Supervised Classification with Parametric Models

A thesis submitted to the
University of Crete
for the award of the degree

of

Bachelor of Science

by

Ioannis Maris

Under the guidance of

Dr. Ioannis Kamarianakis

University of Crete
Department of Mathematics and Applied Mathematics
Foundation for Research and Technology
Heraklion, Greece

Dedicated to,

My beloved family, without whose endless love and support, I could not achieve this.

Abstract

This thesis presents a comprehensive exploration of statistical modeling techniques, focusing on binary supervised classification problems. The research contrasts parametric and non-parametric methods, emphasizing the importance of modern model selection criteria such as Akaike Information Criterion (`AIC`) and Bayesian Information Criterion (`BIC`), thus moving beyond traditional reliance on `p-values`.

The motivating application of this work utilizes a real-world breast cancer dataset. All computations were conducted using `Python (3.11)` and `R (4.2.2)`. Specifically, this work evaluates alternative estimation methods such as ordinary least squares (`OLS`), least absolute deviations (`LAD`) and logistic regression, coupled with model building methodologies like stepwise model building and shrinkage procedures. This investigation also extends to the inclusion of second order and interaction terms to original predictors.

The study offers a discussion of the methodological underpinnings of the models, explaining optimization techniques ranging from classical approaches like Newton's method and Fisher scoring to modern stochastic approaches such as stochastic gradient descent and pathwise coordinate descent. The models' performance is evaluated using cross-validation and resampling techniques such as bootstrapping. Of particular note is the development of a software that supports the implementation of these techniques. Ultimately, the thesis explores the potential for improving model performance through model averaging, aiming to offer practical solutions and valuable insights for addressing complex supervised classification problems.

Keywords: Supervised Learning, Statistical Learning, Ensemble Learning, Binary Classification, Regression, Parametric Models, Cross-Validation, Bootstrapping, Stepwise Model Selection, Stepwise Feature Selection, Breast Cancer.

Contents

List of Tables

List of Figures

List of Abbreviations

`AIC` Akaike Information Criterion

`AICc` Akaike Information Criterion Corrected

`BIC` Bayesian Information Criterion

`CI` Confidence Interval

`LASSO` Least Absolute Shrinkage and Selection Operator

`MLE` Maximum Likelihood Estimation

`OLS` Ordinary Least Squares

`VIF` Variance Inflation Factor

`LAD` Least Absolute Deviations

`SVD` Singular Value Decomposition

`pdf` Probability Density Function

`cdf` Cumulative Distribution Function

`ROC` Receiver Operating Characteristic Curve

`AUC` Area Under Curve

`CV` Cross Validation

`LRT` Likelihood Ration Test

`HP` Hyper-parameter

1

Introduction

1.1 Motivation

In this thesis, we tackle a classification problem by employing both traditional methods, such as logistic regression and non-obvious approaches like least absolute deviations (LAD), which is typically utilized for regression problems. We explore and compare simple linear models with various modifications, including models with second-order terms, penalized models, stepwise selected models and those filtered by variance inflation factor (VIF). While it may seem counterintuitive to apply LAD regression or ordinary least squares (OLS) to a binary classification problem, these techniques can potentially offer feasible and straightforward ways to make predictions. Our work is inspired by papers such as:

- "Ordinary Least Squares" by Jan de Leeuw (2004) [10].
- "Asymptotic theory of least absolute error regression" by Roger Koenker and Gilbert Bassett Jr. (1978) [6].
- "Regression Shrinkage and Selection via the Lasso" by Robert Tibshirani (1996) [29].
- "Regularization Paths for Generalized Linear Models via Coordinate Descent" by Jerome Friedman, Trevor Hastie and Robert Tibshirani (2010) [16].
- "The Adaptive Lasso and Its Oracle Properties" by Hui Zou (2006) [34].
- "Least Angle Regression, Lasso and Forward Stagewise" by Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) [12].
- "Bootstrap Methods: Another Look at the Jackknife", by Bradley Efron (1992) [11]
- "Bootstrap Methods for Developing Predictive Models", by Peter C. Austin and Jack V. Tu (2004) [3].
- "Bootstrapping the out-of-sample predictions for efficient and accurate cross-validation" by Ioannis Tsamardinos, Elissavet Greasidou and Giorgos Borboudakis (2018) [30]

Furthermore, we contrast these parametric methods with non-parametric approaches. Parametric methods make certain assumptions about the underlying distribution of the data and involve a fixed number of parameters, whereas non-parametric methods do not impose specific assumptions about the data distribution and allow for more flexibility by using a varying number of parameters. By comparing these diverse techniques, we aim to identify the most effective strategy for addressing binary classification problems in the context of the dataset under study. This thesis is inspired by [10] and [6], which evaluate logistic regression models against OLS in terms of prediction accuracy. Unlike them, who only had a few predictors to deal with, I have to work with many. So, I use techniques like penalized lasso estimations, stepwise methods and so on. These methods are useful because they recognize that not all predictors are necessarily useful.

1.2 Motivating Application

The Breast Cancer Wisconsin (Diagnostic) dataset on Kaggle[1] is a collection of medical data that pertains to breast cancer diagnosis. Kaggle provides a platform for data scientists, machine learning engineers and researchers to collaborate and compete on real-world data science challenges. It also hosts a large collection of datasets and notebooks for education and experimentation. Kaggle has a strong community of users who share knowledge and expertise through discussions, forums and code sharing. Additionally, Kaggle is known for its machine learning and data science competitions, where individuals or teams can compete to develop the best solution or model for a given problem or dataset.

The dataset contains a total of 569 observations, each representing a patient with a biopsied breast cell. Each observation includes various features such as the size of the cell nucleus, texture, smoothness, perimeter, area and more, as well as a diagnosis of whether the cell is benign (not cancerous or negative class) or malignant (cancerous or positive class). They describe characteristics of the cell nuclei present in the image. This dataset is a great resource for scientists who are interested in statistical and machine learning models for the early detection of breast cancer. Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass.

Ten real-valued features are computed for each cell nucleus, denoted as:

- ◇ *radius* (mean of distances from center to points on the perimeter)
- ◇ *texture* (standard deviation of gray-scale values)
- ◇ *perimeter*

[1] https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data

- ◇ *area*
- ◇ *smoothness* (local variation in radius lengths)
- ◇ *compactness* ($\frac{perimeter^2}{area} - 1$)
- ◇ *concavity* (severity of concave portions of the contour)
- ◇ *concave points* (number of concave portions of the contour)
- ◇ *symmetry*
- ◇ *fractal dimension* ("coastline approximation" - 1)

The mean, standard error and "worst" or largest (mean of the three largest values) of these features were computed for each image. For instance, field 3 is Mean Radius, field 13 is Radius SE and field 23 is Worst Radius. All feature values are recorded with four significant digits. There are no missing attribute values. The class distribution is 357 benign and 212 malignant.

The dataset comprises of 45% malignant and 55% benign samples and is composed of 30+1 attributes that are categorized as numeric, nominal, or binary, among others. A comprehensive description of each attribute is presented in Table 1.1. The target attribute is the only attribute with categorical values and it is binary, while the remaining 30 attributes are numeric.

The `R` code shown below imports the data into `R` and then divide it into a training set comprising 80% of the data and a holdout set comprising 20%. Furthermore, it ensures that the split is stratified.

```r
library(splitstackshape)

data <- read.csv("data.csv",
                 header = TRUE)
data <- data[, -c(1, 33)]
#set target's name: y
names(data)[names(data) == "diagnosis"] <- "y"
data$y <- ifelse(data$y == "M", 1, 0)

set.seed(666) #for reproducibility

#Add a unique sequential ID to track rows in the sample
data$rowId <- 1:nrow(data)

train_data <- stratified(data, 'y', size = 0.8)
hold_out_data <- data[!(data$rowId %in% train_data$rowId),]

data <- data[, 1:31]
train_data <- train_data[, 1:31]
hold_out_data <- hold_out_data[, 1:31]
```

```r
nrow(train_data) + nrow(hold_out_data)==dim(data)[1] #should return TRUE
X_train = subset(train_data, select=names(data[-data$y]))
y_train = train_data$y

X_hold_out <- as.matrix(hold_out_data[,
                        -which(names(hold_out_data) == "y")])

y_hold_out <- hold_out_data$y
```

In python:

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

data = pd.read_csv("data.csv").drop(['id', 'Unnamed: 32'],
                                    axis='columns')
data['diagnosis'] = data['diagnosis'].map({'M':1, 'B':0})

X = data.loc[:, data.columns != 'diagnosis']
y = data[['diagnosis']]

X_train, X_hold_out,\
y_train, y_hold_out = train_test_split(X, y,
                                   stratify=y,
                                   test_size=0.20,
                                   random_state=69)

