of the dataset.

```r
feats2 <- feats2[,c(4:18)]; head(feats2);
```

```
##   Type.1 Type.2 HP Attack Defense Sp..Atk Sp..Def Speed Generation
## 1  Grass Poison 45     49      49      65      65    45          1
## 2  Grass Poison 60     62      63      80      80    60          1
## 3  Grass Poison 80     82      83     100     100    80          1
## 4  Grass Poison 80    100     123     122     120    80          1
## 5   Fire        39     52      43      60      50    65          1
## 6   Fire        58     64      58      80      65    80          1
##   Legendary win_counts first_counts second_counts losses win_ratio
## 1     False         37           70            63     96 0.2781955
## 2     False         46           55            66     75 0.3801653
## 3     False         89           68            64     43 0.6742424
## 4     False         70           62            63     55 0.5600000
## 5     False         55           50            62     57 0.4910714
## 6     False         64           66            52     54 0.5423729
```

```r
dim(feats2)
```

```
## [1] 783  15
```

Now we must distinguish between numerical and categorical features. To do this, we create an object with all features and their class. We then create another object that includes only the numerical features' `colnames` and the same for categorical features.

```r
feature_classes <- sapply(names(feats2),function(x){class(feats[[x]])})
feature_classes
```

```
##        Type.1        Type.2            HP        Attack       Defense
##      "factor"      "factor"     "integer"     "integer"     "integer"
##       Sp..Atk       Sp..Def         Speed    Generation     Legendary
##     "integer"     "integer"     "integer"     "integer"      "factor"
##    win_counts  first_counts second_counts        losses     win_ratio
##     "integer"     "integer"     "integer"     "integer"     "numeric"
```

```r
numeric_feats <-names(feats2[feature_classes != "character" &
                      feature_classes != "factor"])
numeric_feats
```

```
##  [1] "HP"            "Attack"        "Defense"       "Sp..Atk"
##  [5] "Sp..Def"       "Speed"         "Generation"    "win_counts"
##  [9] "first_counts"  "second_counts" "losses"        "win_ratio"
```

```r
categorical_feats <- names(feats2[feature_classes == "character" |
                            feature_classes == "factor"])
categorical_feats
```

```
## [1] "Type.1"    "Type.2"    "Legendary"
```

Finally we use the R dummyvars function to do one hot encode the categorical variables, which fills with zeros all fields with NaN values.

```
dummies <- dummyVars(~.,feats2[categorical_feats])
categorical_1_hot <- predict(dummies,feats2[categorical_feats])
categorical_1_hot[is.na(categorical_1_hot)] <- 0

head(dummies)
```

```
## $call
## dummyVars.default(formula = ~., data = feats2[categorical_feats])
##
## $form
## ~.
##
## $vars
## [1] "Type.1"    "Type.2"    "Legendary"
##
## $facVars
## [1] "Type.1"    "Type.2"    "Legendary"
##
## $lvls
## $lvls$Type.1
##  [1] "Bug"      "Dark"     "Dragon"   "Electric" "Fairy"    "Fighting"
##  [7] "Fire"     "Flying"   "Ghost"    "Grass"    "Ground"   "Ice"
## [13] "Normal"   "Poison"   "Psychic"  "Rock"     "Steel"    "Water"
##
## $lvls$Type.2
##  [1] ""         "Bug"      "Dark"     "Dragon"   "Electric" "Fairy"
##  [7] "Fighting" "Fire"     "Flying"   "Ghost"    "Grass"    "Ground"
## [13] "Ice"      "Normal"   "Poison"   "Psychic"  "Rock"     "Steel"
## [19] "Water"
##
## $lvls$Legendary
## [1] "False" "True"
##
##
## $sep
## [1] "."
```

```
head(categorical_1_hot)
```

```
##   Type.1.Bug Type.1.Dark Type.1.Dragon Type.1.Electric Type.1.Fairy
## 1          0           0             0               0            0
## 2          0           0             0               0            0
## 3          0           0             0               0            0
## 4          0           0             0               0            0
## 5          0           0             0               0            0
## 6          0           0             0               0            0
##   Type.1.Fighting Type.1.Fire Type.1.Flying Type.1.Ghost Type.1.Grass
## 1               0           0             0            0            1
## 2               0           0             0            0            1
## 3               0           0             0            0            1
## 4               0           0             0            0            1
## 5               0           1             0            0            0
```

```
## 6               0           1             0            0            0
##   Type.1.Ground Type.1.Ice Type.1.Normal Type.1.Poison Type.1.Psychic
## 1             0          0             0             0              0
## 2             0          0             0             0              0
## 3             0          0             0             0              0
## 4             0          0             0             0              0
## 5             0          0             0             0              0
## 6             0          0             0             0              0
##   Type.1.Rock Type.1.Steel Type.1.Water Type.2. Type.2.Bug Type.2.Dark
## 1           0            0            0       0          0           0
## 2           0            0            0       0          0           0
## 3           0            0            0       0          0           0
## 4           0            0            0       0          0           0
## 5           0            0            0       1          0           0
## 6           0            0            0       1          0           0
##   Type.2.Dragon Type.2.Electric Type.2.Fairy Type.2.Fighting Type.2.Fire
## 1             0               0            0               0           0
## 2             0               0            0               0           0
## 3             0               0            0               0           0
## 4             0               0            0               0           0
## 5             0               0            0               0           0
## 6             0               0            0               0           0
##   Type.2.Flying Type.2.Ghost Type.2.Grass Type.2.Ground Type.2.Ice
## 1             0            0            0             0          0
## 2             0            0            0             0          0
## 3             0            0            0             0          0
## 4             0            0            0             0          0
## 5             0            0            0             0          0
## 6             0            0            0             0          0
##   Type.2.Normal Type.2.Poison Type.2.Psychic Type.2.Rock Type.2.Steel
## 1             0             1              0           0            0
## 2             0             1              0           0            0
## 3             0             1              0           0            0
## 4             0             1              0           0            0
## 5             0             0              0           0            0
## 6             0             0              0           0            0
##   Type.2.Water Legendary.False Legendary.True
## 1            0               1              0
## 2            0               1              0
## 3            0               1              0
## 4            0               1              0
## 5            0               1              0
## 6            0               1              0
```

