STATISTICAL LEARNING PROJECT

2024-07-20

NASA: Asteroids Classification

Filippin Giacomo, Kaci Flavio, Lovato Emma

Introduction

Near-Earth Objects (NEOs) are asteroids and comets with orbits that bring them into proximity with Earth. These objects are categorized based on their potential threat to Earth, as they can occasionally collide with our planet, causing significant damage.

The purpose of this project is to predict whether an asteroid is hazardous based on various features of the *nasa dataset* and to understand which characteristics are useful for this goal.

Dataset presentation

The dataset used in this analysis, obtained from Kaggle, comprises various attributes of asteroids, including their size, velocity, distance from Earth, and whether they are classified as hazardous. The dataset contains 4687 observations whose characteristics are described by 38 attributes.

The dataset includes the following features:

1. **Neo Reference ID**: A unique identifier assigned to each Near-Earth Object (NEO).
2. **Name**: The name or identifier assigned to the NEO.
3. **Absolute Magnitude**: A measure of the NEO's brightness, with lower values indicating brighter objects.
4. **Est Dia in KM (min)**: The estimated minimum diameter of the NEO in kilometers.
5. **Est Dia in KM (max)**: The estimated maximum diameter of the NEO in kilometers.
6. **Est Dia in M (min)**: The estimated minimum diameter of the NEO in meters.
7. **Est Dia in M (max)**: The estimated maximum diameter of the NEO in meters.
8. **Est Dia in Miles (min)**: The estimated minimum diameter of the NEO in miles.
9. **Est Dia in Miles (max)**: The estimated maximum diameter of the NEO in miles.
10. **Est Dia in Feet (min)**: The estimated minimum diameter of the NEO in feet.
11. **Est Dia in Feet (max)**: The estimated maximum diameter of the NEO in feet.
12. **Relative Velocity km per sec**: The relative velocity of the NEO with respect to Earth, measured in kilometers per second.
13. **Relative Velocity km per hr**: The relative velocity of the NEO with respect to Earth, measured in kilometers per hour.
14. **Miles per hour**: The relative velocity of the NEO with respect to Earth, measured in miles per hour.
15. **Miss Dist (Astronomical) Units**: The closest distance by which the NEO will pass Earth, measured in astronomical units (AU).
16. **Miss Dist (lunar)**: The closest distance by which the NEO will pass Earth, measured in lunar distances.

17. **Miss Dist (kilometers)**: The closest distance by which the NEO will pass Earth, measured in kilometers.
18. **Miss Dist (miles)**: The closest distance by which the NEO will pass Earth, measured in miles.
19. **Orbiting Body**: The celestial body around which the NEO orbits.
20. **Orbit ID**: A unique identifier assigned to the NEO's orbit.
21. **Orbit Determination Date**: The date on which the orbit determination parameters were calculated.
22. **Orbit Uncertainity**: A measure of the uncertainty in the NEO's orbit, with lower values indicating higher precision.
23. **Minimum Orbit Intersection**: The minimum distance between the NEO's orbit and Earth's orbit.
24. **Jupiter Tisserand Invariant**: A parameter used to categorize small bodies in the solar system in relation to Jupiter's orbit.
25. **Epoch Osculation**: The moment in time at which the NEO's orbital parameters were calculated.
26. **Eccentricity**: A measure of the deviation of the NEO's orbit from a perfect circle.
27. **Semi Major Axis**: The longest diameter of the NEO's elliptical orbit.
28. **Inclination**: The tilt of the NEO's orbit relative to the plane of the ecliptic.
29. **Asc Node Longitude**: The longitude of the ascending node of the NEO's orbit.
30. **Orbital Period**: The time it takes for the NEO to complete one orbit around its primary body.
31. **Perihelion Distance**: The closest distance between the NEO and the Sun.
32. **Perihelion Arg**: The argument of perihelion, describing the orientation of the NEO's orbit within its orbital plane.
33. **Aphelion Dist**: The farthest distance between the NEO and the Sun.
34. **Perihelion Time**: The time at which the NEO is closest to the Sun in its orbit.
35. **Mean Anomaly**: The fraction of an orbital period that has elapsed since the NEO passed perihelion, expressed in degrees.
36. **Mean Motion**: The rate at which the NEO moves along its orbit.
37. **Equinox**: The equinox reference for the orbital elements.
38. **Hazardous**: A binary indicator (True/False) of whether the NEO is classified as potentially hazardous to Earth.

These features provide a comprehensive view of each NEO's physical characteristics and orbital parameters, which are essential for predicting their potential hazard to Earth

Data preparation and cleaning

We start by loading the necessary libraries and the dataset.

Data Uploading The dataset is loaded and cleaned to ensure an effective analysis .

```
# We upload the dataset
data <- read.csv("nasa.csv")
```

Data Pre-Processing Before proceeding with the analysis we explore the structure of the dataset. The first thing we do is to check the presence of missing values in the dataset:

```
# Remove NA
sum(is.na(data))
```

```
## [1] 0
```

We don't have any missing values, so we can proceed. We consider now the features described in the dataset that represent the characteristics for each unit. We have noticed that some variables are redundant, as they

are presented for different units of measure; some features are simple identifiers and some features do not provide useful information for our purposes. So, we proceed removing the unnecessary features:

```
# We clean up the dataset by removing redundant variables
data <- data[, !names(data) %in% c("Close.Approach.Date",
"Orbit.Determination.Date","Equinox", "Orbit.ID", "Orbiting.Body",
"Relative.Velocity.km.per.hr", "Miss.Dist..lunar.", "Miss.Dist..Astronomical.",
"Miss.Dist..miles.", "Neo.Reference.ID", "Name", "Est.Dia.in.Miles.min.",
"Est.Dia.in.Miles.max.", "Est.Dia.in.Feet.min.", "Est.Dia.in.Feet.max.",
"Est.Dia.in.KM.min.", "Est.Dia.in.KM.max.", "Miles.per.hour")]
```

Then, we check the type of our features. We look at their nature and we codify them in a correct and meaningful way and the *hazardous* variable is converted to a factor to represent whether an asteroid is hazardous (1) or not (0).

```
# Type of variables
str(data)
```

```
## 'data.frame':    4687 obs. of  22 variables:
##  $ Absolute.Magnitude          : num  21.6 21.3 20.3 27.4 21.6 19.6 19.6 19.2 17.8 21.5 ...
##  $ Est.Dia.in.M.min.           : num  127.2 146.1 231.5 8.8 127.2 ...
##  $ Est.Dia.in.M.max.           : num  284.5 326.6 517.7 19.7 284.5 ...
##  $ Epoch.Date.Close.Approach   : num  7.89e+11 7.89e+11 7.90e+11 7.90e+11 7.90e+11 ...
##  $ Relative.Velocity.km.per.sec: num  6.12 18.11 7.59 11.17 9.84 ...
##  $ Miss.Dist..kilometers.      : num  62753692 57298148 7622912 42683616 61010824 ...
##  $ Orbit.Uncertainity          : int  5 3 0 6 1 1 1 0 0 0 ...
##  $ Minimum.Orbit.Intersection  : num  0.02528 0.18693 0.04306 0.00551 0.0348 ...
##  $ Jupiter.Tisserand.Invariant : num  4.63 5.46 4.56 5.09 5.15 ...
##  $ Epoch.Osculation            : num  2458001 2458001 2458001 2458001 2458001 ...
##  $ Eccentricity                : num  0.426 0.352 0.348 0.217 0.21 ...
##  $ Semi.Major.Axis             : num  1.41 1.11 1.46 1.26 1.23 ...
##  $ Inclination                 : num  6.03 28.41 4.24 7.91 16.79 ...
##  $ Asc.Node.Longitude          : num  314.4 136.7 259.5 57.2 84.6 ...
##  $ Orbital.Period              : num  610 426 644 514 496 ...
##  $ Perihelion.Distance         : num  0.808 0.718 0.951 0.984 0.968 ...
##  $ Perihelion.Arg              : num  57.3 313.1 248.4 18.7 158.3 ...
##  $ Aphelion.Dist               : num  2.01 1.5 1.97 1.53 1.48 ...
##  $ Perihelion.Time             : num  2458162 2457795 2458120 2457902 2457814 ...
##  $ Mean.Anomaly                : num  264.8 173.7 292.9 68.7 135.1 ...
##  $ Mean.Motion                 : num  0.591 0.845 0.559 0.7 0.726 ...
##  $ Hazardous                   : chr  "True" "False" "True" "False" ...
```

```
# Modify the "Hazardous" variable to binary
data$Hazardous <- ifelse(data$Hazardous == "True", 1, 0)
# Change the nature of the features
data$Orbit.Uncertainity = as.numeric(data$Orbit.Uncertainity)
data$Hazardous = as.factor(data$Hazardous)
# Check the size of the cleaned dataset
dim(data)
```

```
## [1] 4687   22
```

```r
n=nrow(data)
```

Exploratory Data Analysis

Before training the model, we perform exploratory data analysis (EDA) to understand the distribution of the features and their relationships with the target variable.

Univariate exploratory analysis We visualize the standard deviation of the numerical variables.

```r
# Calculate the standard deviation for all numerical variables
numeric_vars <- sapply(data, is.numeric)
(std_devs <- sapply(data[, numeric_vars], sd))
```

```
##           Absolute.Magnitude            Est.Dia.in.M.min.
##                 2.890972e+00                 3.695734e+02
##            Est.Dia.in.M.max.    Epoch.Date.Close.Approach
##                 8.263912e+02                 1.981540e+11
## Relative.Velocity.km.per.sec       Miss.Dist..kilometers.
##                 7.293223e+00                 2.181110e+07
##           Orbit.Uncertainity   Minimum.Orbit.Intersection
##                 3.078307e+00                 9.029997e-02
##  Jupiter.Tisserand.Invariant             Epoch.Osculation
##                 1.237818e+00                 9.202975e+02
##                 Eccentricity              Semi.Major.Axis
##                 1.804438e-01                 5.241539e-01
##                  Inclination          Asc.Node.Longitude
##                 1.093623e+01                 1.032768e+02
##                Orbital.Period         Perihelion.Distance
##                  3.709547e+02                 2.420591e-01
##               Perihelion.Arg               Aphelion.Dist
##                  1.035130e+02                 9.515195e-01
##              Perihelion.Time                 Mean.Anomaly
##                  9.442264e+02                 1.075016e+02
##                   Mean.Motion
##                  3.426271e-01
```

The distribution of the target variable ***hazardous*** is visualized.

```r
# HAZARDOUS: True = 1 and False = 0
table(data$Hazardous)
```

```
##
##    0    1
## 3932  755
```

```r
round(table(data$Hazardous)/n, 5)
```

```
##
##       0       1
## 0.83892 0.16108
```