train_data = pd.concat([X_train, y_train],
                       axis=1)
```

The table below provides a comprehensive overview of the features present in the dataset:

Feature	Description
radius_mean	Mean of distances from center to points on the perimeter
texture_mean	Standard deviation of gray-scale values
perimeter_mean	Mean size of core tumor
area_mean	Mean area of tumor
smoothness_mean	Mean of local variation in radius lengths
compactness_mean	Mean of $\text{perimeter}^2/area - 1$
concavity_mean	Mean severity of concave portions of the contour
concave.points_mean	Mean number of concave portions of the contour
symmetry_mean	Mean symmetry of tumor
fractal_dimension_mean	Mean "coastline approximation" − 1
radius_se	Standard error of distances from center to points on the perimeter
texture_se	Standard error of gray-scale values
perimeter_se	Standard error of size of core tumor
area_se	Standard error of area of tumor
smoothness_se	Standard error of local variation in radius lengths
compactness_se	Standard error of $\text{perimeter}^2/area - 1$
concavity_se	Standard error of severity of concave portions of the contour
concave points_se	Standard error of number of concave portions of the contour
symmetry_se	Standard error of symmetry of tumor
fractal_dimension_se	Standard error of "coastline approximation" − 1
radius_worst	Worst or largest mean value of distances from center to points on the perimeter
texture_worst	Worst or largest mean value of gray-scale values
perimeter_worst	Worst or largest mean value of size of core tumor
area_worst	Worst or largest mean value of area of tumor
smoothness_worst	Worst or largest mean value of local variation in radius lengths
compactness_worst	Worst or largest mean value of $\text{perimeter}^2/area - 1$
concavity_worst	Worst or largest mean value of severity of concave portions of the contour
concave.points_worst	Worst or largest mean value of number of concave portions of the contour
symmetry_worst	Worst or largest mean value of symmetry of tumor
fractal_dimension_worst	Worst or largest mean value of "coastline approximation" − 1
diagnosis (label)	Diagnosis of breast tissues (M: malignant $\rightarrow$ 1, B: benign $\rightarrow$ 0)

Table 1.1: Description of the features in breast cancer dataset

1.3 Methodological Tools

The subsequent chapters cover various aspects of statistical modeling, including estimation methods, penalized estimation methods, stepwise feature selection, penalized estimation, cross-validation and bootstrapping methods and non-parametric models.

Chapter 2 delves into various estimation techniques, contrasting logistic regression with conventional regression. The chapter highlights the utility of `VIF` filtering in the analysis. A comparison is drawn between the outcomes derived from a full model incorporating second order terms and those from a VIF-filtered model. To choose a `p-value` threshold, we adopt a moderate viewpoint that enables us to transcend the conventional reliance on a 0.05 threshold for `p-values` [7]. Additionally, alternative methodologies are employed to ascertain consensus on "strongly significant" features. In this chapter, we implement the regression models in `R` using K-fold and nested cross-validation techniques. The details and discussion of these methodologies will be presented in the final chapter.

Chapter 3 discusses forward/backward selection and `AIC` versus `BIC` for feature selection. The chapter reports on the results of `AICc`-based forward/backward selection for both logistic and conventional regression. The chapter also discusses the details of the results and uses figures to show if there are models that lead to dramatically different interpretations.

Chapter 4 presents penalized estimation and discusses lasso and ridge methods. The chapter also discusses the algorithm behind `glmnet`[2], pathwise coordinate descent. The chapter compares the results of penalized logistic vs penalized `LS` vs penalized least absolute deviations and analyzes if they are significantly different in terms of the features they include. Prediction accuracy of alternative estimations is evaluated using the following criteria: AUC[3] and `F1` score.

Chapter 5 The chapter investigates whether there is a dominant model or if there are competing specifications from chapters 3-4, reporting bootstrapping confidence intervals for each model and evaluate their generalization capabilities using repeated 10-fold cross-validation and nested cross-validation. In this chapter, we have also developed a custom nested cross-validation class using Python, which allows us to obtain unbiased performance metrics for any model, including those from `R`. Additionally, we present non-parametric estimation approaches based on bootstrapping and compare them with their parametric counterparts. The chapter investigates whether there is a dominant model or if there are competing specifications that perform very similarly and explores the potential for improvement through model averaging. Finally, **chapter 6** concludes the thesis.

[2] A package in `R`, that fits generalized linear and similar models via penalized maximum likelihood
[3] Area Under the ROC Curve

1.3.1 Parametric Models

OLS (Ordinary Least Squares) regression is a widely applied statistical method that helps to determine the relationship between one or more independent variables and a continuous dependent variable. It estimates the coefficients of the linear equation by minimizing the sum of the squared differences between the observed values and the predicted values. The approach to solving OLS regression involves using a set of equations known as normal equations. These equations help to determine the best values of the model parameters, which are the coefficients of the independent variables. The values are obtained by setting the partial derivatives of the sum of squared errors with respect to each of the parameters to zero, then solving for the parameters using matrix algebra.

An alternative estimation method for linear regressions is the Least Absolute Deviations (LAD) method, which minimizes the sum of absolute values of residuals instead of the sum of squared deviations. LAD estimation is more robust to outliers compared to OLS, similar to the median compared to the mean.

Logistic Regression is a statistical method for supervised classification problems, commonly used for binary classification. It is valued for its probability estimates, interpretability, robustness and flexibility.

Since the outcome is a probability, the dependent variable is bounded between 0 and 1. In logistic regression, a logit transformation is applied on the odds—that is, the probability of success divided by the probability of failure. This is also commonly known as the log odds, or the natural logarithm of odds and this logistic function is represented by the following formulas:

$$\text{Logit}(\pi) = \frac{1}{1 + e^{-\pi}} \quad \text{(Sigmoid)} \ ,$$

$$\log\left(\frac{\pi}{1-\pi}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_k X_k.$$

In this logistic regression equation, $\text{Logit}(\pi)$ is the dependent variable. The beta parameter, or coefficient, in this model is commonly estimated via maximum likelihood estimation (MLE).

1.4 Data Visualization

Figure 1.1 shows the distribution of the binary variable, which is the response variable. Figure 1.2, on the other hand, is a pairplot that includes some explanatory features that are highly correlated with the target variable.

```
import matplotlib.pyplot as plt
plt.subplot(1, 2, 1) #1-row, 2-column goes to 1
sns.histplot(np.ravel(y), color=(0.15,0.5,0.98))
plt.title('Distribution of $\mathbf{diagnosis}$', size=10)
plt.xlabel(f'count 1: {list(np.ravel(y)).count(1)},\
        count 0: {list(np.ravel(y)).count(0)}', size=8)

plt.subplot(1, 2, 2) #1-row, 2-column goes to 2
sns.histplot(np.ravel(y_hold_out), color='orange',
                 edgecolor='black', label='hold-out-set')
sns.histplot(np.ravel(y_train), label='train-set')
plt.ylabel('')
plt.legend(loc='best')
plt.title('$\mathbf{Stratified}$ split', size=10)
plt.show()
```

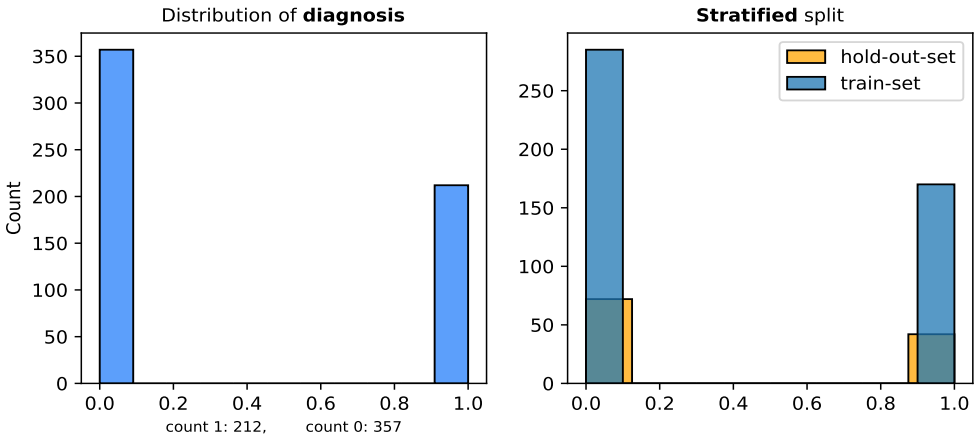


Figure 1.1: Distribution of the target (left) versus the validation data (right)

```
import seaborn as sns
# Pairplot with seaborn
sns.pairplot(data, hue="diagnosis", vars =
             ['texture_worst', 'concave points_worst',
             'texture_mean', 'area_mean',
             'concave points_mean'])
plt.show()
```

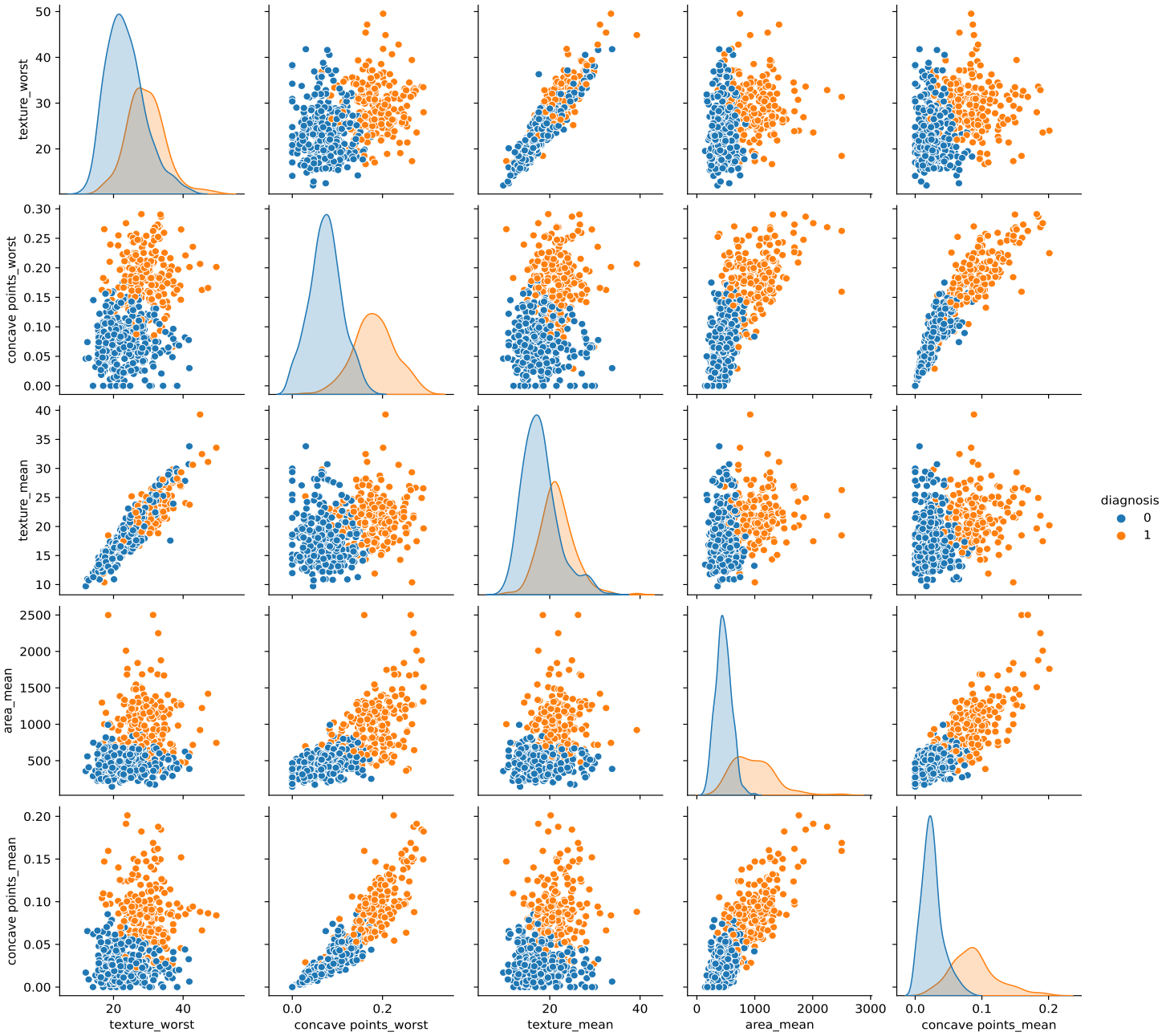


Figure 1.2: Pairplot of features highly correlated with the target variable

One can achieve the same outcome in R by implementing the following code.

The resulting outputs are illustrated in Figures 1.3 and 1.4.