To finish the pre-processing step, we merge the numerical data with the categorical data after hot encoding in a data frame, and the result will be saved in a file for data modeling.

```
final_data <- cbind(feats2[numeric_feats],categorical_1_hot)
write.csv(final_data, file="./data/Model_data.csv")
```

3 Model

For our win ratio prediction, we tried out three different models, which we will discuss in the following. The predictor variables are the same for all three models, which are all the numerical feature values for each pokemon as well as its legendary status and how often it attacked first and how often it attacked second. We also tried to one hot encode the type of the pokemon and include this information into the model but it did not improve the model quality, so we dropped the type information.

```
# Load Data
feats <- read.csv('data/features.csv')
colSums(is.na(feats)) # some pokemon never fought
```

```
##             X           X.         Name       Type.1       Type.2
##             0            0            0            0            0
##            HP       Attack      Defense      Sp..Atk      Sp..Def
##             0            0            0            0            0
##         Speed   Generation    Legendary   win_counts first_counts
##             0            0            0           17           16
## second_counts       losses    win_ratio
##            16           17           17
```

```
# remove pokemons that never won
feats_clean <- na.omit(feats)
# convert category legendary into numerical values
feats_clean$Legendary <- as.integer(as.logical(feats_clean$Legendary))
# drop unnecessary generation column
feats_clean <- select(feats_clean, -c(Generation, win_counts, losses))
# typ <- as.integer(feats_clean$Type.1)
#split train test data
trainIndex = createDataPartition(feats_clean$win_ratio,
                                 p=0.6, list=FALSE, times=1)
train = feats_clean[trainIndex,] # 60% training data
test_val = feats_clean[-trainIndex,]
trainIndex = createDataPartition(test_val$win_ratio,
                                 p=0.5, list=FALSE, times=1)
val = test_val[trainIndex,] # 20% validation data
test = test_val[-trainIndex,] # 20% test data
# create Target variable
y_train <- train$win_ratio
y_val <- val$win_ratio
y_test <- test$win_ratio
x_train <- train[6:13]
x_val <- val[6:13]
x_test <- test[6:13]
```

3.1 Linear Regression Ridge

As the first model, we fitted the data into a Ridge regression model. The Ridge regression model is as well as the Lasso regression a shrinkage method. Instead of minimizing the Residual sum of squares (RSS) it minimizes the RSS combined with the coefficients shrunken by a lambda factor.

To find the best model we tried different values for lambda and chose the model with the smallest Root mean squared error (RMSE). The RMSE is the metric to evaluate the quality of a model. The smaller the error, the better the predicted values.