The Shapiro-Wilk test is used to test the null hypothesis that the data is normally distributed for each variable.

```
# NORMALITY CHECK
shapiro_results <- c(
  "Absolute.Magnitude" = shapiro.test(data$Absolute.Magnitude)$p.value,
  "Est.Dia.in.M.min." = shapiro.test(data$Est.Dia.in.M.min.)$p.value,
  "Est.Dia.in.M.max." = shapiro.test(data$Est.Dia.in.M.max.)$p.value,
  "Epoch.Date.Close.Approach" = shapiro.test(data$Epoch.Date.Close.Approach)$p.value,
  "Relative.Velocity.km.per.sec" = shapiro.test(data$Relative.Velocity.km.per.sec)$p.value,
  "Miss.Dist..kilometers" = shapiro.test(data$Miss.Dist..kilometers)$p.value,
  "Orbit.Uncertainity" = shapiro.test(data$Orbit.Uncertainity)$p.value,
  "Minimum.Orbit.Intersection" = shapiro.test(data$Minimum.Orbit.Intersection)$p.value,
  "Jupiter.Tisserand.Invariant" = shapiro.test(data$Jupiter.Tisserand.Invariant)$p.value,
  "Epoch.Osculation" = shapiro.test(data$Epoch.Osculation)$p.value,
  "Eccentricity" = shapiro.test(data$Eccentricity)$p.value,
  "Semi.Major.Axis" = shapiro.test(data$Semi.Major.Axis)$p.value,
  "Inclination" = shapiro.test(data$Inclination)$p.value,
  "Asc.Node.Longitude" = shapiro.test(data$Asc.Node.Longitude)$p.value,
  "Orbital.Period" = shapiro.test(data$Orbital.Period)$p.value,
  "Perihelion.Distance" = shapiro.test(data$Perihelion.Distance)$p.value,
  "Perihelion.Arg" = shapiro.test(data$Perihelion.Arg)$p.value,
  "Aphelion.Dist" = shapiro.test(data$Aphelion.Dist)$p.value,
  "Perihelion.Time" = shapiro.test(data$Perihelion.Time)$p.value,
  "Mean.Anomaly" = shapiro.test(data$Mean.Anomaly)$p.value,
  "Mean.Motion" = shapiro.test(data$Mean.Motion)$p.value
)

# Results table
results_df <- data.frame(
  Variable = names(shapiro_results),
  P_value = shapiro_results,
  Normality = ifelse(shapiro_results < 0.05,
                     "Not normal", "Normal"),
  stringsAsFactors = FALSE)
```

```
## | Variable | P_value | Normality |

## |----------|---------|-----------|

## | Absolute.Magnitude | 0 | Not normal |
## | Est.Dia.in.M.min. | 0 | Not normal |
## | Est.Dia.in.M.max. | 0 | Not normal |
## | Epoch.Date.Close.Approach | 0 | Not normal |
## | Relative.Velocity.km.per.sec | 0 | Not normal |
## | Miss.Dist..kilometers | 0 | Not normal |
## | Orbit.Uncertainity | 0 | Not normal |
## | Minimum.Orbit.Intersection | 0 | Not normal |
## | Jupiter.Tisserand.Invariant | 0 | Not normal |
## | Epoch.Osculation | 0 | Not normal |
## | Eccentricity | 0 | Not normal |
## | Semi.Major.Axis | 0 | Not normal |
## | Inclination | 0 | Not normal |
```

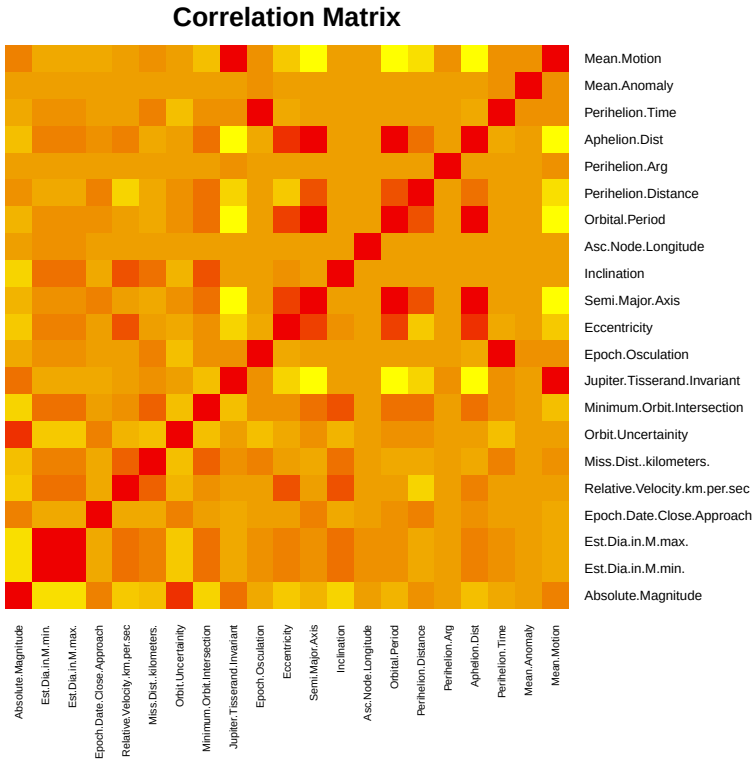
```
## | Asc.Node.Longitude | 0 | Not normal |
## | Orbital.Period | 0 | Not normal |
## | Perihelion.Distance | 0 | Not normal |
## | Perihelion.Arg | 0 | Not normal |
## | Aphelion.Dist | 0 | Not normal |
## | Perihelion.Time | 0 | Not normal |
## | Mean.Anomaly | 0 | Not normal |
## | Mean.Motion | 0 | Not normal |
```

All the continuous variables show p-values less than 0.05, leading to the rejection of the null hypothesis for each variable. This suggests that none of the variables follow a normal distribution.

Then we plot the distribution of each numerical variable in the dataset using histograms. The histograms provide a visual representation of the distribution of each numerical variable.

Bivariate exploratory analysis We exam the relationships between numerical variables in the dataset and address multicollinearity issues to ensure robust model performance. We have calculated the correlation matrix for all numerical variables using the Pearson correlation coefficient and then visualized it using a heatmap to identify patterns and strengths of correlations between variables.

```r
colors <- colorRampPalette(c("yellow", "orange2", "red2"))(20)
par(mfrow = c(1, 1))
par(mar = c(5, 4, 4, 2) + 0.1)
par(cex = 0.5, cex.main = 0.7, cex.lab = 0.5, cex.axis = 0.4)
# CORRELATION ANALYSIS
concomitant_vars <- data[, names(data) != "Hazardous"]
cor_matrix <- cor(concomitant_vars, use = "complete.obs")
heatmap(cor_matrix, main = "Correlation Matrix", Rowv = NA,
        Colv = NA, cexRow=0.5, cexCol=0.4, col = colors,
        symm = TRUE)
```




```
# Collinearity addressing:
# Correlation matrix:
cor_matrix <- cor(data[, names(data) != "Hazardous"])
# Function to find highly correlated variables
find_high_correlation <- function(cor_matrix, cutoff) {
  highly_correlated <- c()
  for (i in 1:(ncol(cor_matrix) - 1)) {
    for (j in (i + 1):ncol(cor_matrix)) {
      if (abs(cor_matrix[i, j]) > cutoff) {
        if (!i %in% highly_correlated & !j %in%
            highly_correlated) {
          highly_correlated <- c(highly_correlated, j)}}}}
  return(highly_correlated)}
# Calculate highly correlated variables
cutoff <- 0.9
highly_correlated <- find_high_correlation(cor_matrix, cutoff)
highly_correlated_vars <- names(data)[highly_correlated]
print(highly_correlated_vars)
```

```
## [1] "Est.Dia.in.M.max." "Semi.Major.Axis"   "Mean.Motion"
## [4] "Perihelion.Time"   "Aphelion.Dist"
```

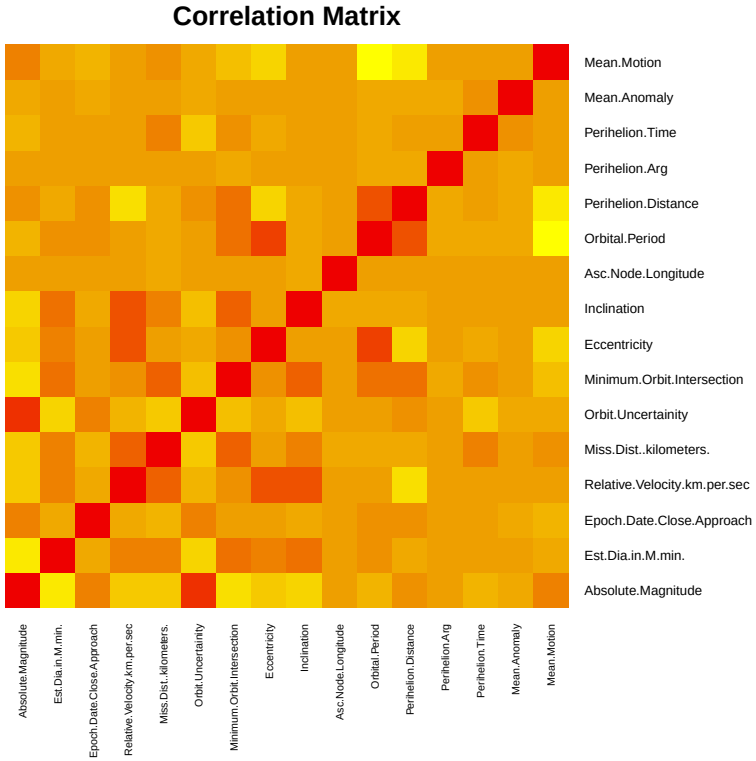
```
highly_correlated_matrix <- cor_matrix[highly_correlated, highly_correlated]
print(highly_correlated_matrix)
```

```
##                   Est.Dia.in.M.max. Semi.Major.Axis Mean.Motion Perihelion.Time
## Est.Dia.in.M.max.         1.0000000      0.12122353 -0.10435013      0.06216750
## Semi.Major.Axis           0.1212235      1.00000000 -0.90139590     -0.05930282
```

```
## Mean.Motion              -0.1043501     -0.90139590  1.00000000      0.04703532
## Perihelion.Time           0.0621675     -0.05930282  0.04703532      1.00000000
## Aphelion.Dist             0.1518364      0.97532564 -0.84016585     -0.06460914
##                    Aphelion.Dist
## Est.Dia.in.M.max.     0.15183637
## Semi.Major.Axis       0.97532564
## Mean.Motion          -0.84016585
## Perihelion.Time      -0.06460914
## Aphelion.Dist         1.00000000
```