```
par(mfrow=c(1,3), mai=c(0.8,0.3,1,0.4))
hist(data$y, main="Full Data",
     xlab="diagnosis", col="red", cex.main=2)
hist(train_data$y, main="Training Data",
     xlab="diagnosis", col="steelblue", cex.main=2)
hist(hold_out_data$y, main="Hold-Out Data",
     xlab="diagnosis", col="steelblue", cex.main=2)
```

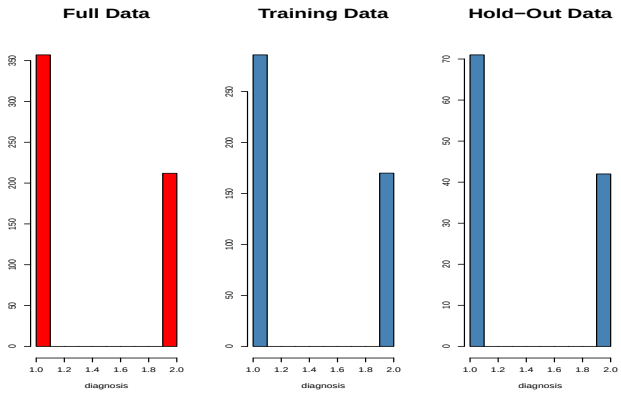


Figure 1.3: Distribution of `diagnosis` for each split

```
library(ggplot2)
library(GGally)

data$y <- as.factor(data$y)
# Create a custom function for scatter plots
custom_scatter <- function(data, mapping, ...) {
  ggplot(data = data, mapping = mapping) +
    geom_point(shape = 1, ...) +
    theme(axis.text.x = element_blank(),
          axis.text.y = element_blank())
}
# Create the pair plot with the custom scatter plot function
pairplot <- ggpairs(data, columns = c(1,5,6,22,23,16),
                    mapping = ggplot2::aes(color = y),
                    upper = list(continuous =
                                   wrap(custom_scatter, alpha = 0.3)))
pairplot <- pairplot + theme(axis.text.x = element_blank(),
                             axis.text.y = element_blank())
data$y <- as.numeric(as.vector(data$y)) #reformat
pairplot
```

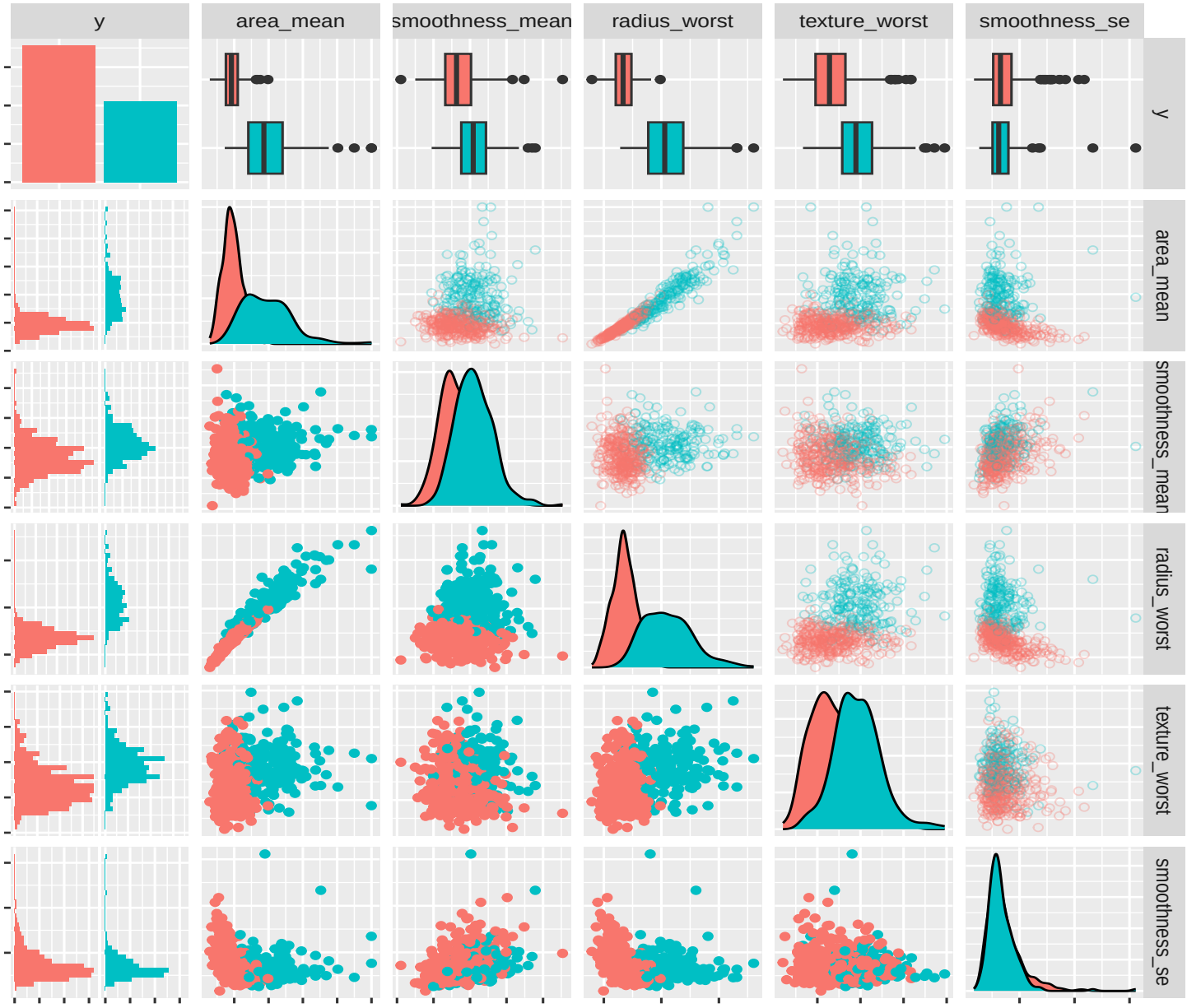


Figure 1.4: Pairplot via `ggplot2`

The following figure (Figure 1.5) showcases the initial images (raw images), also known as histopathological images of the breast, which are utilized for feature measurement.

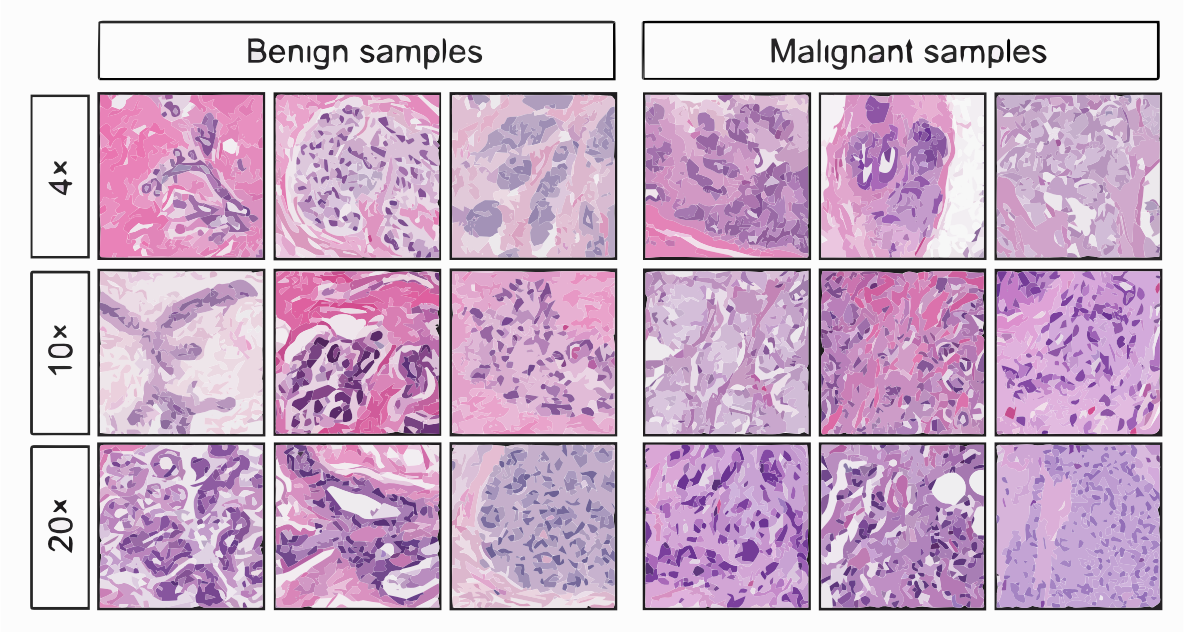


Figure 1.5: Histopathological images of the breast for each class (Reference)

The Python code below, allows us to visualize some of the distributions (Figure 1.6) of our predictors.