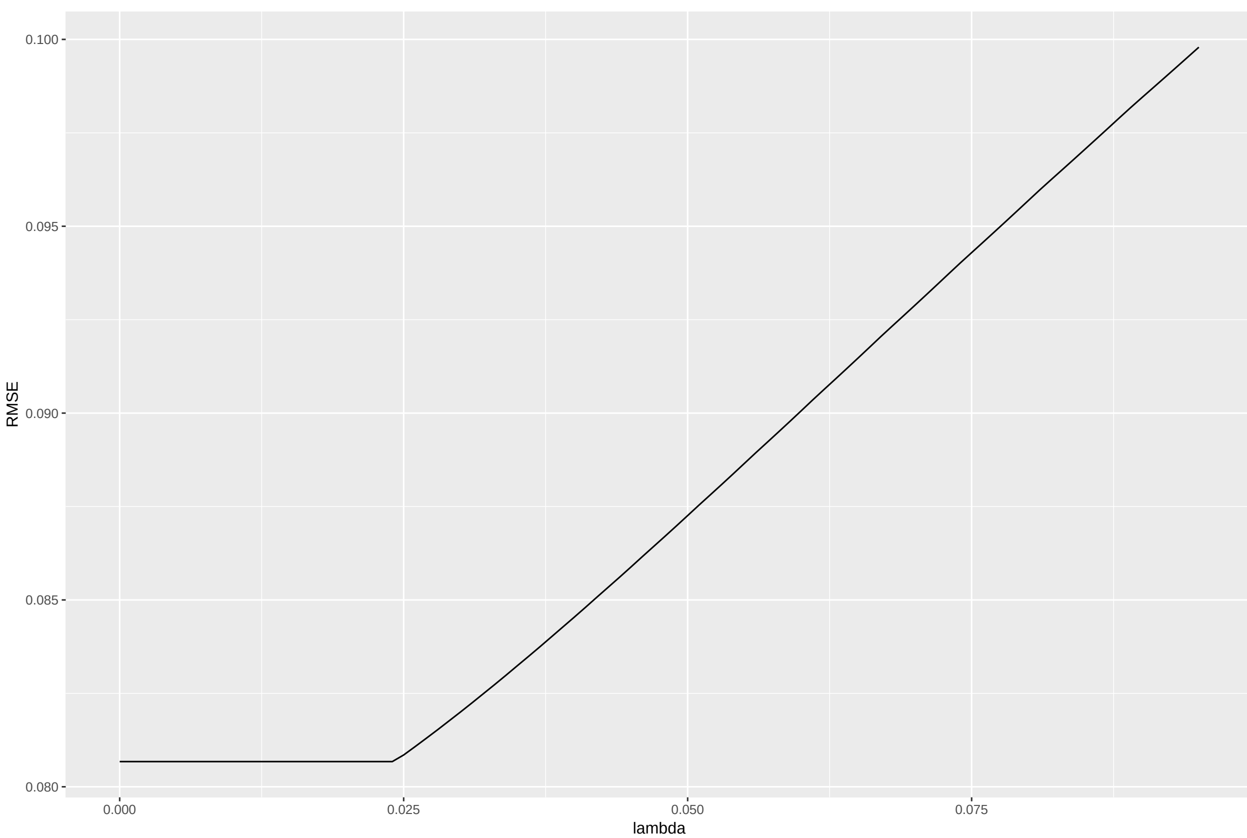
```
# Ridge regression
trainControl = trainControl(method="repeatedcv",
                            number=5,
                            repeats=5,
```

```
                                    verboseIter=FALSE)
lambdas <- seq(1,0,-0.001)
model_ridge <- train(x=x_train,y=y_train,
                     method="glmnet",
                     metric="RMSE",
                     maximize=FALSE,
                     trControl=trainControl,
                     tuneGrid=expand.grid(alpha=0, # Ridge regression
                                          lambda=lambdas))
# model_ridge
```

Results: The model with the best prediction quality is with lambda = 0.023. This model has a RMSE of around 0.08, which is pretty precise in its prediction.

This plot shows the relation between the different lambda values and the RMSE. As one can see in the the smaller the lambda the smaller RMSE.

```
# plot the effect of chosen lambda on the RMSE
ggplot(data=model_ridge$results[model_ridge$results$RMSE<=0.10,]) +
  geom_line(aes(x=lambda,y=RMSE))
```




```
# quality of model
cat("The best RMSE for this model is:", mean(model_ridge$resample$RMSE))
```

```
## The best RMSE for this model is: 0.08067419
```

We also tried to include the information about the types of the Pokemons and therefore, we used one hot encoding for that categorical variable. But that additional information did not improve the model quality. To keep the model sparse we decided to not use the variable.

3.2 Linear Regression Lasso

To compare our results to another regression model, we decided to fit the data to a Lasso regression model. Lasso regression is a variant of the Ridge regression. Lasso uses the absolute value if the coefficients in the penalty term and thus, can shrink the estimates coefficients to zero. This results in a variable selection and makes the model more sparse and easier to interpret.

```r
# Lasso regression
model_lasso <- train(x=x_train,y=y_train,
                     method="glmnet",
                     metric="RMSE",
                     maximize=FALSE,
                     trControl=trainControl,
                     tuneGrid=expand.grid(alpha=1,  # Lasso regression
                                          lambda=c(1,0.1,0.05,0.01,
                                                   seq(0.009,0.001,-0.001),
                                                   0.00075,0.0005,0.0001)))
```

```
## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info =
## trainInfo, : There were missing values in resampled performance measures.
```

```r
#model_lasso
```

Results: The best Lasso model has a lambda value of 0.003 and a RMSE slightly below the previous RMSE. So it seems that Lasso has a slightly better prediction quality than the Ridge model. Later we will discover which estimator variables were shrunken to zero and therefore not used.

```r
# quality of model
cat("The best RMSE for this model is:", mean(model_lasso$resample$RMSE))
```

```
## The best RMSE for this model is: 0.07778464
```

In the next step we validated the two regression models with our splitted validation data set.

```r
# quality of model
predLasso <- predict(model_lasso,newdata = x_val)
predRidge <- predict(model_ridge, newdata = x_val)
# function for calculating the metric
rmse <- function(actual, predicted)
{
  error <- actual - predicted
  sqrt(mean(error^2))
}
cat("The RMSE for the Lasso model is", rmse(y_val, predLasso),
    "and for the Ridge model", rmse(y_val, predRidge))
```

```
## The RMSE for the Lasso model is 0.09139764 and for the Ridge model 0.09198485
```

The results indicate that the models are not overfitted and can predict the win ratio of a certain Pokemon quite accurate. The Lasso model seems to perform better on this task.