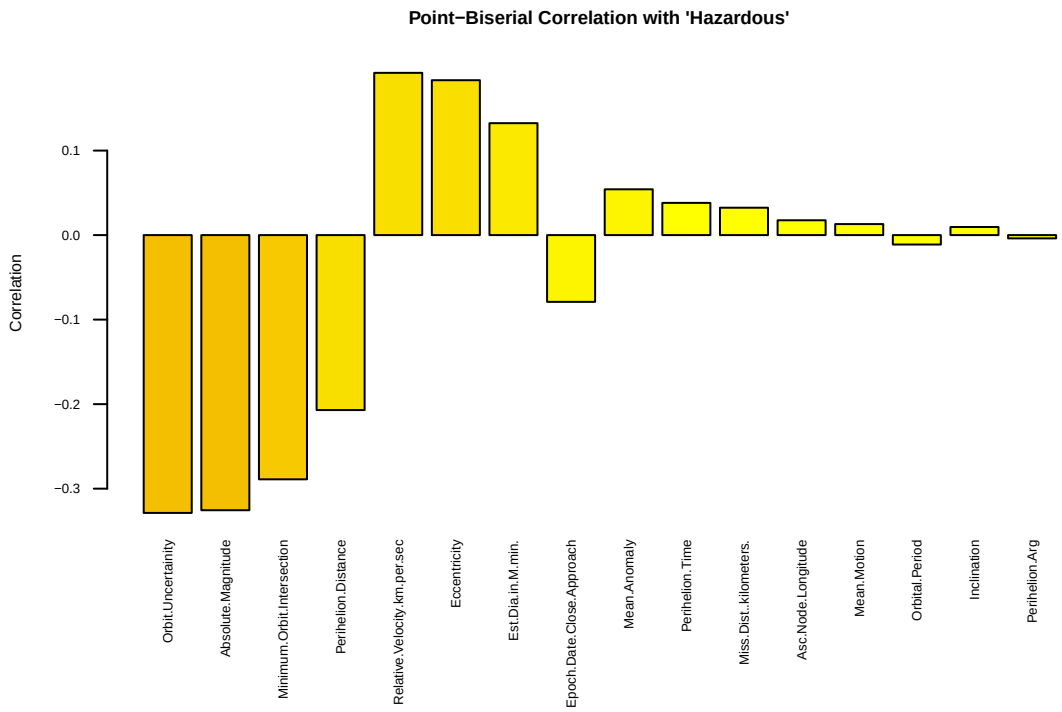
```r
# Remove highly correlated variables from training and validation data
data <- data[, !(names(data) %in% c("Jupiter.Tisserand.Invariant",
                                    "Aphelion.Dist", "Est.Dia.in.M.max.",
                                    "Epoch.Osculation", "Semi.Major.Axis",
                                    "Jupiter.Tisserand.Invariant"))]
concomitant_vars <- data[, names(data) != "Hazardous"]
par(mfrow = c(1, 1))
par(mar = c(5, 4, 4, 2) + 0.1)
par(cex = 0.5, cex.main = 0.7, cex.lab = 0.5, cex.axis = 0.4)

cor_matrix <- cor(concomitant_vars, use = "complete.obs")
heatmap(cor_matrix, main = "Correlation Matrix", Rowv = NA, Colv = NA,
        cexRow=0.5, cexCol=0.4, col = colors, symm = TRUE)
```




```r
# Function to calculate point-biserial correlation
point_biserial_corr <- function(numeric_var, binary_var) {
  binary_var <- as.numeric(binary_var) - 1
  cor(numeric_var, binary_var)
```

```r
}
concomitant_vars <- data[, names(data) != "Hazardous"]
# Calculate point-biserial correlation
# for all numerical variables with respect to 'Hazardous'
correlations <- sapply(concomitant_vars,
                       function(x) point_biserial_corr(x, data$Hazardous))
# Convert correlations to a dataframe
correlations_df <- data.frame(Variable = names(correlations),
                              Correlation = correlations)
# Calculate absolute values of correlations
correlations_df$AbsCorrelation <- abs(correlations_df$Correlation)
# Sort correlations dataframe
# by absolute correlation values
# for better color gradient in barplot
correlations_df <- correlations_df[order(correlations_df$AbsCorrelation,
                                         decreasing = TRUE), ]
# Adjust margins and text size for barplot
par(mar = c(8, 4, 4, 2) + 0.1)
par(cex = 0.7, cex.main = 0.8, cex.lab = 0.7, cex.axis = 0.6)
# Create barplot with colors based on absolute correlation values
barplot(correlations_df$Correlation, names.arg = correlations_df$Variable,
        las = 2, col = colors[findInterval(correlations_df$AbsCorrelation,
                                           seq(0, 1, length.out = 20))],
        main = "Point-Biserial Correlation with 'Hazardous'",
        ylab = "Correlation")
```



The initial heatmap shows varying degrees of correlation between numerical variables, with some pairs exhibiting very high correlations (close to ± 1). Variables such as *Jupiter.Tisserand.Invariant* and *Aphelion.Dist* showed high correlations with other variables, indicating potential multicollinearity. So

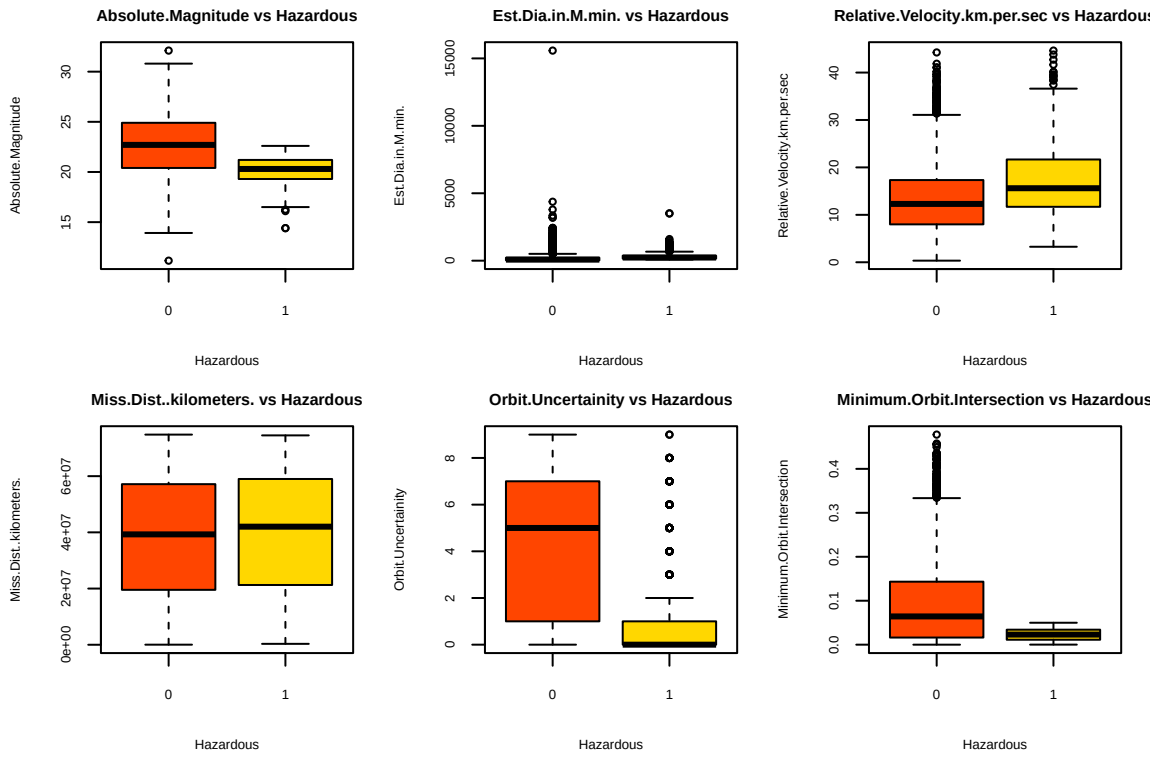
we remove the following highly correlated variables: *Jupiter.Tisserand.Invariant*, *Aphelion.Dist*, *Est.Dia.in.M.max.*, *Epoch.Osculation*, *Semi.Major.Axis*.

After removing the highly correlated variables, the new heatmap showed a more balanced distribution of correlations, indicating reduced multicollinearity.

Finally, the bar plot of point-biserial correlations revealed the relationship between each numerical variable and the binary target variable *Hazardous*. Variables such as *Absolute.Magnitude* and *Perihelion.Distance* showed notable correlations with *Hazardous*.

Then we explore the relationships between numerical variables and the binary target variable *Hazardous* using box plots. This helps in understanding the distribution of these variables for hazardous and non-hazardous cases.

```
# VARIABLES IN RELATION TO HAZARDOUS
# Select variables that are apparently relevant
# for analysis to further explore relationships with hazardous
relevant_variables <- c("Absolute.Magnitude", "Est.Dia.in.M.min.",
                        "Relative.Velocity.km.per.sec",
                        "Miss.Dist..kilometers.", "Orbit.Uncertainity",
                        "Minimum.Orbit.Intersection")
# Set layout for multiple plots
par(mfrow = c(2, 3), mar = c(4, 4, 2, 1) + 0.1)
# Adjust text sizes
par(cex.main = 0.7, cex.lab = 0.6, cex.axis = 0.6)
# Create boxplots
for (var in relevant_variables) {
  boxplot(data[[var]] ~ data$Hazardous,
          main = paste(var, "vs Hazardous"),
          xlab = "Hazardous", ylab = var,
          col = c("orangered", "gold"))}
```



```r
# Remove outlier
max_non_hazardous <- which.max(data$Est.Dia.in.M.min.)
data <- data[-max_non_hazardous, ]
```

Standardization Each numerical variable in the dataset was transformed to have a mean of 0 and a standard deviation of 1. This ensures that all variables are on a comparable scale.

```r
# Standardizing numerical variables
data_std <- data[, names(data) != "Hazardous"]
standardize <- function(x) {
  return((x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE))}
data_std <- as.data.frame(lapply(data_std, standardize))
data_std$Hazardous <- data$Hazardous
```

Splitting in train, test and validation sets

The dataset is split into training/validation (75%) and test (25%) sets using random sampling with a seed for reproducibility. Furthermore, the training/validation set is split into training (2/3) and validation (1/3) sets.

Oversampling and Undersampling:

- **Oversampling**: Duplicates minority class samples until both classes are equally represented.
- **Undersampling**: Reduces the majority class samples until both classes are equally represented.