```
##Visualize the distributions with seaborn package (python)##
import seaborn as sns
import matplotlib.pyplot as plt
features = list(data.columns[1:14]) +\
        ['texture_worst', 'smoothness_se']
colors = ['red', 'green', (0,0.67,0.88), 'purple', 'orange',
          (0,0.4,0.8), (0.3,0.4,0.8), 'brown', 'gray']
# Create the grid of plots
fig, axes = plt.subplots(nrows=5, ncols=3, figsize=(12, 12))
# Loop over the features and plot them in the grid
for i, feature in enumerate(features):
    row = i // 3
    col = i % 3
    sns.histplot(data[feature],
                 kde=True, ax=axes[row, col],
                 color=colors[i%len(colors)])
    axes[row, col].set_xlabel(feature)
plt.tight_layout()
```

```
plt.show()
```

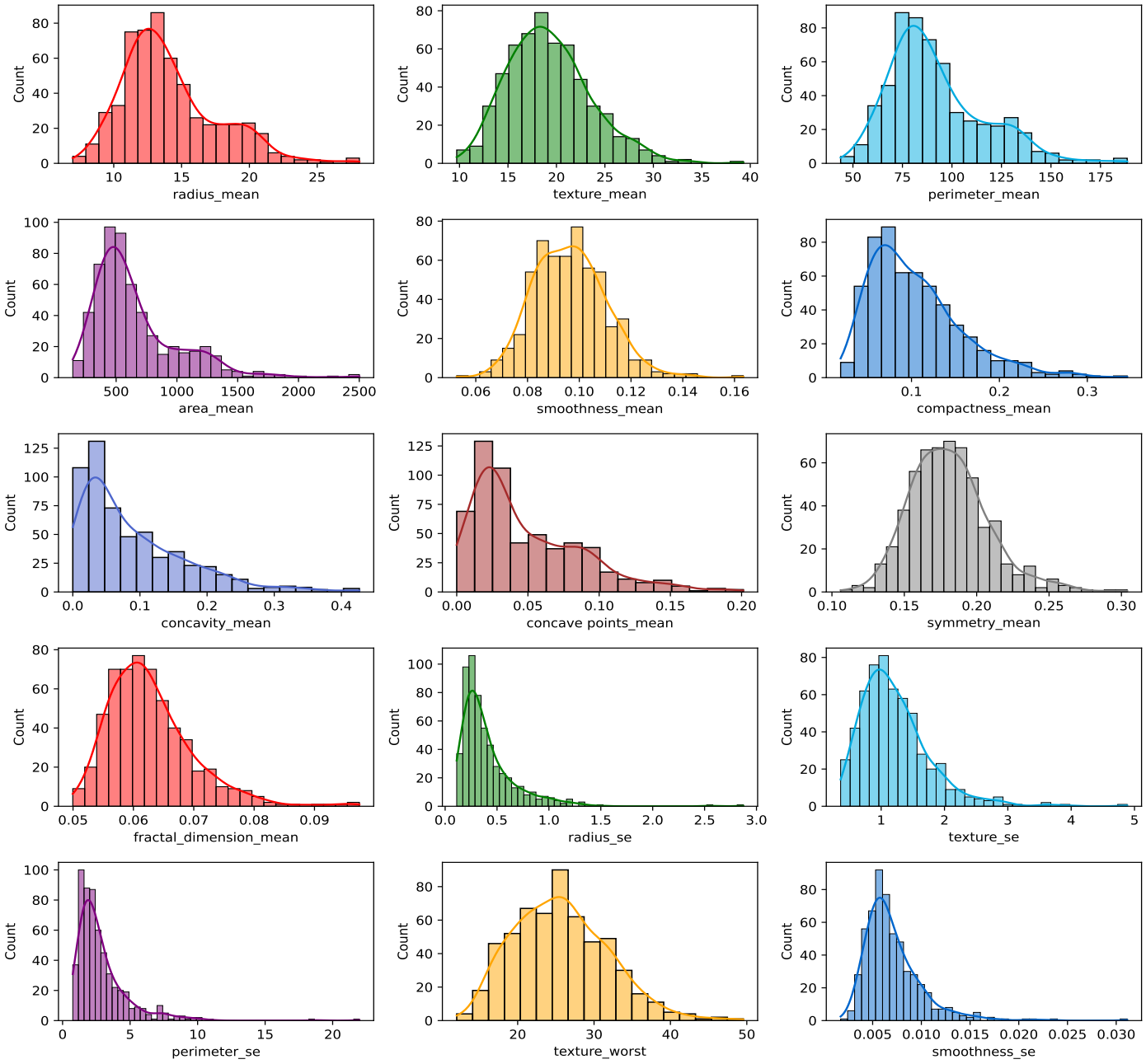


Figure 1.6: Distribution of the Predictors

1.4.1 Pearson and Spearman Correlations

The Pearson correlation coefficient is a measure of linear association that can be viewed as a metric for standardized covariance. The coefficient is formulated as follows:

$$r_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

where $\text{cov}(X,Y)$ is the covariance between X and Y, σ_X and σ_Y are the standard deviations of X and Y and $\bar{X}$ and $\bar{Y}$ are their respective means.

Spearman's rank correlation coefficient is an alternative, non-parametric measure that quantifies the strength of monotonic (not necessarily linear) association. It is defined as:

$$r_s = \frac{\text{cov}(\text{rank}(X), \text{rank}(Y))}{\sigma_{\text{rank}(X)}\sigma_{\text{rank}(Y)}} = \frac{\sum_{i=1}^{n}(R_i - \bar{R})(S_i - \bar{S})}{\sqrt{\sum_{i=1}^{n}(R_i - \bar{R})^2}\sqrt{\sum_{i=1}^{n}(S_i - \bar{S})^2}}$$

where $\text{rank}(X)$ and $\text{rank}(Y)$ are the ranks of X and Y, R_i and S_i are the ranks of the ith observation of X and Y and $\bar{R}$ and $\bar{S}$ are their respective means.
Spearman correlation is a statistical measure that assesses the strength and direction of the relationship between two variables. It is a non-parametric measure of correlation that does not assume that the variables follow a normal distribution. Instead, it ranks the data points of each variable, calculates the difference between each pair of ranks and then computes the correlation coefficient between the two sets of ranks.
The key difference between Spearman and Pearson correlation is that Spearman correlation is based on the ranks of the data points, while Pearson correlation is based on the actual values of the data points. As a result, Spearman correlation is more robust to outliers and non-linear relationships between the variables, while Pearson correlation is more sensitive to these factors. Figure 1.7 depicts differences between the two correlation metrics for different types of data.

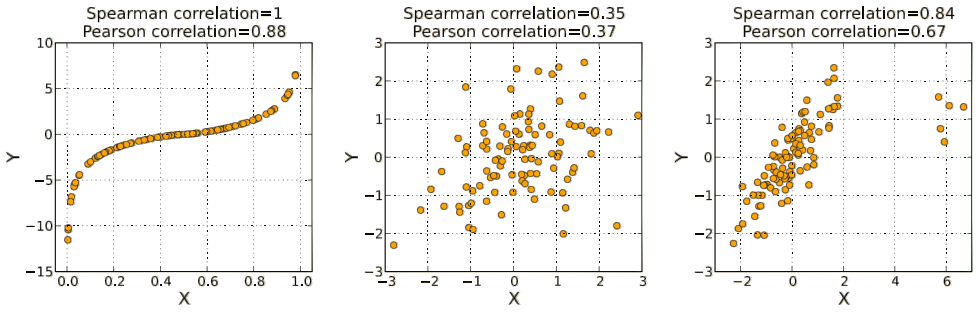


Figure 1.7: Pearson vs Spearman (Reference)

One can calculate the `t-score` and `p-value`, which correspond to the bivariate normality assumption and the null hypothesis stating that the two examined variables are not correlated. Clearly, `p-values` very close to zero provide evidence against the null hypothesis. The formula to calculate the `t-score` of a correlation coefficient (`coef`) is as follows:

$$t = \texttt{coef}\frac{\sqrt{n-2}}{\sqrt{1-\texttt{coef}^2}}$$

The `p-value` is calculated as the corresponding two-sided `p-value` for the t-distribution with $n-2$ degrees of freedom. To calculate the `p-value` for a Pearson correlation coefficient in `pandas`, one could use the `pearsonr()` and `spearmanr()` methods from the `scipy` library:

```
from scipy.stats import pearsonr, spearmanr
from matplotlib.colors import LinearSegmentedColormap

## make a theme ##
my_purple = (0.3, 0.1, 0.3) #RGB
my_cyan = (0.15, 0.98, 0.98) #RGB
cmap = LinearSegmentedColormap.from_list(
    "custom_cmap",
    [
        (0.0, my_cyan),  # Close to -1: blue
        (0.5, "white"),  # Close to 0: white
        (1.0, my_purple),
    ],
)

## The following code shows how to calculate the Pearson
    correlation
# coefficient and corresponding p-value for the x and y columns

# create function to calculate p-values for each
# pairwise correlation coefficient
def r_pvalues_pearson(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.DataFrame(columns=df.columns)
    p = cols.transpose().join(cols, how='outer')
    for r in df.columns:
        for c in df.columns:
            tmp = df[df[r].notnull() & df[c].notnull()]
            p[r][c] = round(pearsonr(tmp[r], tmp[c])[1], 2)
    return p

def r_pvalues_spearman(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.DataFrame(columns=df.columns)
    p = cols.transpose().join(cols, how='outer')
    for r in df.columns:
        for c in df.columns:
            tmp = df[df[r].notnull() & df[c].notnull()]
```

```
            p[r][c] = round(spearmanr(tmp[r], tmp[c])[1], 2)
    return p
```

The following function, `'heatmap_pval'`, plots the heatmaps (Pearson's, Spearman's) and their corresponding `p-values` for the explanatory variables of the motivating application.

```
import matplotlib.patches as mpatches

def heatmap_pval(data: pd.DataFrame, corr_method: str):
    # Calculate the correlation matrix
    corr_matrix = data.corr(corr_method)
    # Create masks for the upper and lower triangles, including
        the diagonals
    mask_upper = np.triu(np.ones_like(corr_matrix, dtype=bool))
    mask_lower = np.tril(np.ones_like(corr_matrix, dtype=bool))
    ### p-values ###
    if corr_method == 'spearman':
        p_values = np.array(r_pvalues_spearman(data))
    else:
        p_values = np.array(r_pvalues_pearson(data))
    # Create a masked matrix for the p-values
    p_values_masked = np.where(mask_lower, np.nan, p_values)
    # Create the heatmap with masked correlation
    # coefficients and p-values as annotations
    ax = sns.heatmap(corr_matrix, mask=mask_upper,
                     vmin=-1, vmax=1, cmap=cmap)
    # Add p-value annotations to the upper triangle
    for i in range(corr_matrix.shape[0]):
        for j in range(corr_matrix.shape[1]):
            if i < j:
                ax.text(j + 0.5, i + 0.5, f'{p_values_masked[i,
                    j]:.2g}',
                        ha='center', va='center',
                        fontsize=4.8, color='black')
    #make a legend for the p-values
    handles, labels = plt.gca().get_legend_handles_labels()
    patch = mpatches.Patch(color='black', label='p-values')
    handles.extend([patch])
    plt.title(f'{corr_method.capitalize()} Correlations Heatmap
        ')
    plt.legend(handles=handles, loc='upper left',
               bbox_to_anchor=(1.03, -0.05),
               prop={'size': 7.5})
    plt.show()
```

The two heatmaps (Figure 1.8, Figure 1.9) are pretty similar; In other situations, they could look a lot more different. One observes that the vast majority of bivariate associations are positive. 86.7% of Pearson and 90% of Spearman coefficients were found to be positive.