Now we compute the RMSE metric with our test data to compare our models.

```r
# RMSE for test data
testLasso <- predict(model_lasso,newdata = x_test)
testRidge <- predict(model_ridge,newdata = x_test)
rmse1 <- rmse(y_test, testLasso)
cat("The RMSE for the test data is", rmse(y_test, testLasso))
```

```
## The RMSE for the test data is 0.07246947
```

```
rmse2 <- rmse(y_test, testRidge)
```

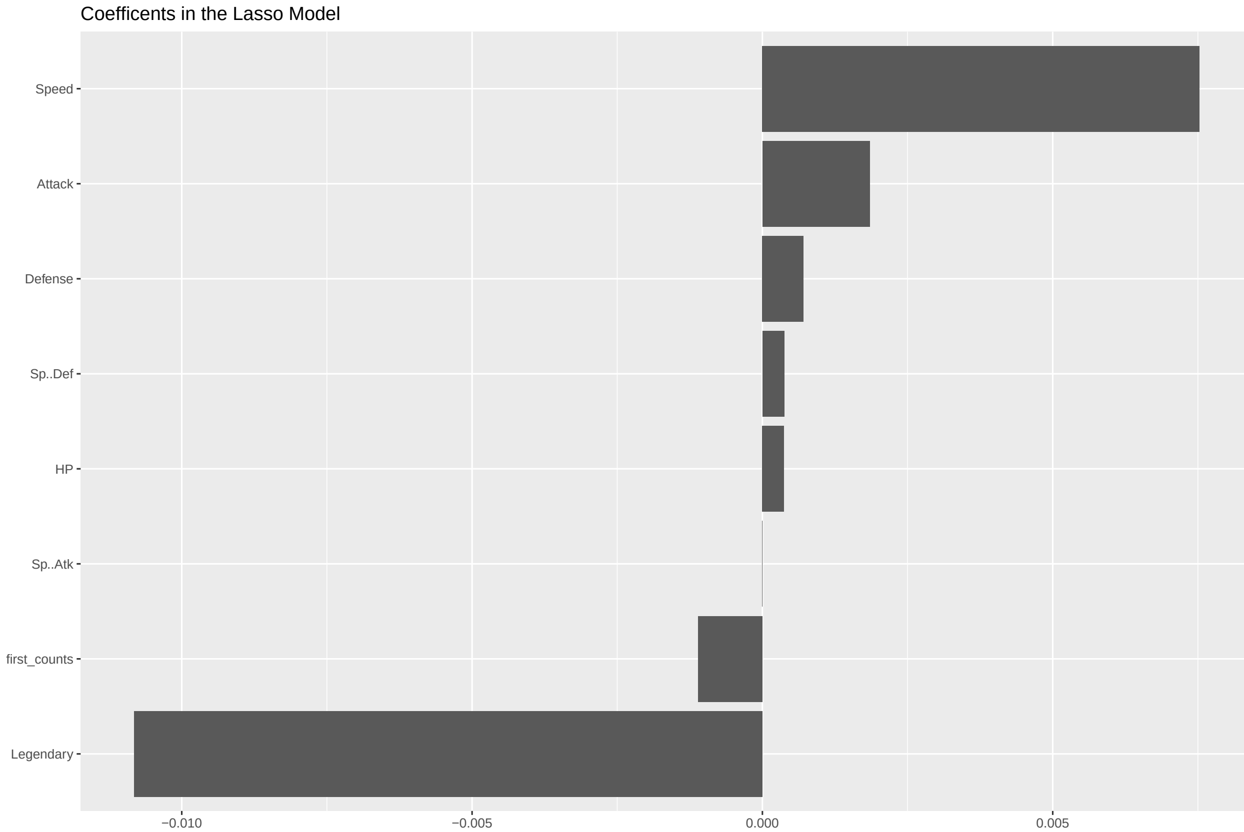
The next plot shows which estimate variables are more important for the prediction.

```
# quality of model
# extract coefficients for the best performing model
coef <- data.frame(coef.name = dimnames(
  coef(model_lasso$finalModel,s=model_lasso$bestTune$lambda))[[1]],
  coef.value = matrix(coef(model_lasso$finalModel,s=model_lasso$bestTune$lambda)))
# exclude the (Intercept) term
coef <- coef[-1,]
# print summary of model results
picked_features <- nrow(filter(coef,coef.value!=0))
not_picked_features <- nrow(filter(coef,coef.value==0))
cat("Lasso picked",picked_features,"variables and eliminated the other",
    not_picked_features,"variables\n")
```

```
## Lasso picked 7 variables and eliminated the other 1 variables
```

```
# sort coefficients in ascending order
coef <- arrange(coef,-coef.value)
# extract the top 10 and bottom 10 features
imp_coef <- rbind(head(coef,10),
                  tail(coef,10))
ggplot(imp_coef) +
  geom_bar(aes(x=reorder(coef.name,coef.value),y=coef.value),
           stat="identity") +
  ylim(min(imp_coef$coef.value),max(imp_coef$coef.value)) +
  coord_flip() +
  ggtitle("Coefficents in the Lasso Model") +
  theme(axis.title=element_blank())
```