Applied separately to the training, validation, and combined training/validation sets.

```r
data = data_std
set.seed(123) # Set seed
n = NROW(data) # Dataset dimensions
# TRAINVAL and TEST
trainval_index <- sample(1:n, n * 0.75)  # 75% per TRAINVAL
test_index <- setdiff(1:n, trainval_index)  # 25% per TEST
trainvalData <- data[trainval_index, ]
testData <- data[test_index, ]
# TRAINVAL in TRAIN and VALIDATION
n_trainval = NROW(trainvalData)
train_index <- sample(1:n_trainval, n_trainval * 2/3)
# 2/3 di TRAINVAL per TRAIN
validation_index <- setdiff(1:n_trainval, train_index)
# 1/3 di TRAINVAL per VALIDATION
trainData <- trainvalData[train_index, ]
valData <- trainvalData[validation_index, ]
# Oversampling
trainvalData_os <- ovun.sample(Hazardous ~ .,
                               data = trainvalData, method = "over",
                               N = 2 * max(table(trainvalData$Hazardous)))$data
trainData_os <- ovun.sample(Hazardous ~ .,
                            data = trainData, method = "over",
                            N = 2 * max(table(trainData$Hazardous)))$data
valData_os <- ovun.sample(Hazardous ~ ., data = valData, method = "over",
```

```
                          N = 2 * max(table(valData$Hazardous)))$data
# Undersampling
trainvalData_us <- ovun.sample(Hazardous ~ ., data = trainvalData, method = "under",
                               N = 2 * min(table(trainvalData$Hazardous)))$data
trainData_us <- ovun.sample(Hazardous ~ ., data = trainData, method = "under",
                            N = 2 * min(table(trainData$Hazardous)))$data
valData_us <- ovun.sample(Hazardous ~ ., data = valData, method = "under",
                          N = 2 * min(table(valData$Hazardous)))$data
# Target variable
x_trainval <- as.matrix(trainvalData[, names(trainvalData) != "Hazardous"])
y_trainval <- trainvalData$Hazardous
x_trainval_us <- as.matrix(trainvalData_us[, names(trainvalData_us) != "Hazardous"])
y_trainval_us <- trainvalData_us$Hazardous
x_trainval_os <- as.matrix(trainvalData_os[, names(trainvalData_os) != "Hazardous"])
y_trainval_os <- trainvalData_os$Hazardous

x_train <- as.matrix(trainData[, names(trainData) != "Hazardous"])
y_train <- trainData$Hazardous
x_train_us <- as.matrix(trainData_us[, names(trainData_us) != "Hazardous"])
y_train_us <- trainData_us$Hazardous
x_train_os <- as.matrix(trainData_os[, names(trainData_os) != "Hazardous"])
y_train_os <- trainData_os$Hazardous
x_validation <- as.matrix(valData[, names(valData) != "Hazardous"])
y_validation <- valData$Hazardous
x_validation_os <- as.matrix(valData_os[, names(valData_os) != "Hazardous"])
y_validation_os <- valData_os$Hazardous
x_validation_us <- as.matrix(valData_us[, names(valData_us) != "Hazardous"])
y_validation_us <- valData_us$Hazardous
x_test <- as.matrix(testData[, names(testData) != "Hazardous"])
y_test <- testData$Hazardous
```

Why oversampling works well? In unbalanced datasets, the model tends to be biased towards the majority class. Oversampling ensures that the minority class is represented equally, allowing the model to learn its characteristics better.

Without oversampling, the model might achieve high accuracy by predominantly predicting the majority class, which is misleading in the context of imbalanced datasets. By balancing the classes, oversampling forces the model to consider the minority class more seriously, leading to better performance metrics for that class.

Test and Validation set remains unbalanced, which is a realistic representation of the "real-world" scenario we are trying to model. Evaluating the model on an unbalanced test set after training on an oversampled training set demonstrates how well the model can handle class unbalance in practice. Testing on the oversampled validation set, on the other hand, can inflate the F1 score, making the classification task seem easier than it actually is. For this reason, the models will only be evaluated on the original, unbalanced validation set.

When the model is evaluated on the unbalanced test set, it should ideally show improved metrics for the minority class without compromising too much on the majority class performance. This indicates the model's robustness and effectiveness in dealing with real-world data distributions.

Metrics to evaluate the models

This section of the project contains functions essential for evaluating and comparing models. The key functions are: *calculate_confusion_matrix:* Computes the confusion matrix to assess classification performance.

plot_roc_curve: Plots the ROC curve and calculates the AUC for model evaluation. *calculate_metrics:* Derives performance metrics such as accuracy, balanced accuracy, sensitivity, specificity, precision, and F1 score from the confusion matrix. *evaluate_model_with_roc:* Integrates confusion matrix calculation, metric evaluation, and ROC curve plotting for comprehensive model assessment. *combine_metrics:* Aggregates performance metrics from multiple models into a single data frame to facilitate the comparison. *calculate_means*: Computes average performance metrics across several models. *plot_radar_chart:* Visualizes and compares model performance using a radar chart. These functions facilitate detailed model evaluation, comparison, and visualization, aiding in the selection of the best-performing models.

```
# functions definition cell:
# function to calculate confusion matrix
calculate_confusion_matrix <- function(actual, predicted) {
  table(Predicted = predicted, Actual = actual)
}
# Function to plot ROC curve
plot_roc_curve <- function(model, valData, y_val, link) {
  pred_val <- predict(model, newdata = valData, type = "response")
  roc_obj <- roc(y_val, pred_val)
  auc_value <- auc(roc_obj)
  plot(roc_obj, main = paste("ROC Curve -", link),
       col = "darkred", lwd = 2, print.auc = TRUE,
       cex.main = 0.7, cex.lab = 0.8,
       cex.axis = 0.8)
  return(auc_value)}
# function to calculate metrics, given a confusion matrix
calculate_metrics <- function(conf_matrix) {
  # Val. conf. matrix
  TN <- conf_matrix[1, 1]
  FP <- conf_matrix[1, 2]
  FN <- conf_matrix[2, 1]
  TP <- conf_matrix[2, 2]
  # Obs.
  n <- sum(conf_matrix)
  # Accuracy
  accuracy <- (TP + TN) / n
  # Calcolo della balanced accuracy
  sensitivity <- TP / (TP + FN)  # anche noto come recall
  specificity <- TN / (TN + FP)
  balanced_accuracy <- (sensitivity + specificity) / 2
  # F1 score
  precision <- TP / (TP + FP)
  recall <- sensitivity
  f1_score <- 2 * (precision * recall) / (precision + recall)
  # list
  list(
    accuracy = accuracy,
    balanced_accuracy = balanced_accuracy,
    f1_score = f1_score
  )
}
# function to evaluate the model (and printing roc curve)
evaluate_model_with_roc <- function(model, valData, y_val, link) {
  pred_val <- predict(model, newdata = valData, type = "response")
  pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
```

```
  conf_matrix_val <- calculate_confusion_matrix(y_val, pred_val_class)
  metrics <- calculate_metrics(conf_matrix_val)
  auc_value <- plot_roc_curve(model, valData, y_val, link)
  return(list(metrics = metrics, auc = auc_value))
}
# Function to combine metrics into a data frame
combine_metrics <- function(..., model_names = NULL) {
  metrics_list <- list(...)
  # Check if model names are provided
  if (is.null(model_names)) {
    model_names <- paste("Model", seq_along(metrics_list))
  }
  # Extract the names of the metrics
  metric_names <- names(metrics_list[[1]])
  # Create a list of values for each metric
  metric_values <- lapply(metric_names, function(metric) {
    sapply(metrics_list, function(metrics) metrics[[metric]])
  })
  # Create the data frame
  metrics_data <- data.frame(model = model_names,
                             do.call(cbind, metric_values))
  colnames(metrics_data)[-1] <- metric_names
  return(metrics_data)
}
# Function to calculate the average of the metrics
calculate_means <- function(metrics_list) {
  # Extract the metrics as vectors
  accuracies <- sapply(metrics_list, function(x) x$accuracy)
  balanced_accuracies <- sapply(metrics_list,
                                function(x) x$balanced_accuracy)
  f1_scores <- sapply(metrics_list, function(x) x$f1_score)
  # Calculate the averages
  mean_accuracy <- mean(accuracies)
  mean_balanced_accuracy <- mean(balanced_accuracies)
  mean_f1_score <- mean(f1_scores)
  list(accuracy = mean_accuracy, balanced_accuracy = mean_balanced_accuracy,
       f1_score = mean_f1_score)
}  # Return a list with the averages
# Function to draw a radar chart. will be used
# to compare models and approaches
plot_radar_chart <- function(metrics_data) {
  num_models <- nrow(metrics_data)
  num_vars <- ncol(metrics_data) - 1
  angles <- seq(0, 2 * pi, length.out = num_vars + 1)
  plot(0, 0, type = "n", xlim = c(-1.5, 1.5), ylim = c(-1.5, 1.5),
       axes = FALSE, xlab = "", ylab = "", asp = 1)
  for (i in 1:num_vars) {
    lines(c(0, cos(angles[i])), c(0, sin(angles[i])), col = "gray")
    text(1.1 * cos(angles[i]), 1.1 * sin(angles[i]),
         colnames(metrics_data)[i + 1], cex = 0.8)
  }  # Add the lines of the radar chart and the axis values
  for (j in seq(0.75, 1, by = 0.05)) {
    coords <- cbind((j - 0.75) / 0.25 * cos(angles),
```

```
                    (j - 0.75) / 0.25 * sin(angles))
    lines(coords, col = "gray")
    text((j - 0.75) / 0.25 * 1.1 * cos(angles[-(num_vars + 1)]),
         (j - 0.75) / 0.25 * 1.1 * sin(angles[-(num_vars + 1)]),
         labels = rep(j, num_vars), col = "gray", cex = 0.7)
  }  # Add concentric circles with values from 0.75 to 1
  colors <- rainbow(num_models)
  for (model in 1:num_models) {
    model_data <- as.numeric(metrics_data[model, 2:(num_vars + 1)])
    coords <- cbind((model_data - 0.75) / 0.25 * cos(angles[-(num_vars + 1)]), (model_data - 0.75) / 0.1
    polygon(coords, col = adjustcolor(colors[model], alpha.f = 0.3),
            border = colors[model])
  }  # Draw the polygons for each model with distinct colors
  legend("topright", legend = metrics_data$model,
         fill = adjustcolor(rainbow(num_models),
                            alpha.f = 0.3), cex = 0.8)
}
```