```
heatmap_pval(data,
             corr_method='pearson')
```

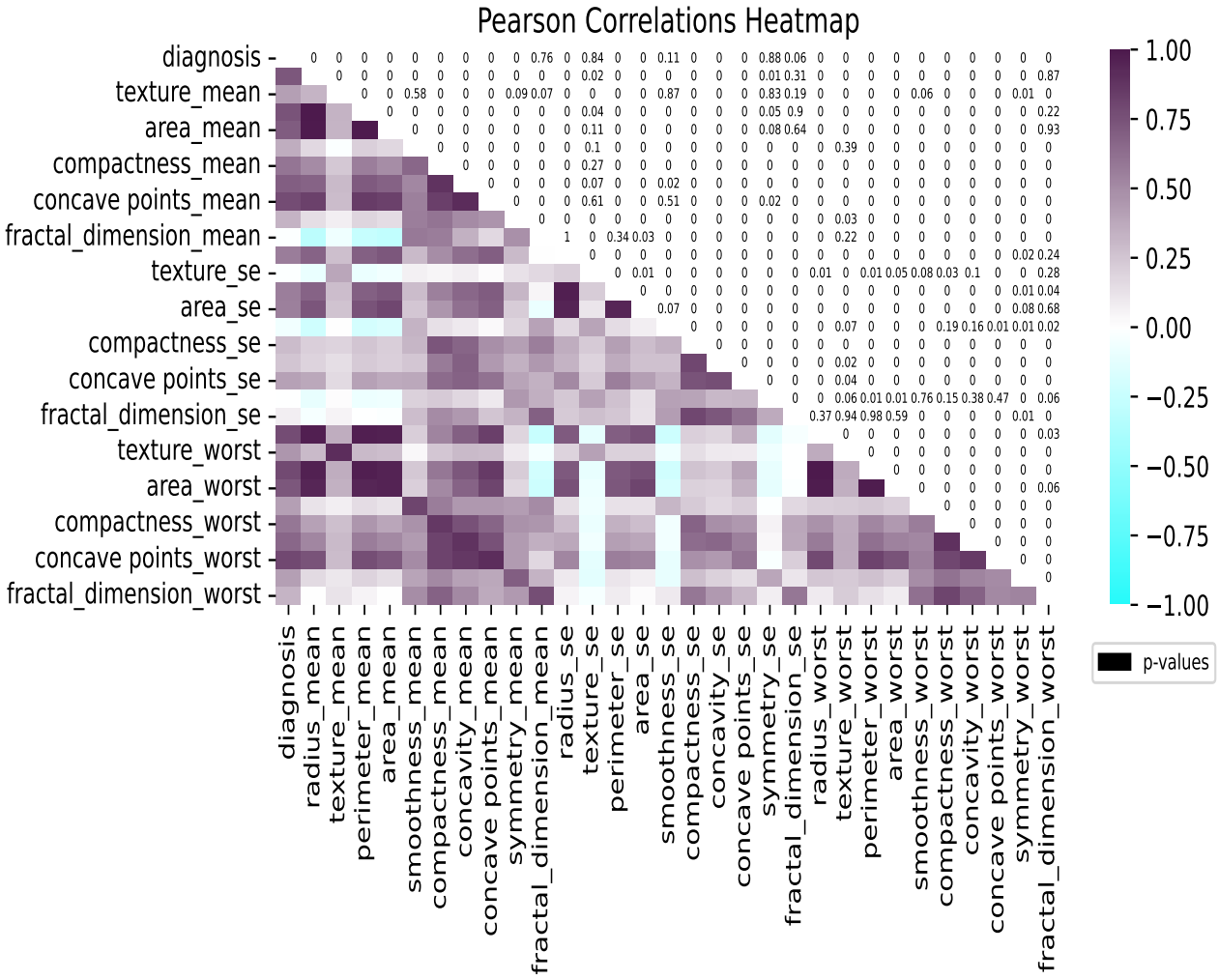


Figure 1.8: Heatmap (Pearson)

```
heatmap_pval(data,
             corr_method='spearman')
```

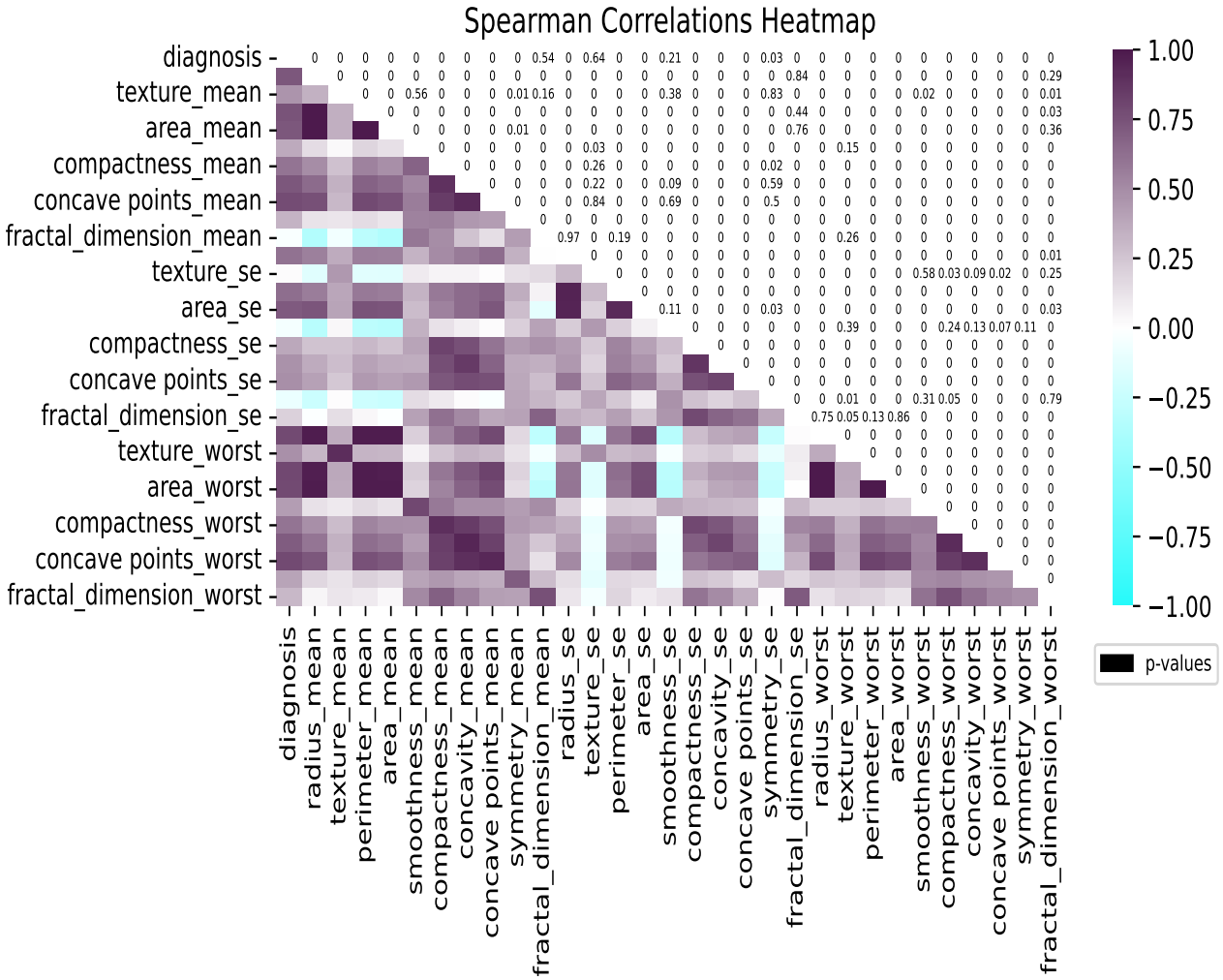


Figure 1.9: Heatmap (Spearman)

1.5 Second Order Terms and Feature Standardization

Standardizing data is an essential step in data analysis. Here are several key benefits of data standardization:

- **Preventing feature dominance:** Standardizing the data ensures that all features have the same scale, which prevents one feature from dominating another in the logistic regression model.
- **Improving convergence:** Standardization helps the logistic regression model converge more quickly and reliably, especially when features have different scales.
- **Better interpretation of coefficients:** Standardizing the data allows the coefficients of the logistic regression model to be compared directly, making it easier to interpret the results.

In the figure shown below, we can observe the effects of standardizing the data.

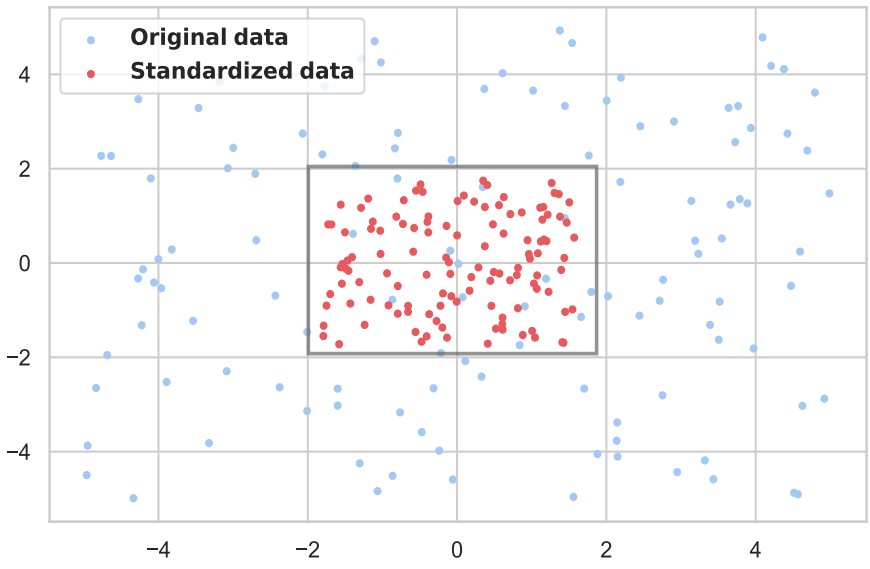


Figure 1.10: Data after standardization

Data standardisation deals with units of measurement by ensuring that all variables in the dataset are on the same scale. This is important for several reasons, including:

- Ensuring that different variables contribute equally to the model, regardless of their original units.
- Improving the convergence of gradient-based optimization algorithms. Facilitating the interpretation of model coefficients.

Let's consider a simple linear model example: We want to predict the price of a house (y) based on its size in square feet (X_1) and the number of bedrooms (X_2). The original data is given in different units: size in square feet and the number of bedrooms as a discrete count.

Linear model: $y = \beta_0 + \beta_1 \cdot X_1 + \beta_2 \cdot X_2 + \epsilon$

Without standardisation, the coefficients (β_1 and β_2) would be influenced by the units of measurement. For instance, a one-unit change in X_1 (square feet) would have a much smaller effect on the price than a one-unit change in X_2 (number of bedrooms), simply due to the difference in the scales of the two variables. This could make it difficult to compare the relative importance of each predictor in the model.
By standardising the data, we ensure that the estimated coefficients do not depend on measurement units. After standardisation, the new linear model becomes:

$$y = \beta_0 + \beta_1 \cdot (X_1 - \mu_1)/\sigma_1 + \beta_2 \cdot (X_2 - \mu_2)/\sigma_2 + \epsilon,$$

where μ_1 and σ_1 are the mean and standard deviation of X_1 and μ_2 and σ_2 are the mean and standard deviation of X_2.
Now, the model coefficients (β_1 and β_2) can be interpreted on a comparable scale and we can better understand the relative importance of each predictor. Furthermore, standardisation can help improve the convergence of gradient-based optimization algorithms, such as stochastic gradient descent, used to estimate the model parameters.

In summary, data standardisation deals with units of measurement in a linear model by transforming the variables to a common scale, ensuring equal contribution of each predictor, facilitating the interpretation of coefficients and improving the performance of optimization algorithms.

1.5.1 Multicollinearity

Multicollinearity is a statistical phenomenon where two or more predictor variables in a regression model exhibit a strong correlation with each other. This interdependence between predictor variables makes it difficult to determine their individual contributions to the outcome variable.

The multicollinearity problem arises when the high correlation between predictor variables leads to unreliable and unstable estimates of regression coefficients. This can make it challenging to interpret the true effect of each predictor variable on the outcome variable and may result in misleading or erroneous conclusions. Additionally, it can cause increased sensitivity to small changes in the data, leading to poor model performance and reduced generalizability. Multicollinearity generates high variance of the estimated coefficients and hence, the coefficient estimates corresponding to those interrelated explanatory variables will not be accurate in giving us the actual picture. They can become very sensitive to small changes in the model. How can we deal with Multicollinearity?

Variance Inflation Factor (VIF) filtering is a technique used in statistics to identify and remove multicollinearity among predictor variables. The VIF algorithm involves fitting a linear regression model for each predictor variable while using all other predictor variables as predictors. The VIF for each predictor variable is then calculated by dividing the variance of its coefficient estimate by the variance of the error term in the linear regression model. The predictor variables with the highest VIF values are then identified and removed from the dataset. This process is repeated until no predictor variables have a VIF value above a specified threshold. The VIF formula is given by:

$$\texttt{VIF} = \frac{1}{1 - R^2}\,.$$

Where $R^2 = 1 - \dfrac{\sum(y - \hat{y})^2}{\sum(y - \bar{y})^2}$. A popular choice for feature filtering retains explanatory variables with a VIF value less than 10. In the subsequent analysis, a less conservative approach is adopted; the analysis utilizes features with a VIF less than 20. To achieve this in R, you can utilize the following code:

```
vif_values <- vif(lm(y ~ ., data = train_data)) #fit a lm

vif_df <- data.frame(Feature = names(vif_values),
                     VIF_value = vif_values[names(vif_values)],
                     Tolerance = 1/vif_values[names(vif_values)])

vif_selected_vars <- c()
j <- 0

for (i in seq_along(vif_values)) {
  vif_val <- vif_values[i]
  if (vif_val < 20) {
    j = j+1
    ### Print the features selected ###
    #print(paste('feature selected', j, ':',
    #            names(vif_values)[i], ":",
    #            round(vif_val, 2)))
    vif_selected_vars[j] <- names(vif_values)[i]
```

```
  }
}

VIF_data <- subset(train_data, select = c('y', vif_selected_vars))

print(VIF_data)
```

```
'y' · 'texture_mean' · 'smoothness_mean' · 'symmetry_mean' ·
'fractal_dimension_mean' · 'texture_se' · 'smoothness_se' · 'concavity_se'
· 'concave.points_se' · 'symmetry_se' · 'fractal_dimension_se' ·
'smoothness_worst' · 'symmetry_worst'
```

In `R`, the package `'nestedcv'` includes the `'collinear'` filter, which can be used to reduce collinearity among predictors.

```
library(nestedcv)

cutoff <- 0.85
collinear(X_train, rsq_cutoff = cutoff, verbose = TRUE)
```

This function identifies predictors with R^2 above a given cut-off and produces an index of predictors to be removed. This method is usefull when we have to deal with aliased coefficients in the model. For example, an error that might occur when trying to compute the `VIF` values is the following:
`Error in vif.default(model) : there are aliased coefficients in the model.`
This error typically occurs when multicollinearity exists in a regression model.

1.5.2 Interaction Terms

In statistics, a second order interaction term is a type of variable that captures the interaction between two predictor variables in a model. Specifically, it represents the combined effect of two variables on the outcome variable that is not accounted for by their individual effects.

To understand this better, let's consider a simple example. Suppose we have a regression model with two predictor variables, X_1 and X_2 and an outcome variable, Y. A second order interaction term would take the form $X_1 \cdot X_2$ and it would capture the unique effect of the combination of X_1 and X_2 on Y.

For instance, suppose X_1 represents a person's age and X_2 represents their level of physical activity and Y represents their risk of developing heart disease. A second order interaction term $(X_1 \cdot X_2)$ in this model would represent the unique effect of the combination of age and physical activity on the risk of developing heart disease. This effect would not be accounted for by the individual effects of age or physical activity on heart disease risk.

Second order interaction terms can be important in modeling complex relationships between predictor variables and the outcome variable. However, including too many interaction terms can lead to overfitting and decreased model performance, so it is important to select only the most important interaction terms to include in a model.

Let's now take a look at an example using the `jtools` package in `R`. `jtools` package is named after its creator, Jacob Long. It is a collection of tools for summarizing and visualizing regression models, designed to simplify common tasks and provide more intuitive output than standard `R` functions. We will create an interaction term which is based on `texture_worst` and `texture_se`.

```
library(jtools)

fit_ <- lm(y ~ texture_worst * texture_se,
           data = data)

summ(fit_)
```

MODEL INFO:
Observations: 569
Dependent Variable: y
Type: OLS linear regression

MODEL FIT:
F(3,565) = 70.21, p = 0.00

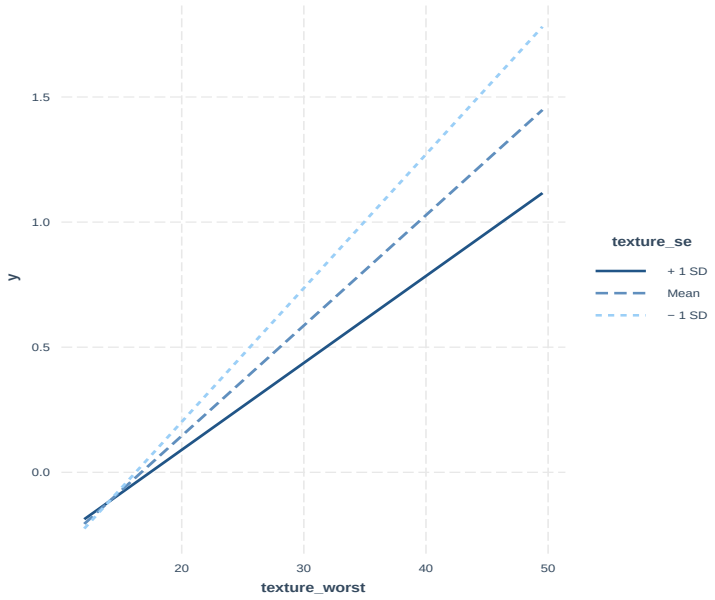
$R^2 = 0.27$
Adj. $R^2 = 0.27$

Standard errors: OLS

	Est.	S.E.	t val.	p
(Intercept)	-1.03	0.16	-6.24	0.00
texture_worst	0.06	0.01	9.80	0.00
texture_se	0.24	0.13	1.88	0.06
texture_worst:texture_se	-0.02	0.00	-3.64	0.00

The figure below presents an interaction plot from a linear regression model. It shows three lines representing the predicted values of y as a function of `texture_worst` at three levels of `texture_se`: one standard deviation below the mean (SD), at the mean, and one standard deviation above the mean.

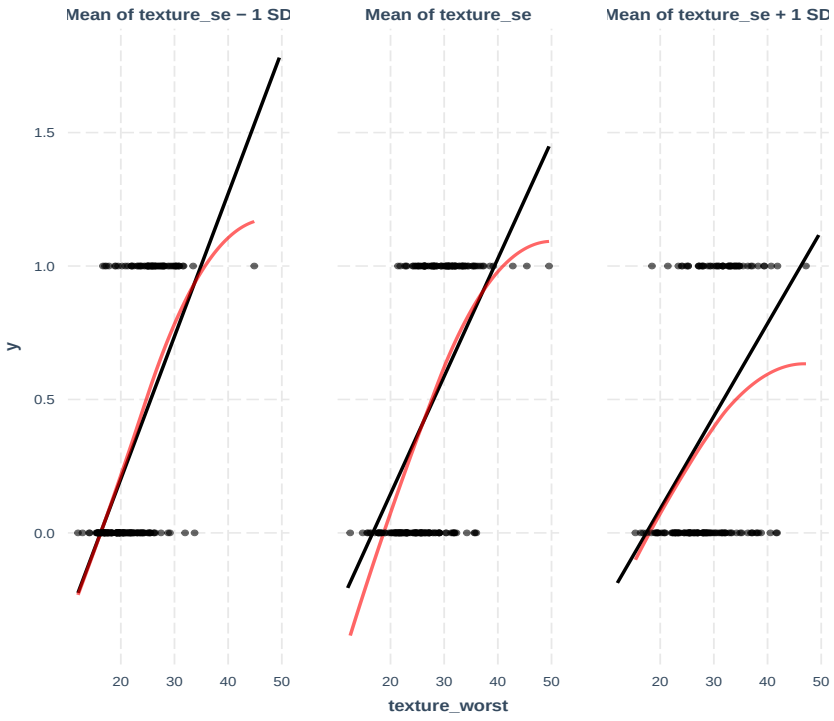
```
interact_plot(fit_,
              pred = texture_worst,
              modx = texture_se)
```



Additionally, the next figure is an enhanced version of the figure above, divided into three subplots. Each subplot corresponds to a specific level of `texture_se` (one standard deviation below the mean, at the mean, and one standard deviation above the mean) and includes actual data points, varying point shapes

based on `texture_se` levels, and a lowess line for checking linearity.