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



3.3 Conclusions

Our conclusion for the regression part is that both models perform well in this task. The Lasso model performes slightly better than its Ridge counterpart. This may result from the fact that in the Lasso models some variables are eliminated through the shrinkage term.

Both models are not overfitted and can handle new data to predict the correct win ratio, as the RMSE of the fitted validation data is also small.

The Speed attribute of Pokemons is most important when it comes to winning a match. Attack and Defense seem also to play a bigger role, while the other estimate variables do not affect the outcome of a fight that much.

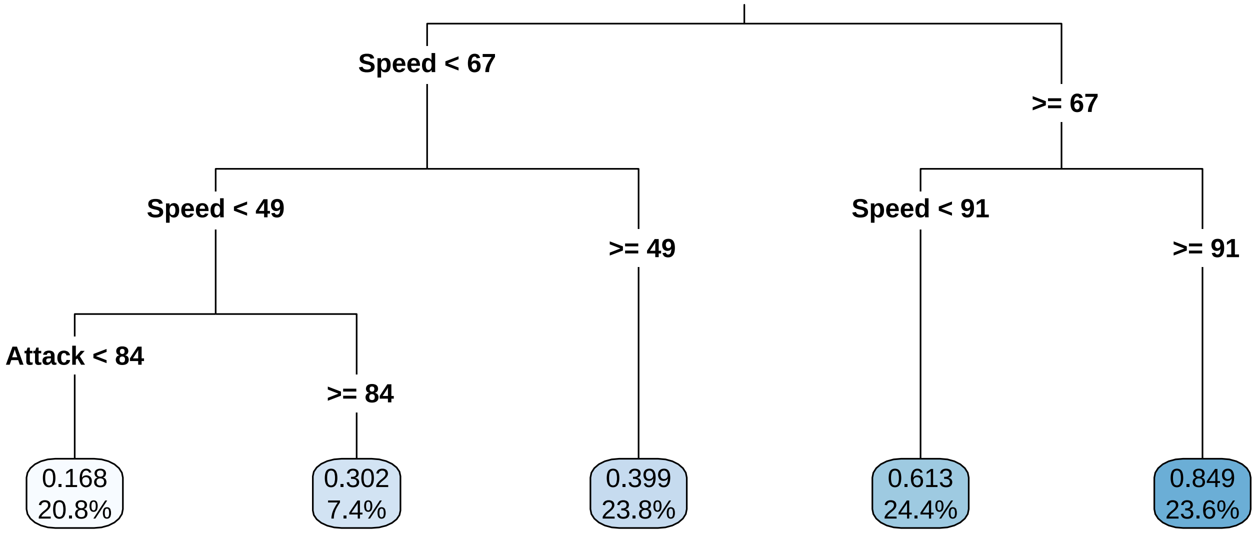
4 Model 2

In this chapture we discuss the second model of our project.

4.1 Decision Tree

We choosed a Decision Tree as a second model. The target variable for the tree is the win_ratio column. This column is a continious variable, so we need to use a regression tree:

```
tree <- rpart(y_train ~ . , data = x_train, method='anova')
rpart.plot(tree, type=3, digits=3, fallen.leaves = TRUE)
```



The rpart function takes many different arguments to improve the model:

rpart(formula, data, weights, subset, na.action = na.rpart, method, model = FALSE, x = FALSE, y = TRUE, parms, control, cost, etc)

The control argument itselfofferes a lot of different customization for the tree algorithm:

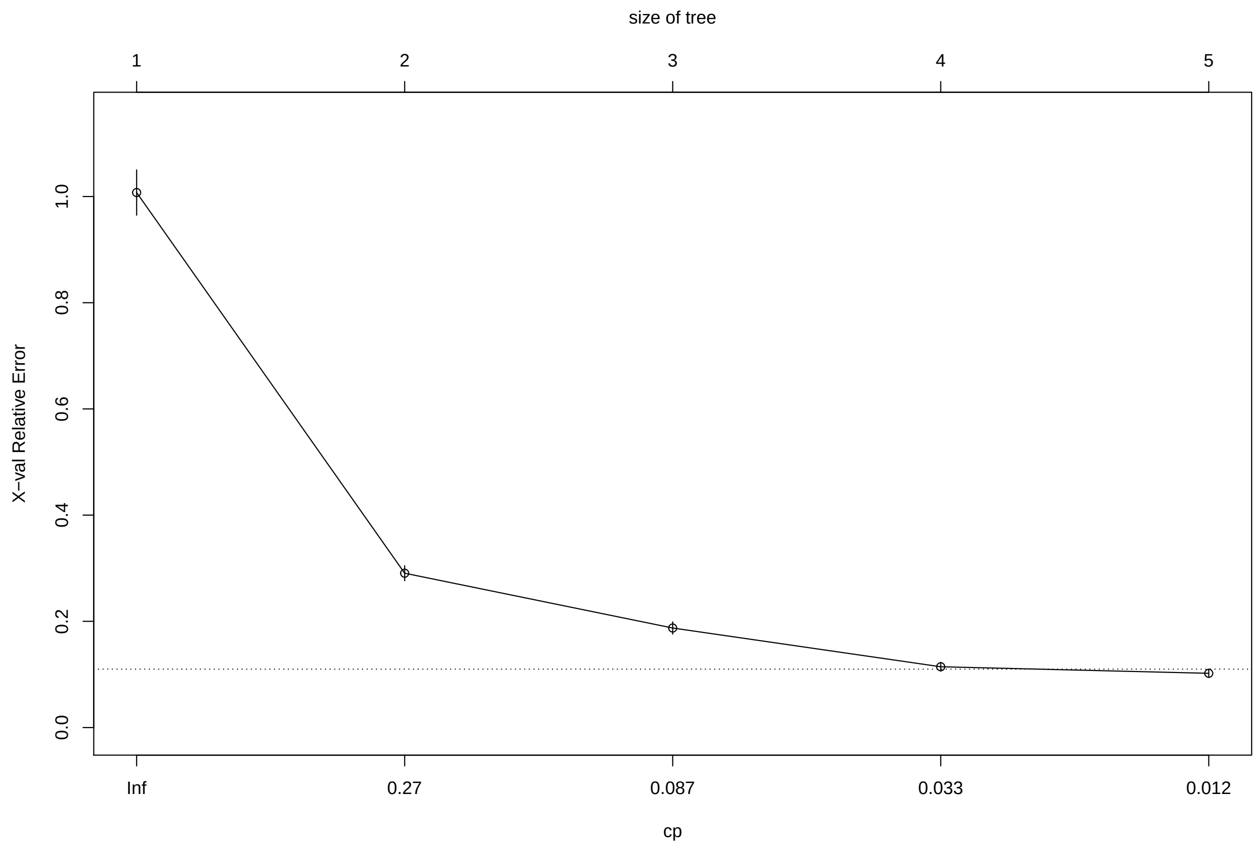
rpart.control(minsplit = 20, minbucket = round(minsplit/3), cp = 0.01, maxcompete = 4, maxsurrogate = 5, usesurrogate = 2, xval = 10, surrogatestyle = 0, maxdepth = 30, etc)

Because we have to run a regression tree, we have to use the method "anova" in the rpart function. It is possible to add directly the complexity parameter to prune the tree. It is the amount by which splitting a node improved the relative error.We made this step after training the model, because we need to find the best value for the complexity parameter.

```
# Find the best cp to prune the tree
tree$cptable
```

```
##           CP nsplit rel error    xerror        xstd
## 1 0.71853928      0 1.0000000 1.0074366 0.042624900
## 2 0.10149914      1 0.2814607 0.2906202 0.014218101
## 3 0.07508772      2 0.1799616 0.1874370 0.011520294
## 4 0.01493585      3 0.1048739 0.1144718 0.008244337
## 5 0.01000000      4 0.0899380 0.1021182 0.007854307
```