GENERAL LINEAR MODELS

Logistic Regression

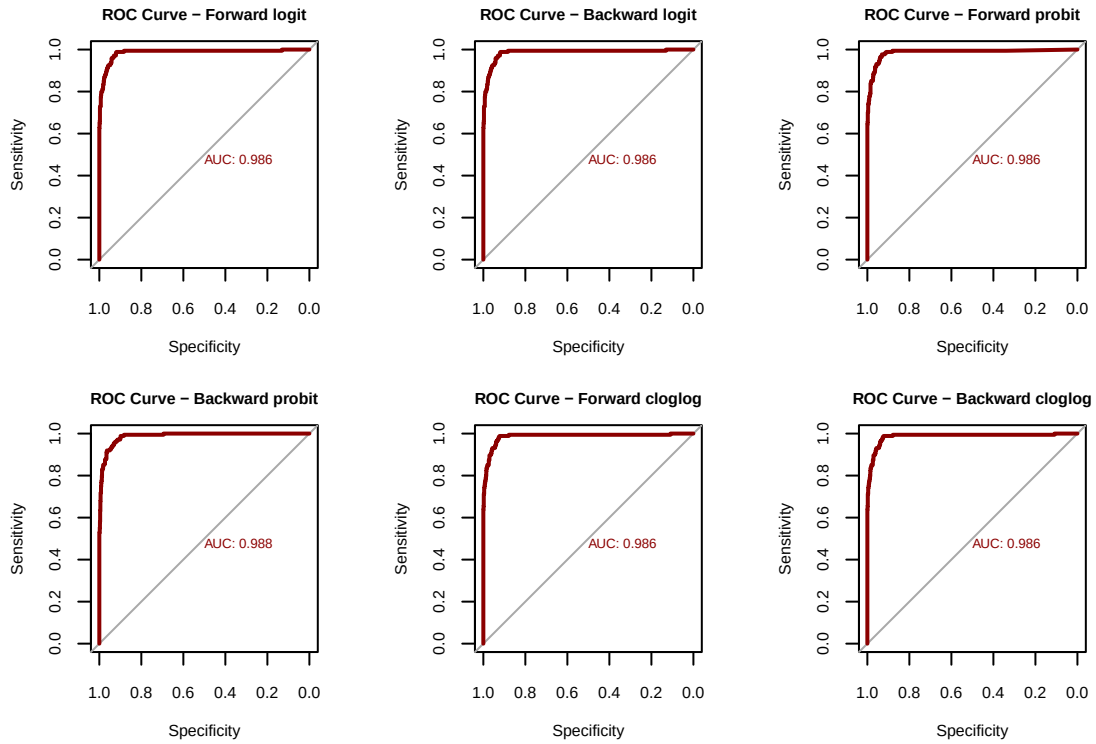
The *run_stepwise* function performs both forward and backward stepwise selection to optimize a logistic regression model. Backward and forward stepwise slection respectively, starts with a full model (all predictors) and a null model (intercept only). - Forward Selection: Uses forward stepwise selection to iteratively add predictors that improve model performance. - Backward Selection: Uses backward stepwise selection to iteratively remove predictors that do not significantly contribute to the model.

For both forward and backward models, we evaluated performance using the *evaluate_model_with_roc* function, which includes metrics calculation and ROC curve plotting. At the end we return a list containing the performance metrics for both forward and backward selected models. This approach ensures that the best subsets of predictors are chosen to optimize models performances.

```
# Stepwise forward and backward
run_stepwise <- function(trainData, valData, y_val, link) {
  full_model <- glm(Hazardous ~ ., data = trainData,
                    family = binomial(link = link))
  null_model <- glm(Hazardous ~ 1, data = trainData,
                    family = binomial(link = link))
  # Forward Selection
  stepwise_forward <- step(null_model, scope = list(lower = null_model,
                                                    upper = full_model),
                           direction = "forward")
  metrics_forward <- evaluate_model_with_roc(stepwise_forward,
                                             valData, y_val,
                                             paste("Forward", link))
  # Backward Selection
  stepwise_backward <- step(full_model, direction = "backward")
  metrics_backward <- evaluate_model_with_roc(stepwise_backward,
                                              valData, y_val,
                                              paste("Backward", link))
  return(list(forward = metrics_forward, backward = metrics_backward))
}
```

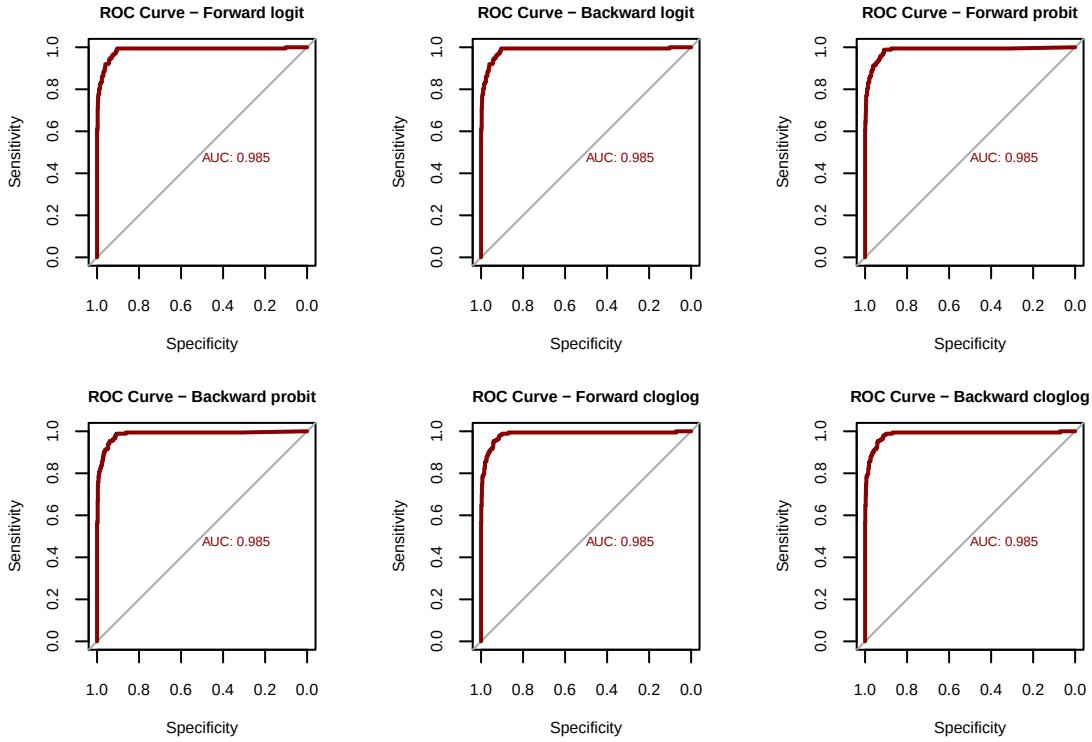
No sampling To compare which dataset, among the balanced one, the over-sampled and the undersampled, is the best to use for our purpose, we evaluated stepwise models on them with different link functions logit, probit, and complementary log-log.

```
# NO SAMPLING
par(mfrow = c(2,3))
# Logit link
metrics_logit <- run_stepwise(trainData, valData,
                              y_validation, "logit")
# Probit link
metrics_probit <- run_stepwise(trainData, valData,
                               y_validation, "probit")
# Cloglog link
metrics_cloglog <- run_stepwise(trainData, valData,
                                y_validation, "cloglog")
```



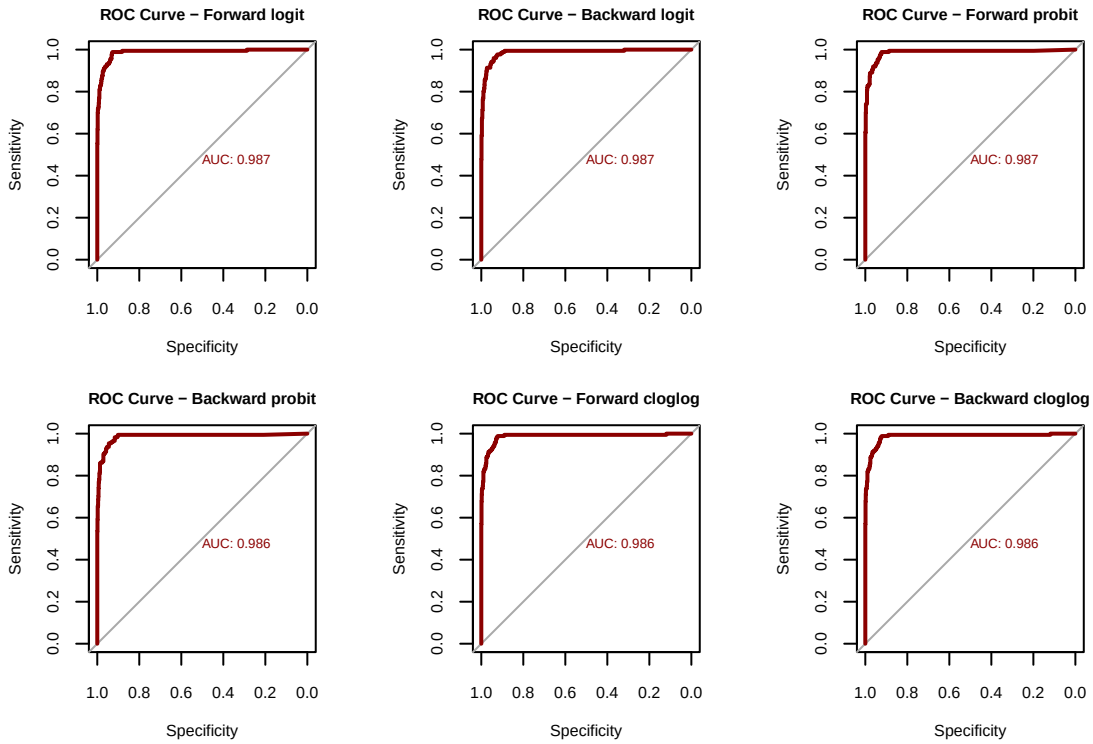

```
# OVER SAMPLING
par(mfrow = c(2,3))
# Logit link
metrics_logit_os <- run_stepwise(trainData_os, valData,
                                 y_validation, "logit")
# Probit link
metrics_probit_os <- run_stepwise(trainData_os, valData,
                                  y_validation, "probit")
# Cloglog link
```

```
metrics_cloglog_os <- run_stepwise(trainData_os, valData,
                                   y_validation, "cloglog")
```



Over sampling

```
# UNDER SAMPLING
par(mfrow = c(2,3))
# Logit link
metrics_logit_us <- run_stepwise(trainData_us, valData,
                                 y_validation, "logit")
# Probit link
metrics_probit_us <- run_stepwise(trainData_us, valData,
                                  y_validation, "probit")
# Cloglog link
metrics_cloglog_us <- run_stepwise(trainData_us, valData,
                                   y_validation, "cloglog")
```