```
interact_plot(fit_, pred = texture_worst, modx = texture_se,
              plot.points = TRUE, point.shape = TRUE,
              linearity.check = TRUE)
```



1.5.3 Interactions with Logistic Regression

There exist several interchangeable terms that refer to statistical interactions. These include, but are not limited to, moderator, modifier, qualifier, magnifier, antagonistic, effect modifier and buffering effect. Interactions entail a cooperative or multiplicative influence that is evaluated by introducing a product variable (XZ) to the model. This implies a non-linear effect that goes beyond the effects of X and Y, which are entered into the model together. The regression coefficient for the product term determines the level of interaction between the two variables. In linear regression, the relationship between X and Y is not uniform across all Z values, which is illustrated by non-parallel slopes on a graph. When slopes are parallel, X affects Y uniformly across all Z values, indicating no interaction. The X and Z variables may be categorical or

continuous. When both X and Z are categorical, the regression model with a continuous response is equivalent to an analysis of variance (ANOVA).

Interactions are similarly specified in logistic regression if the response is binary. The right hand side of the equation includes coefficients for the predictors, X, Z and XZ.

$$\log\left(\frac{\pi}{1-\pi}\right) = a + \beta_1 X + \beta_2 Z + \beta_3 XZ \ .$$

If the interaction coefficient β_3 is significant, we conclude that the association between X and the probability that Y = 1 depends on the values of Z. X and Z may be binary or continuous. For the special case in which X and Z are both binary, then the analysis corresponds with the $2 \times 2 \times 2$ contingency table analysis.

Statistical Tests: The test of the interaction may be conducted with the Wald chi-squared test or a likelihood ratio test comparing models with and without the interaction term. In this particular case, the Wald test appears to perform better than the likelihood ratio test (Allison, 2014). Note that it is always important to have the lower order (main effects) in the model with the interaction variable. Without them, the interaction is potentially confounded with the overall (and additive) effects of the two variables.

Centering: Rescaling the predictors is often recommended (Aiken West, 1991) to improve the interpretation of the lower order effects, β_1 and β_2, sometimes referred to as main effects. Centering or creating deviation scores, which involves subtracting the mean from each predictor's original value, $x = X - \bar{X}$ and $z = Z - \bar{Z}$, before computing the interaction term, $xz = (X - \bar{X})(Z - \bar{Z})$, reduces multicollinearity among the predictors when these new variables are used in the model. The interaction coefficient and its significance are not affected by the rescaling but the standard errors of the main effects are improved because the non-essential multicollinearity is reduced. When coding binary predictors, two methods are commonly used: dummy coding (0 and 1) or centering. Even though centering binary variables may appear strange, it can lead to a more desirable interpretation of the intercept when X and Z are at their mean. This is because the interpretation of their values at 0 (i.e., the logit for Y for the 0 group) can be problematic. However, when dealing with a set of $g - 1$ dummy variables for three or more groups, centering in the usual manner may not make sense, as the set of dummy variables must be orthogonal and considered together. In such cases, effect coding (refer to Hardy, 1993, for effect coding systems) can be considered for achieving a centering interpretation of the intercept and lower order effects, if needed.

The code below fits a `glm` model with a binomial family, as demonstrated in the above example.

```
log_model <- glm(y~texture_worst * texture_se,
                 data=data,
                 family=binomial(link="logit"))
summ(log_model)
```

MODEL INFO:
Observations: 569
Dependent Variable: y
Type: Generalized linear model
Family: binomial
Link function: logit

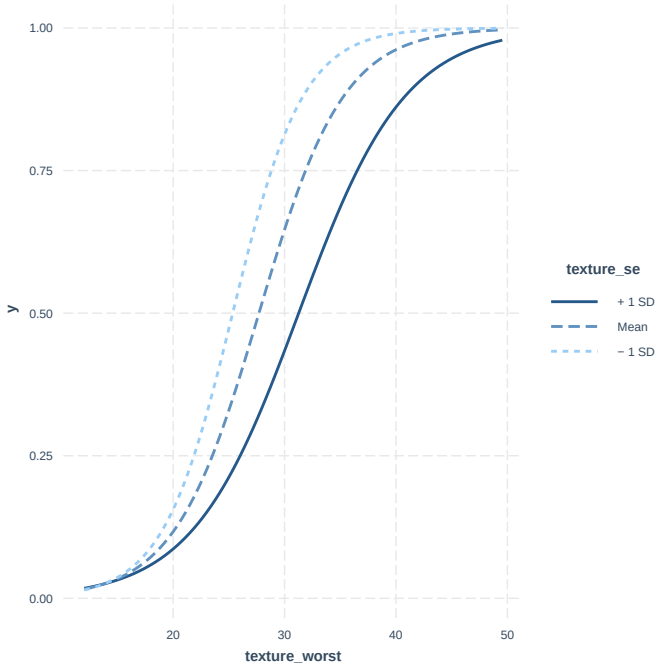
MODEL FIT:
$\chi^2(3) = 177.76$, p = 0.00
Pseudo-R^2 (Cragg-Uhler) = 0.37
Pseudo-R^2 (McFadden) = 0.24
AIC = 581.68, BIC = 599.06

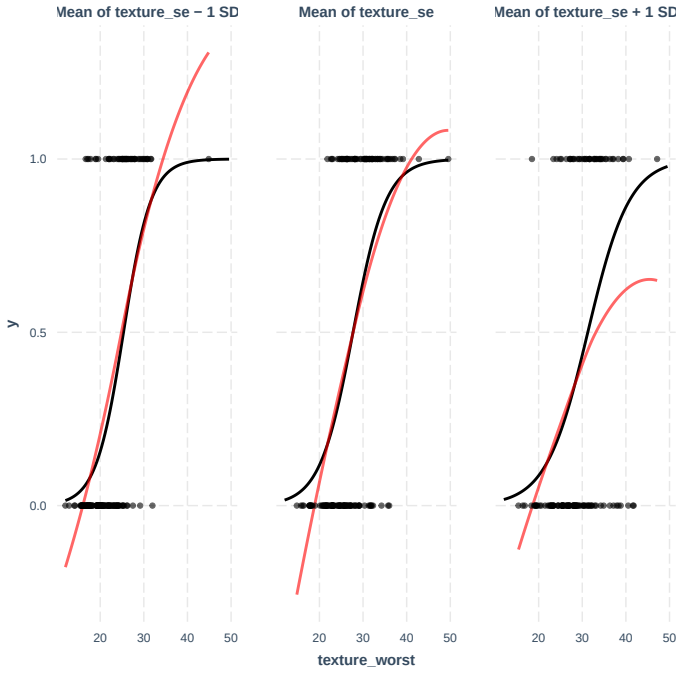
Standard errors: MLE

	Est.	S.E.	z val.	p
(Intercept)	-9.16	1.41	-6.48	0.00
texture_worst	0.39	0.05	7.14	0.00
texture_se	1.68	1.07	1.57	0.12
texture_worst:texture_se	-0.11	0.04	-2.89	0.00

```
interact_plot(log_model,
              pred = texture_worst,
              modx = texture_se)