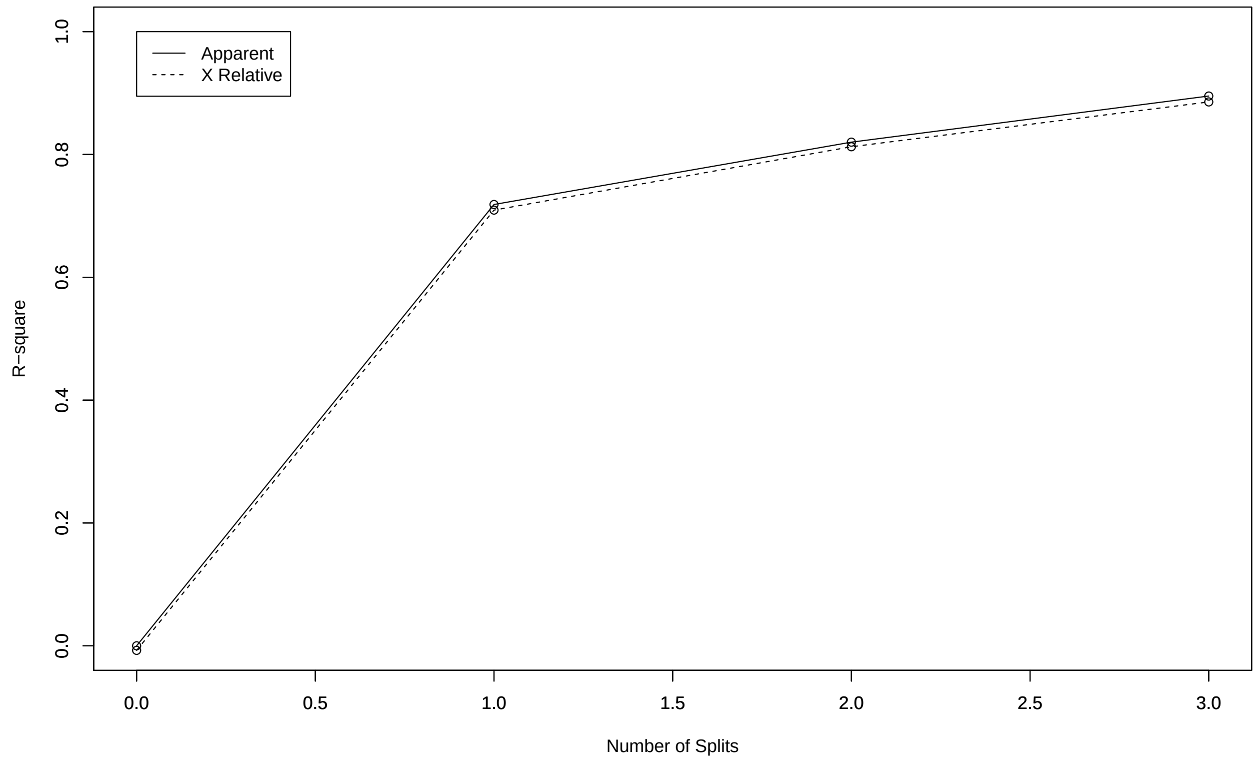
```
plotcp(tree)
```

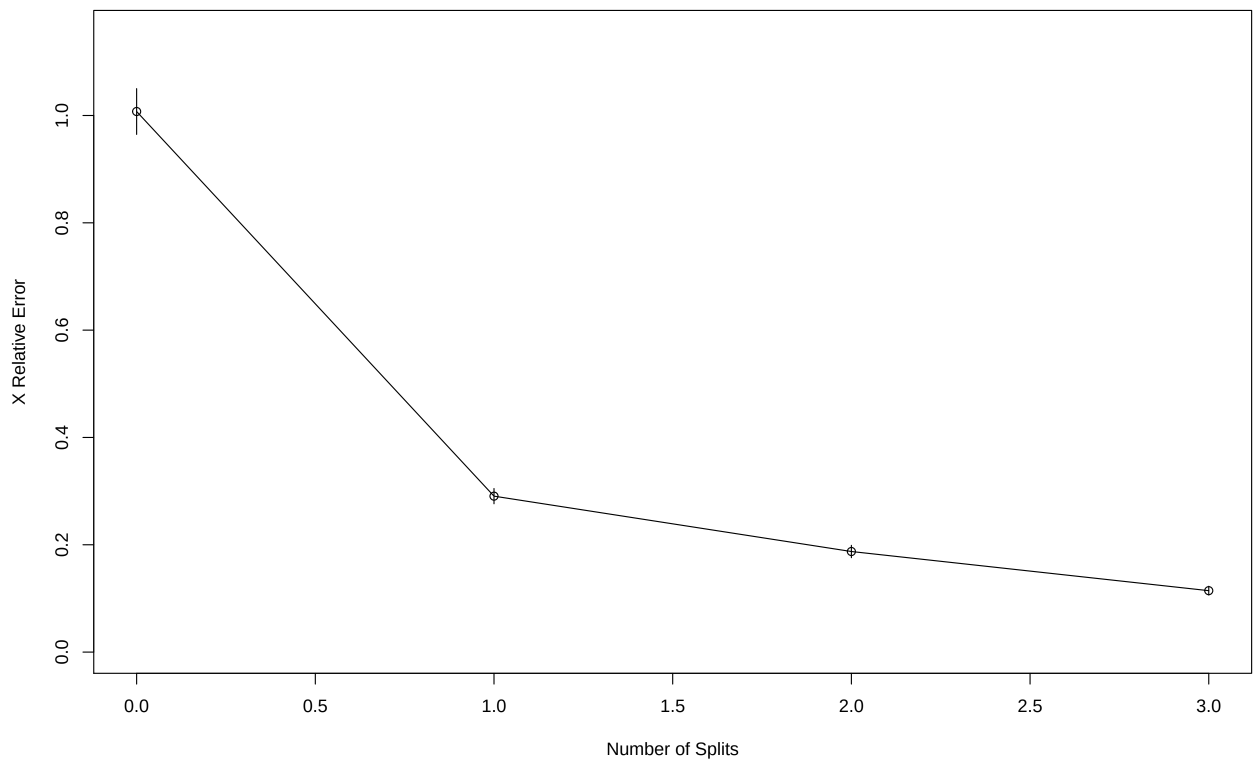


The plot shows us, that 0.039 is the best value for the complexity parameter. The next step is to prune the tree, to get a optimal presentation of the tree:

```
# Prune the tree to get a optimal decision tree
tree_pruned <- prune(tree,cp=0.039)
rsq.rpart(tree_pruned)
```

```
##
## Regression tree:
## rpart(formula = y_train ~ ., data = x_train, method = "anova")
##
## Variables actually used in tree construction:
## [1] Speed
##
## Root node error: 31.041/471 = 0.065905
##
## n= 471
##
##          CP nsplit rel error  xerror      xstd
## 1 0.718539      0   1.00000 1.00744 0.0426249
## 2 0.101499      1   0.28146 0.29062 0.0142181
## 3 0.075088      2   0.17996 0.18744 0.0115203
## 4 0.039000      3   0.10487 0.11447 0.0082443
```

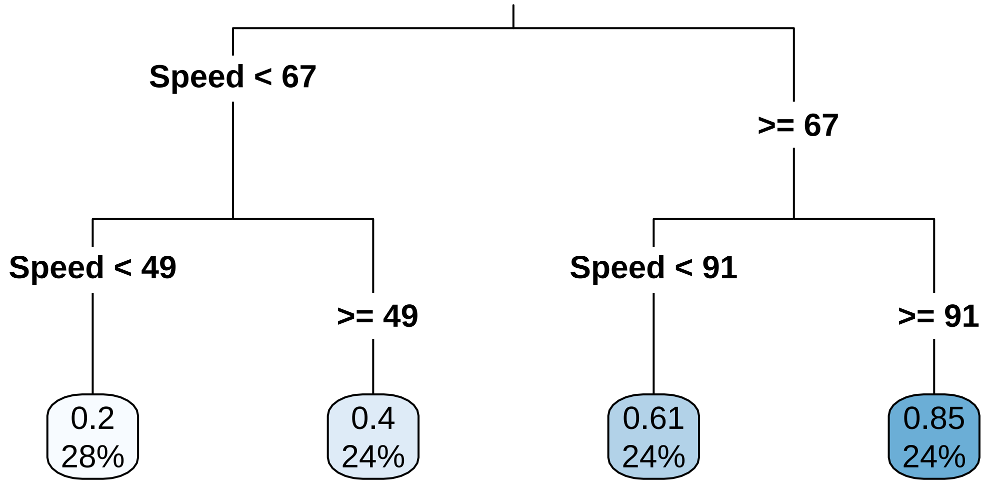




Now we can see, that the speed of a pokemon is the most important property to win a fight.

The final step is to plot the trained tree with type=3 to draw separate split labels for the left and right directions and fallen.leaves = TRUE to position the leaf nodes at the bottom of the graph:

```r
#Plot the final decision tree
rpart.plot(tree_pruned, type=3, fallen.leaves = TRUE)
```



Also for this model, we are using a function to calulate the RMSE:

```r
# Create Function to calculate Root Mean Square Error (RMSE)
rmse <- function(actual, predicted)
{
  error <- actual - predicted
  sqrt(mean(error^2))
}
```

The next step is to see, how accurate our model actually is. We predict the win_ratios for the test data:

```r