Under sampling

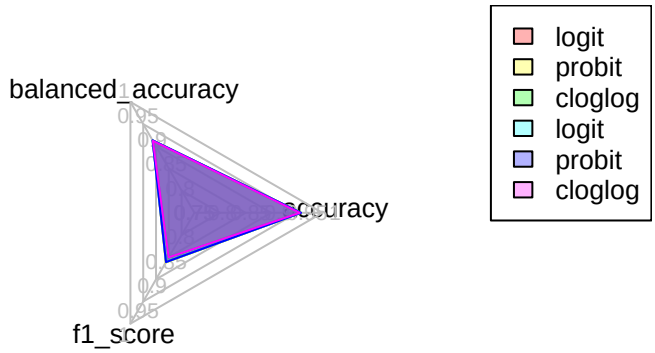
```
#### 4.2 RESULTS ####
# metrics
results_list <- list(
  logit_forward = metrics_logit$forward$metrics,
  logit_backward = metrics_logit$backward$metrics,
  probit_forward = metrics_probit$forward$metrics,
  probit_backward = metrics_probit$backward$metrics,
  cloglog_forward = metrics_cloglog$forward$metrics,
  cloglog_backward = metrics_cloglog$backward$metrics,
  logit_os_forward = metrics_logit_os$forward$metrics,
  logit_os_backward = metrics_logit_os$backward$metrics,
  probit_os_forward = metrics_probit_os$forward$metrics,
  probit_os_backward = metrics_probit_os$backward$metrics,
  cloglog_os_forward = metrics_cloglog_os$forward$metrics,
  cloglog_os_backward = metrics_cloglog_os$backward$metrics,
  logit_us_forward = metrics_logit_us$forward$metrics,
  logit_us_backward = metrics_logit_us$backward$metrics,
  probit_us_forward = metrics_probit_us$forward$metrics,
  probit_us_backward = metrics_probit_us$backward$metrics,
  cloglog_us_forward = metrics_cloglog_us$forward$metrics,
  cloglog_us_backward = metrics_cloglog_us$backward$metrics
)
# metrics' dataframe
results_df <- do.call(rbind, lapply(results_list, as.data.frame))
results_df <- data.frame(Model = names(results_list), results_df)
print(results_df)# Visualization
```

Results and model's choice

```
##                                   Model  accuracy balanced_accuracy  f1_score
## logit_forward              logit_forward 0.9581911         0.9135035 0.8611898
## logit_backward            logit_backward 0.9581911         0.9135035 0.8611898
## probit_forward            probit_forward 0.9581911         0.9120566 0.8619718
## probit_backward          probit_backward 0.9581911         0.9135035 0.8611898
## cloglog_forward          cloglog_forward 0.9556314         0.9122613 0.8505747
## cloglog_backward        cloglog_backward 0.9556314         0.9122613 0.8505747
## logit_os_forward        logit_os_forward 0.9368601         0.8539186 0.8168317
## logit_os_backward      logit_os_backward 0.9368601         0.8539186 0.8168317
## probit_os_forward      probit_os_forward 0.9368601         0.8552780 0.8140704
## probit_os_backward    probit_os_backward 0.9419795         0.8630283 0.8300000
## cloglog_os_forward    cloglog_os_forward 0.9428328         0.8676265 0.8277635
## cloglog_os_backward cloglog_os_backward 0.9428328         0.8676265 0.8277635
## logit_us_forward        logit_us_forward 0.9360068         0.8502372 0.8201439
## logit_us_backward      logit_us_backward 0.9317406         0.8436175 0.8086124
## probit_us_forward      probit_us_forward 0.9343003         0.8476414 0.8153477
## probit_us_backward    probit_us_backward 0.9283276         0.8382237 0.8000000
## cloglog_us_forward    cloglog_us_forward 0.9385666         0.8566233 0.8217822
## cloglog_us_backward cloglog_us_backward 0.9385666         0.8566233 0.8217822
```

As we imagined, forward and backward approaches gave almost identical results, while unbalanced data seems to consistently grant higher metrics.

```
metrics_data <- combine_metrics(metrics_logit$forward$metrics,
                                metrics_probit$forward$metrics,
                                metrics_cloglog$forward$metrics,
                                metrics_logit$backward$metrics,
                                metrics_probit$backward$metrics,
                                metrics_cloglog$backward$metrics,
                                model_names = c("logit", "probit",
                                                "cloglog"))
plot_radar_chart(metrics_data)
```



As for the activation function, cloglog seems to perform best with over and under sampled data, but performed poorly with the unbalanced one, with which logit and probit are very similar. Overall, the logistic regression model with probit activation function and unbalanced data obtained the higher F1 score (0.8619718), therefore we choose this as the best logistic regression model.

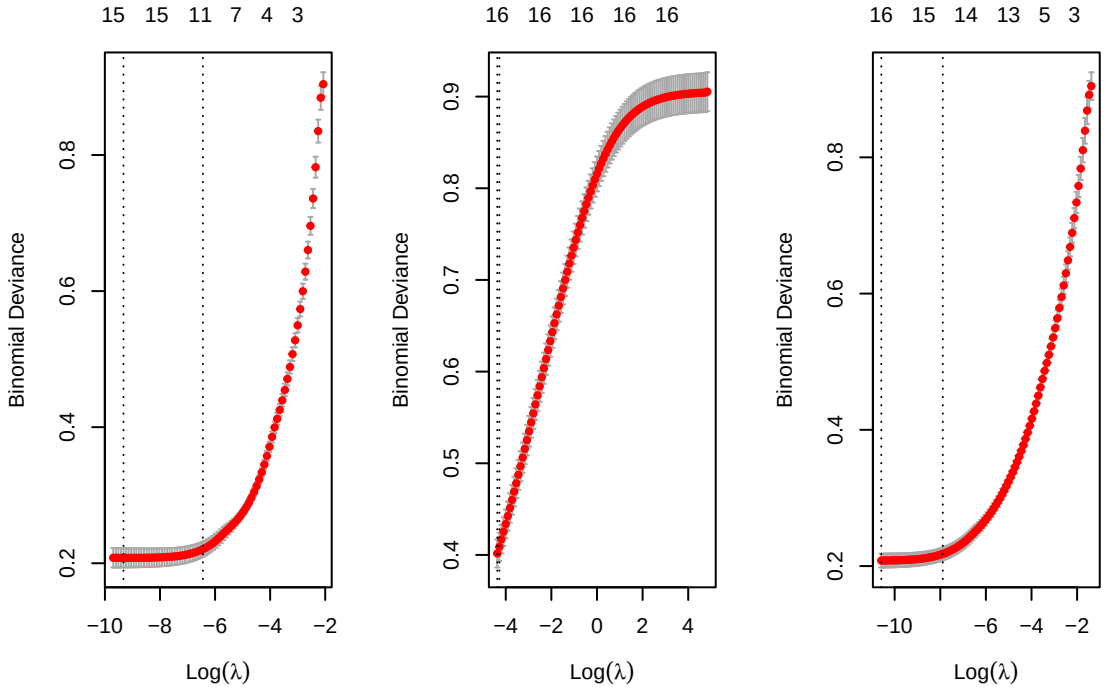
Lasso, Ridge and Elasticnet models

To achieve a better comparison, we have also tested regularized regression models, including Lasso, Ridge, and Elastic Net, on three different datasets: the original (unbalanced), oversampled, and undersampled.

```
# Create and train the Lasso model with cross-validation
lasso_model <- cv.glmnet(x_train, y_train,
                         family = "binomial", alpha = 1)
lasso_model_os <- cv.glmnet(x_train_os, y_train_os,
                            family = "binomial", alpha = 1)
lasso_model_us <- cv.glmnet(x_train_us, y_train_us,
                            family = "binomial", alpha = 1)
ridge_model <- cv.glmnet(x_train, y_train,
                         family = "binomial", alpha = 0)
ridge_model_os <- cv.glmnet(x_train_os, y_train_os,
                            family = "binomial", alpha = 0)
ridge_model_us <- cv.glmnet(x_train_us, y_train_us,
                            family = "binomial", alpha = 0)
elastic_model <- cv.glmnet(x_train, y_train,
                           family = "binomial", alpha = 0.5)
elastic_model_os <- cv.glmnet(x_train_os, y_train_os,
                              family = "binomial", alpha = 0.5)
elastic_model_us <- cv.glmnet(x_train_us, y_train_us,
                              family = "binomial", alpha = 0.5)
```

```r
# The cv.glmnet function performs cross-validation to find the
# optimal lambda (penalty) values. Two lambda values are chosen:
# lambda.min (which gives minimum cross-validated error)
# and lambda.1se (which is within one standard error of the minimum).
```

```r
#RESULTS
par(mfrow = c(1,3))
plot(lasso_model)
plot(ridge_model)
plot(elastic_model)
```



Results

The three graphs show binomial deviance versus log(lambda) for logistic regression models with different types of regularizations. The lasso (left) and the elasticnet (right) appear more robust compared to the Ridge (middle) graph, as their deviance curves have a more stable and well-defined minimum, suggesting that these models are less sensitive to changes in lambda. The Ridge graph's deviance increases much more steeply, indicating that the model is more sensitive to changes in lambda and quickly deteriorates in performance as lambda moves away from the optimal value. #### Models evaluation

```r
#MODEL EVALUATION
# LASSO
# Evaluate the model on the validation set
pred_val <- predict(lasso_model, s = lasso_model$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
```

```
                                                pred_val_class)
metrics_lasso_std <- calculate_metrics(conf_matrix_val)
# Evaluate the oversampled model on the validation set
pred_val <- predict(lasso_model_os, s = lasso_model_os$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
                                                pred_val_class)
metrics_lasso_os <- calculate_metrics(conf_matrix_val)
# Evaluate the undersampled model on the validation set
pred_val <- predict(lasso_model_us, s = lasso_model_us$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
                                                pred_val_class)
metrics_lasso_us <- calculate_metrics(conf_matrix_val)
# Extract the model coefficients at the best lambda
lasso_coefficients <- coef(lasso_model, s = lasso_model$lambda.min)
```

```
# RIDGE
# Evaluate the model on the validation set
pred_val <- predict(ridge_model, s = ridge_model$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
                                                pred_val_class)
metrics_ridge_std <- calculate_metrics(conf_matrix_val)
# Evaluate the oversampled model on the validation set
pred_val <- predict(ridge_model_os, s = ridge_model_os$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
                                                pred_val_class)
metrics_ridge_os <- calculate_metrics(conf_matrix_val)
# Evaluate the undersampled model on the validation set
pred_val <- predict(ridge_model_us, s = ridge_model_us$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
                                                pred_val_class)
metrics_ridge_us <- calculate_metrics(conf_matrix_val)
# Extract the model coefficients at the best lambda
ridge_coefficients <- coef(ridge_model, s = ridge_model$lambda.min)
```

```
# ELASTICNET
# Evaluate the model on the validation set
pred_val <- predict(elastic_model, s = elastic_model$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
                                                pred_val_class)
metrics_elnet_std <- calculate_metrics(conf_matrix_val)
# Evaluate the oversampled model on the validation set
```

```r
pred_val <- predict(elastic_model_os, s = elastic_model_os$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
                                              pred_val_class)
metrics_elnet_os <- calculate_metrics(conf_matrix_val)
# Evaluate the undersampled model on the validation set
pred_val <- predict(elastic_model_us, s = elastic_model_us$lambda.min,
                    newx = x_validation, type = "response")
pred_val_class <- ifelse(pred_val > 0.5, 1, 0)
conf_matrix_val <- calculate_confusion_matrix(y_validation,
                                              pred_val_class)
metrics_elnet_us <- calculate_metrics(conf_matrix_val)
# Extract the model coefficients at the best lambda
elastic_coefficients <- coef(elastic_model, s = elastic_model$lambda.min)
```

```r
# metrics
results_list <- list(
  lasso = metrics_lasso_std,
  lasso_os = metrics_lasso_os,
  lasso_us = metrics_lasso_us,
  ridge = metrics_ridge_std,
  ridge_os = metrics_ridge_os,
  ridge_us = metrics_ridge_us,
  elastic = metrics_elnet_std,
  elastic_os = metrics_elnet_os,
  elastic_us = metrics_elnet_us
)
# metrics' dataframe
results_df <- do.call(rbind, lapply(results_list, as.data.frame))
results_df <- data.frame(Model = names(results_list), results_df)
# Visualization
print(results_df)
```

```
##                 Model  accuracy balanced_accuracy  f1_score
## lasso           lasso 0.9564846         0.9102066 0.8555241
## lasso_os     lasso_os 0.9385666         0.8566233 0.8217822
## lasso_us     lasso_us 0.9291809         0.8396209 0.8019093
## ridge           ridge 0.9351536         0.9026374 0.7548387
## ridge_os     ridge_os 0.8899317         0.7852546 0.7225806
## ridge_us     ridge_us 0.8779863         0.7715499 0.7002096
## elastic       elastic 0.9564846         0.9102066 0.8555241
## elastic_os elastic_os 0.9394198         0.8577524 0.8246914
## elastic_us elastic_us 0.9308874         0.8424497 0.8057554
```

These results highlight even more the lower metrics obtained with balanced data. The best metrics are obtained by both lasso and elasticnet regression on unbalanced data. Our choice for regularized glms is the lasso model, which is simpler than the elasticnet and grants the same results.

KNN

In this phase, we employed the k-nearest neighbors (k-NN) algorithm using balanced training data to address class imbalance in the dataset. We explored different values of k to determine the optimal number of neigh-

bors that maximizes model performance. We initially verified the balance of our training (*trainData_os*), validation (*valData_os*), and test (*testData*) datasets. Ensuring balanced data is crucial for reliable model training and evaluation. We tested odd values of k from 1 to 10, aiming to identify the best-performing k for the k-NN model. For each k, the model was trained on oversampled training data (*x_train_os and y_train_os*) and evaluated on oversampled validation data (*x_validation_os*). Metrics such as accuracy, balanced accuracy, and F1-score were computed for each k value and stored for further analysis.

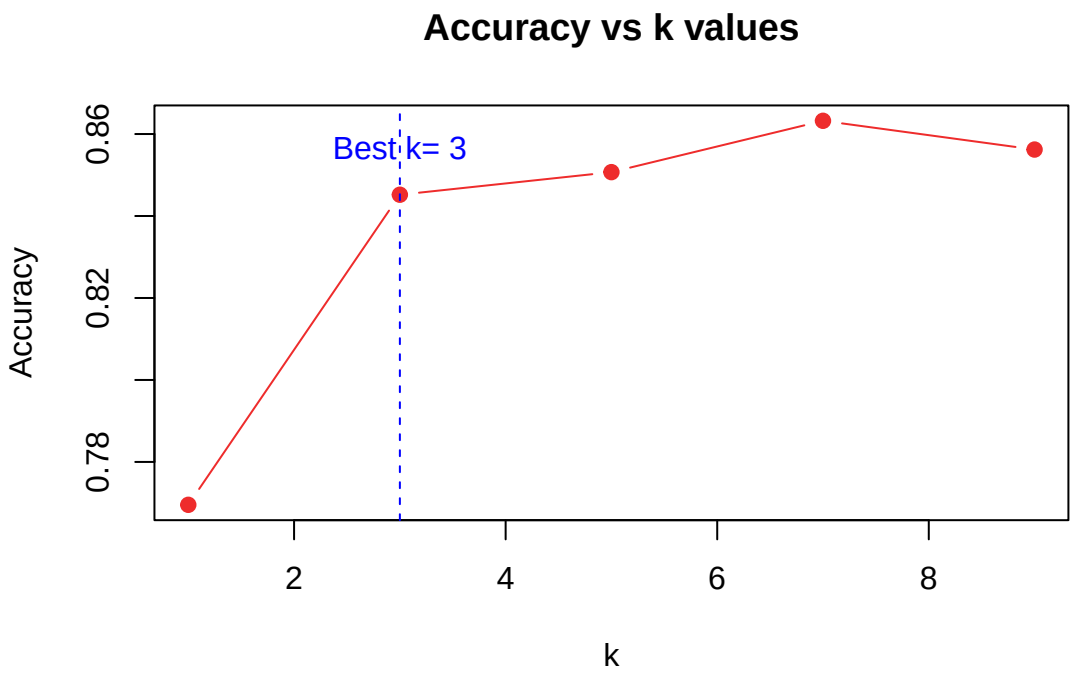
```
#k-nn with BALANCED DATA
# Search for the best k (number of neighbors to consider)
k_values <- seq(1, 10, 2)  # Example: Testing odd values of k from 1 to 10
# Initialize variables to store results
accuracy_values <- numeric(length(k_values))
balanced_accuracy_values <- numeric(length(k_values))
f1_scores <- numeric(length(k_values))
# Loop through different values of k (using balanced data)
for (i in seq_along(k_values)) {
  k <- k_values[i]
  # Train the KNN model
  knn_model <- knn(train = x_train_os, test = x_validation_os,
                   cl = y_train_os, k = k)
  # Evaluate the model
  conf_matrix_test <- calculate_confusion_matrix(y_validation_os,
                                                 knn_model)
  metrics_knn <- calculate_metrics(conf_matrix_test)
  # Store metrics
  accuracy_values[i] <- metrics_knn$accuracy
  balanced_accuracy_values[i] <- metrics_knn$balanced_accuracy
  f1_scores[i] <- metrics_knn$f1_score
}
# Compile results into a data frame
results_knn <- data.frame(k = k_values, Accuracy = accuracy_values,
                          Balanced_Accuracy = balanced_accuracy_values,
                          F1_Score = f1_scores)
print(results_knn)
```

```
##   k  Accuracy Balanced_Accuracy  F1_Score
## 1 1 0.7695391         0.8039968 0.7228916
## 2 3 0.8451904         0.8452684 0.8463451
## 3 5 0.8507014         0.8571399 0.8600939
## 4 7 0.8632265         0.8750227 0.8743672
## 5 9 0.8562124         0.8662228 0.8671911
```

The output table summarizes the performance metrics across different k values. Notably, k = 5 demonstrates the highest balanced accuracy and F1-score, indicating superior performance in handling class imbalances.

To be sure of the choice, we visualized the relationship between validation accuracy and different k values using a line plot (Accuracy vs k values).

```
# Identify the best k based on validation accuracy
par(mfrow=c(1,1))
plot(results_knn$k, results_knn$Accuracy, type = "b",
     col = "firebrick2", pch = 19, xlab = "k", ylab = "Accuracy",
     main = "Accuracy vs k values")
abline(v = 3, col = "blue", lty = 2)
best_k <- 3
text(best_k, max(results_knn$Accuracy),
     labels = paste("Best k=", 3), pos = 1,
     col = "blue")
```




```
best_k <- 3
# Train the final KNN model using the best k
final_knn_model <- knn(train = x_train_os, test = x_test,
                       cl = y_train_os, k = best_k)
conf_matrix_test <- calculate_confusion_matrix(y_test,
                                               final_knn_model)
metrics_knn <- calculate_metrics(conf_matrix_test)
# metrics
results_list <- list(
  knn = metrics_knn
)
# metrics' dataframe
results_df <- do.call(rbind, lapply(results_list, as.data.frame))
results_df <- data.frame(Model = names(results_list), results_df)
# Visualization
print(results_df)
```

```
##     Model  accuracy balanced_accuracy  f1_score
## knn   knn 0.8327645         0.7296936 0.6259542
```

Based on validation results, k=3 was identified as the optimal choice. Although k=5 shows slightly higher accuracy, k=3 represents a significant "elbow point" where the increase in performance starts to diminish. This choice ensures a good balance between model complexity and performance metrics.

That is why the final k-NN model was trained using k=3 on the oversampled training data.

We evaluated the model's performance on the original, unbalanced test dataset. Obtaining, the above confusion matrix, resulting in the following metrics: accuracy 83.78%, balanced_accuracy 73.08%, f1_score 62.60%.

In conclusion the decision to use k=3 for the final k-NN model is supported by its effective balance between accuracy and model simplicity. While k=5 shows marginally better metrics, k=3 marks a critical point

where additional complexity does not significantly improve performance. This strategic choice ensures robust classification of hazardous asteroids while maintaining computational efficiency.

LDA and QDA

LDA implementation We implemented Linear Discriminant Analysis (LDA) on Standard, Oversampled, and Undersampled training data to compare them.

```
# LDA on standard training data
lda_standard <- lda(Hazardous ~ ., data = trainvalData)
predictions_standard <- predict(lda_standard, newdata = testData)$class
conf_matrix_test <- calculate_confusion_matrix(y_test,
                                               predictions_standard)
metrics_lda_std <- calculate_metrics(conf_matrix_test)
# LDA on oversampled training data
lda_oversampled <- lda(Hazardous ~ ., data = trainvalData_os)
predictions_oversampled <- predict(lda_oversampled,
                                   newdata = testData)$class
conf_matrix_val <- calculate_confusion_matrix(y_test,
                                              predictions_oversampled)
metrics_lda_os <- calculate_metrics(conf_matrix_val)
# LDA on undersampled training data
lda_undersampled <- lda(Hazardous ~ ., data = trainvalData_us)
predictions_undersampled <- predict(lda_undersampled,
                                    newdata = testData)$class
conf_matrix_val <- calculate_confusion_matrix(y_test,
                                              predictions_undersampled)
metrics_lda_us <- calculate_metrics(conf_matrix_val)
# metrics
results_list <- list(
  lda = metrics_lda_std,
  lda_os = metrics_lda_os,
  lda_us = metrics_lda_us
)
# metrics' dataframe
results_df <- do.call(rbind, lapply(results_list, as.data.frame))
results_df <- data.frame(Model = names(results_list),
                         results_df)
# Visualization
print(results_df)
```

```
##         Model  accuracy balanced_accuracy  f1_score
## lda       lda 0.9163823         0.8424620 0.7434555
## lda_os lda_os 0.8472696         0.7561308 0.6774775
## lda_us lda_us 0.8447099         0.7540541 0.6738351
```

The confusion matrix for standard data shows 950 true negatives and 129 true positives, with 56 false negatives and 37 false positives. Its metrics outperform the ones obtained with Oversampled and Undersampled data.