# Predict the 20% validation data to compare accuracy with the other model
p1<- predict(tree_pruned, x_val)
rmse(y_val, p1)
```

```
## [1] 0.1026113
```

Finally we predict and calculate the RMSE for the test data, to compare it with the other models:

```r
# Predict the 20% test data to compare accuracy with the other model
p2<- predict(tree_pruned, x_test)
rmse_tree <- rmse(y_test, p2)
```

4.2 Decision Tree Conclusion

The trained regression tree model performed very well in predicting the test and validation data. With the train data (60%) we got a RMSE of 0.07 - 0.088, which is an accurate prediction of the model. The basic model without a complexity parameter worked also very well and without overfitting. Nevertheless, we decided to use the complexity parameter to get a smaller tree plot with the most important splits. So, finally we have 4 splits instead of 7.

5 Conclusion

```r
cat("The RMSE for the Lasso model is", rmse1, "and for the Ridge model", rmse2)
```

```
## The RMSE for the Lasso model is 0.07246947 and for the Ridge model 0.0813534
```

```r
cat("The RMSE for the Regression tree model is", rmse_tree)
```

```
## The RMSE for the Regression tree model is 0.09594235
```

To conclude, we trained a ridge, lasso and regression tree model, to find the best working model for predicting the win_ratio for pokemons. By calculating RMSE, we are able to compare these models easily. Based on this value, we found out that lasso is the best performing model. The regression tree gives us slightly worse predictions for the win_ratio, but still very accurate. With that, we've decided to work with the lasso model.

Wikipedia. 2019. "Pokémon." Web page. https://en.wikipedia.org/wiki/Pokémon.