Despite achieving respectable accuracy and balanced accuracy, the F1 scores obtained from all LDA models remain relatively low compared to previous models examined. This suggests a limitation in capturing both precision and recall effectively. To explore potential improvements in model performance, particularly in terms of F1 score enhancement and handling complex decision boundaries, it is prudent to investigate Quadratic Discriminant Analysis (QDA).

QDA relaxes the assumption of equal covariance matrices across classes, which could better capture the underlying data distribution in this classification task. QDA's ability to model more flexible decision boundaries may lead to improved classification accuracy, balanced accuracy, and notably, F1 score metrics, making it a compelling candidate for further investigation in this context.

```
# QDA on standard training data
qda_standard <- qda(Hazardous ~ ., data = trainvalData)
predictions_standard <- predict(qda_standard,
                                newdata = testData)$class
conf_matrix_test <- calculate_confusion_matrix(y_test,
                                               predictions_standard)
metrics_qda_std <- calculate_metrics(conf_matrix_test)
# QDA on oversampled training data
qda_oversampled <- qda(Hazardous ~ ., data = trainvalData_os)
predictions_oversampled <- predict(qda_oversampled,
                                   newdata = testData)$class
conf_matrix_val <- calculate_confusion_matrix(y_test,
                                              predictions_oversampled)
metrics_qda_os <- calculate_metrics(conf_matrix_val)
# QDA on undersampled training data
qda_undersampled <- qda(Hazardous ~ ., data = trainvalData_us)
predictions_undersampled <- predict(qda_undersampled,
                                    newdata = testData)$class
conf_matrix_val <- calculate_confusion_matrix(y_test,
                                              predictions_undersampled)
metrics_qda_us <- calculate_metrics(conf_matrix_val)
# metrics
results_list <- list(
  qda = metrics_qda_std,
  qda_os = metrics_qda_os,
  qda_us = metrics_qda_us
)
# metrics' dataframe
results_df <- do.call(rbind, lapply(results_list, as.data.frame))
results_df <- data.frame(Model = names(results_list),
                         results_df)
# Visualization
print(results_df)
```

QDA implementation

```
##         Model  accuracy balanced_accuracy  f1_score
## qda       qda 0.9581911         0.9089990 0.8765743
## qda_os qda_os 0.9385666         0.8618561 0.8385650
## qda_us qda_us 0.9368601         0.8587786 0.8355556
```

As for LDA, in QDA we see that the model trained on the unbalanced data is better than the ones trained on Oversampled or Undersampled data.

The QDA models consistently outperform the LDA models across all metrics, demonstrating higher accuracy, balanced accuracy, and notably improved F1 scores. This indicates that QDA's ability to model non-linear

decision boundaries and its flexibility in handling covariance differences between classes have effectively addressed the classification challenges observed with LDA. The significant increase in F1 scores, suggests that QDA is better suited for capturing the underlying data distribution and improving predictive performance in this classification task.

Based on these findings, transitioning from LDA to QDA appears justified and beneficial for enhancing model performance. QDA's enhanced capability to capture complex relationships within the data makes it a preferred choice for further exploration and refinement in similar classification scenarios.

Model comparison

When comparing the metrics of our unbalanced, oversampled, and undersampled models, it becomes evident that the unbalanced models consistently outperform their balanced counterparts. To confirm this observation, we plot the marginal means of the metrics for the unbalanced, oversampled, and undersampled models.

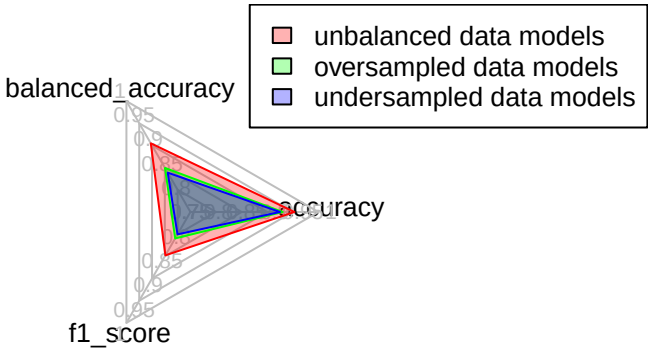
If the unbalanced models are indeed consistently superior, we can streamline the model selection process by focusing solely on the unbalanced models, thus simplifying the model comparison.

```
# knn is only oversampled due to need of balanced classes,
# so we do not count it to confront the different approaches!
# List
results_std <- (list(metrics_logit$forward$metrics,
                     metrics_probit$forward$metrics,
                     metrics_cloglog$forward$metrics,
                     metrics_logit$backward$metrics,
                     metrics_probit$backward$metrics,
                     metrics_cloglog$backward$metrics,
                     metrics_lasso_std, metrics_elnet_std,
                     metrics_lda_std, metrics_qda_std))
mean_metrics_std <- calculate_means(results_std)
results_os <- (list(metrics_logit_os$forward$metrics,
                    metrics_probit_os$forward$metrics,
                    metrics_cloglog_os$forward$metrics,
                     metrics_logit_os$backward$metrics,
                    metrics_probit_os$backward$metrics,
                    metrics_cloglog_os$backward$metrics,
                     metrics_lasso_os, metrics_elnet_os,
                    metrics_lda_os, metrics_qda_os))
mean_metrics_os <- calculate_means(results_os)
results_us <- (list(metrics_logit_us$forward$metrics,
                    metrics_probit_us$forward$metrics,
                    metrics_cloglog_us$forward$metrics,
                     metrics_logit_us$backward$metrics,
                    metrics_probit_us$backward$metrics,
                    metrics_cloglog_us$backward$metrics,
                     metrics_lasso_us, metrics_elnet_us,
                    metrics_lda_us, metrics_qda_us))
mean_metrics_us <- calculate_means(results_us)
metrics_data <- combine_metrics(mean_metrics_std,
                                mean_metrics_os, mean_metrics_us,
                                model_names = c("unbalanced data models",
                                                "oversampled data models",
```

```
                                                "undersampled data models"))
print(metrics_data)
```

```
##                       model  accuracy balanced_accuracy  f1_score
## 1   unbalanced data models 0.9531570         0.9048964 0.8477769
## 2  oversampled data models 0.9302048         0.8493759 0.8095777
## 3 undersampled data models 0.9249147         0.8387870 0.8004724
```

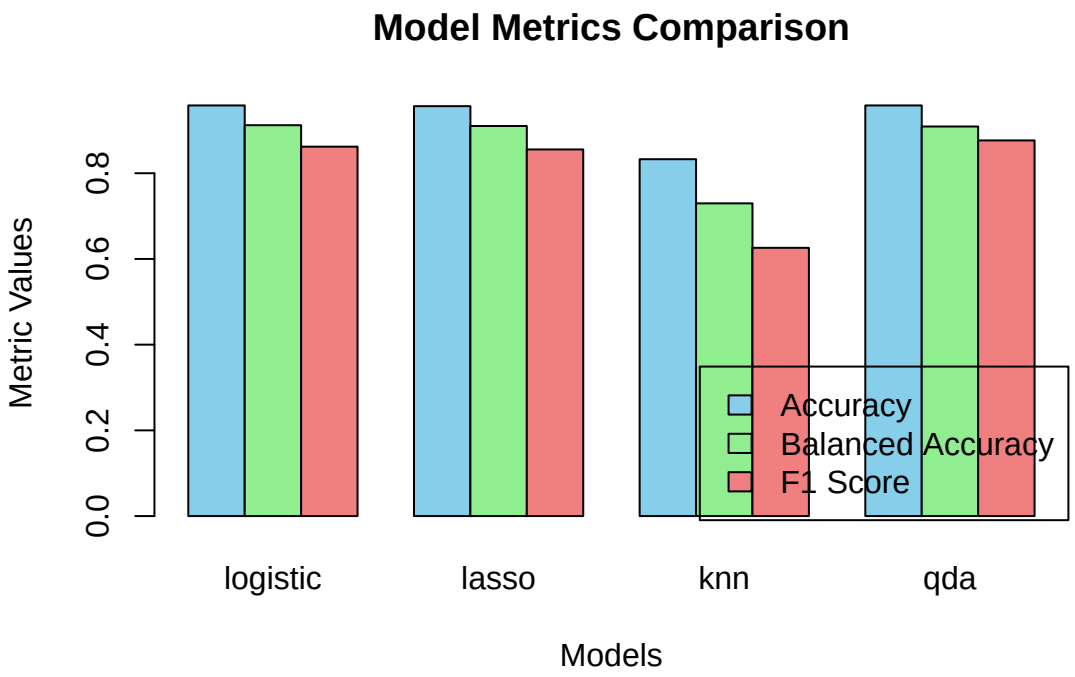
```
plot_radar_chart(metrics_data)
```



Conclusions

The radar chart above confirms that the best approach is the unbalanced one. Consequently, we will focus on using the unbalanced data models from now on, with the exception of the K-Nearest Neighbors (KNN) algorithm, which requires balanced data to perform effectively.

```
##             Model  accuracy balanced_accuracy  f1_score
## logistic logistic 0.9581911         0.9120566 0.8619718
## lasso       lasso 0.9564846         0.9102066 0.8555241
## knn           knn 0.8327645         0.7296936 0.6259542
## qda           qda 0.9581911         0.9089990 0.8765743
```



Final Choice and conclusion

As evidenced by both the metrics and the barplot, the KNN model has the poorest performance. This is likely due to its requirement for balanced data, which has shown to produce less favorable results. On the other hand, the QDA model appears to be the best option. Although it has a slightly lower balanced accuracy, the highest F1 score of 0.877 indicates a superior balance between precision and recall. Thus, despite the marginally lower balanced accuracy, the QDA model's higher F1 score makes it the most effective choice for this classification problem.