interact_plot(log_model, pred = texture_worst, modx = texture_se,
              plot.points = TRUE, point.shape = TRUE,
              linearity.check = TRUE)
```

The figure below presents an interaction plot from a binomial logistic model. It illustrates three sigmoid-like curves showing how the log-odds of y being 1 change with `texture_worst` at three levels of `texture_se`: one standard deviation below the mean, at the mean, and one standard deviation above the mean. Furthermore, The subsequent figure enhances the previous one by dividing the interaction plot into three distinct subplots. Each subplot corresponds to a specific level of `texture_se` (one standard deviation below the mean, at the mean, and one standard deviation above the mean) and includes actual data points, varying point shapes based on `texture_se` levels, and a lowess line for checking the linearity of the logit-transformed probabilities.


1.00
0.75
0.50
0.25
0.00
y
20
30
40
50
texture_worst
texture_se
+ 1 SD
Mean
− 1 SD


Mean of texture_se − 1 SD
Mean of texture_se
Mean of texture_se + 1 SD
1.0
0.5
0.0
y
20
30
40
50
texture_worst

1.6 Classification Metrics

There exists a diverse range of classification metrics, including binary classification metrics, that offer more meaningful alternatives to the conventional `'accuracy'` measure. The selection of an appropriate metric is crucial for accurately assessing model performance. Let's explore a few of these metrics.

1.6.1 Receiver Operating Characteristic Curves (ROC) & Area Under Curve (AUC)

`ROC curve` (Receiver Operating Characteristic curve) is a graphical representation used to evaluate the performance of a binary classification model. The curve plots the true positive rate (sensitivity or recall) against the false positive rate (1−specificity) for different classification thresholds. The true positive rate is the proportion of actual positive instances that are correctly identified by the model, while the false positive rate is the proportion of actual negative instances that are incorrectly identified as positive.

`ROC AUC` (Area Under the ROC Curve) is a scalar value derived from the ROC curve, which quantifies the overall performance of the binary classifier. The AUC score ranges from 0 to 1, where a value of 0.5 indicates a random classifier and a value of 1 indicates a perfect classifier. The closer the AUC is to 1, the better the model is at distinguishing between the positive and negative classes.

- Imbalanced data: ROC AUC is a preferable metric compared to accuracy, especially when dealing with imbalanced datasets. Accuracy can be misleading in such cases, as a model that always predicts the majority class will have high accuracy but may not be useful in practice. ROC AUC accounts for both true positive and false positive rates, providing a better indication of the model's performance.

- Threshold independence: The ROC curve helps visualize the trade-off between sensitivity and specificity for different threshold values. This makes it possible to select an optimal threshold based on the problem's requirements, such as minimizing false positives or maximizing true positives.

- Model comparison: ROC AUC allows for easy comparison of different classification models, regardless of their underlying algorithms. A higher AUC indicates a better-performing model, which makes it simple to select the best one among multiple options.

In summary, the ROC curve and ROC AUC provide a more comprehensive and interpretable evaluation of binary classification models, particularly when dealing with imbalanced datasets or when selecting an optimal decision threshold. This makes them preferable to accuracy as a performance metric in many situations.

How to calculate the threshold that leads to the best accuracy?

$$\texttt{Accuracy} = \texttt{sensitivity} \cdot \texttt{Prob(+class)} + \texttt{specificity} \cdot \texttt{Prob(-class)}$$

For every threshold T, that has a specific sensitivity and specificity, calculate the accuracy. `Prob(+class)` and `Prob(-class)` will be given. After that, simply choose the threshold that led to the maximum accuracy.

How to calculate AUC 'by hand'? ROC AUC can be approximated using the following formula:

$$\textbf{AUC} \approx \frac{\#\{\texttt{score}(p) > \texttt{score}(n), \text{ for ever pair where } p \text{ is Pos. and } n \text{ is Neg. sample}\}}{\#\text{ pairs}}$$

1.6.2 Implementation of ROC Curve in Python

We have developed a function called `'Roc_Auc'` capable of plotting the ROC curve of a model given the predicted data and the observed data. The function includes various features, such as identifying the threshold point with maximum accuracy based on the prior probability of each class and calculating the `AUC score`, among others. Additionally, the `'mRoc_Auc'` method concats `'Roc_Auc'` objects together in the same plot.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_auc_score

norm01 = lambda x: (x-np.min(x))/(np.max(x)-np.min(x))

# The following functions are implemented by me,
# It plots the ROC Curve given score and true values
# possible features: AUC score, confusion matrix, max accuracy
    threshold ROC (...)

def Roc_Auc(score: list, true_class: list, #score in [0,1]
            thresholds: int = 45,
            info: bool = False,
            show_ROC: bool = True,
            show_points: bool = False,
            point_size: float = 6,
            save: str = None, #save as pdf
            POSclassProb: float = 0.5,
            title: str = '$\mathbf{ROC}$ curve ' +\
            'on $\mathbf{Hold}$-$\mathbf{Out}$-$\mathbf{Set}$',
            get_max_acc: bool = False,
            norm_score: bool = True,
            model_name: str = '',
            return_: bool = True,
```

```
          fontsize: str = '10.5') -> plt.plot or tuple:

    if norm_score: score = norm01(score)

    threshold = sorted(list(np.linspace(0,1,thresholds)),
        reverse=True)
    pred_given_threshold = lambda scr, t: list(map(lambda x: 1
        if t<=x else 0, scr))

    predictions_t = np.array([pred_given_threshold(score, t)
        for t in threshold])

    mult_true_class = np.array(true_class*len(threshold)).
        reshape(len(threshold),-1)

    tuplethem = lambda P,R: [(pred,real) for (pred,real) in zip
        (P,R)]
    true_pred_pairs = [tuplethem(p,r) for p,r in zip(
        predictions_t, mult_true_class)]

    TP = lambda D: D.count((1,1))
    TN = lambda D: D.count((0,0))
    FN = lambda D: D.count((0,1))
    FP = lambda D: D.count((1,0))

    ##simply choice: the threshold that led to the max accuracy
    NEGclassProb = 1 - POSclassProb
    roc_points = [] ; Accuracies = []

    for i in range(len(threshold)):
        f = lambda X: list(map(X, true_pred_pairs))
        tp = f(TP); tn = f(TN)
        fn = f(FN); fp = f(FP)
        TPR = tp[i]/(tp[i] + fn[i]); FPR = fp[i]/(fp[i] + tn[i
            ])

        roc_points.append((FPR, TPR))

        conf_matrix = pd.DataFrame({
            'Pos': [tp[i], fn[i]],
            'Neg': [fp[i], tn[i]]
        }).T
        conf_matrix.columns = ['PP', 'PN']

        if info:
            #PP: predicted positive, PN: predicted negative
            print(f'\nfor threshold T={round(threshold[i], 3)}:
                ')
            print(f'TP = {tp[i]}')
            print(f'TN = {tn[i]}')
```

```
            print(f'FN = {fn[i]}')
            print(f'FP = {fp[i]}')
            print(f'sensitivity = TPR = {round(TPR, 4)}')
            print(f'1-specifisity = FPR = {round(FPR, 4)}')
            acc_str = f'Accuracy =\
            {round(TPR*POSclassProb + (1-FPR)*NEGclassProb, 4)
                }\n'
            print(" ".join(acc_str.split()))
            print(f'\nConfusion  Matrix:\n\n {conf_matrix}\n')
            print(20*'___', '\n\n')

        Accuracies.append((TPR*POSclassProb + (1-FPR)*
            NEGclassProb,
                           threshold[i]))

    roc_points.sort() #in order to calc the auc later on

    max_acc = max(dict(Accuracies).keys())

    max_acc_threshold = threshold.index(dict(Accuracies)[
        max_acc])
    acc_point = tuple(map(lambda x: round(x,3),
                      roc_points[max_acc_threshold]))
    if info:
        print(" ".join(f'max accuracy = {round(max_acc, 4)}\
        for threshold T = {round(dict(Accuracies)[max_acc], 4)}
            '.split()))
        print('point that maximizes accuracy: (1-spe, sens): ',
              acc_point)

    x = list(map(lambda x: x[0], roc_points))
    y = list(map(lambda x: x[1], roc_points))

    #AUC = auc(x,y) #one way
    AUC = roc_auc_score(true_class, score) #better way

    if not return_:
        label = model_name + ' $\mathbf{AUC}$: ' + f'{AUC:.3f}'

        return roc_points, AUC, label

    mark = '-'
    if show_ROC:
        mark = '.-'
        plt.plot(np.linspace(0,1,70),np.linspace(0,1,70),'--',
                 label='$\mathbf{Random}$ cls', alpha=0.3,
                 color='orange') # random classifier
        plt.plot(x,y,mark, color=(0.25,0.65,1), markersize=
            point_size,
                 label=model_name + '$\mathbf{ROC}$')
```

```
        plt.fill_between(x,y, color='lightpink', alpha=0.1,
                         label='$\mathbf{AUC}$='+f'{AUC:.3f}')
        #view the points
        fix_number = lambda k: int(k) if k==int(k) else k
        if show_points:
            for i in range(len(roc_points)):
                plt.text(roc_points[i][0]-.034, roc_points[i
                    ][1]+.03,
                    " ".join(f'({fix_number(round(roc_points[i
                        ][0], 3))},\
                    {fix_number(round(roc_points[i][1],3))})'.split
                        ()),
                    fontsize=6.9)
        if get_max_acc:
            x_acc, y_acc = acc_point[0], acc_point[1]
            plt.text(0.2,0.13,
                     f'(1-spe, sens) that maximizes accuracy:',
                     fontsize=7.5)
            plt.text(0.335,0.08, f'({x_acc},{y_acc})',
                     fontsize=7.6)
            plt.text(0.3,0.03, f'At threshold: {round(dict(
                Accuracies)[max_acc],4)}',
                     fontsize=7.6)
            plt.scatter(x_acc, y_acc, color='green',
                        marker='o', s=63, label="$\mathbf{max}$
                            Acc.")

        #plt.text(0.56,0.456,f'AUC = {AUC:.3f}', fontsize=12)
        plt.title(title, size=11)
        plt.xlabel('$1-$specifisity ($\mathbf{FPR}$)', size=9)
        plt.ylabel('sensitivity ($\mathbf{TPR}$)', size=9)
        plt.legend(loc="best", fontsize=fontsize)

        if save!=None: plt.savefig(str(save)+".pdf", format='
            pdf',
                                   bbox_inches='tight')
        plt.show()

### The following function merges ROC plots ###

def mRoc_Auc(*roc_auc_curves,
             show_ROC: bool = True,
             point_size: float = 6,
             save: bool = False,
             title: str = " ".join('$\mathbf{ROC}$ curves\
                        on $\mathbf{Hold}$ - $\mathbf{Out}$\
                        - $\mathbf{Set}$'.split()),
             fontsize: str = '9.5',
             average_aucs: bool = False,
             comment: str = '',
```

```
            pink_col: float = 0.015) -> plt.plot:
    avg_aucs = []
    plt.plot(np.linspace(0, 1, 70),
             np.linspace(0, 1, 70), '--',
             label='$\mathbf{Random}$ classifier', alpha=0.5,
             color='orange')  # random classifier

    for index, (roc_points, AUC, label) in enumerate(
        roc_auc_curves):
        x = list(map(lambda x: x[0], roc_points))
        y = list(map(lambda x: x[1], roc_points))
        avg_aucs.append(AUC)
        plt.plot(x, y, '.-',
                 markersize=point_size,
                 label=label)
        plt.fill_between(x,y, color='lightpink',
                         alpha=pink_col) #0.01)
    if average_aucs:
        plt.fill_between([],[], color='lightpink',
                         label=f'Average AUC = {np.mean(
                             avg_aucs):.2}',
                         alpha=0.01)
    plt.title(title + comment, size=11)
    plt.xlabel('1-specifisity $\mathbf{(FPR)}$', size=9)
    plt.ylabel('sensitivity $\mathbf{(TPR)}$', size=9)
    plt.legend(loc="best", fontsize=fontsize)

    ### Save the plots as .pdf file ###
    if save:
        plt.savefig(str(save) + ".pdf", format='pdf',
                    bbox_inches='tight')
    plt.show()
```

We can use the above functions in `R`. The `R` package `'reticulate'` provides a bridge between R and Python, allowing users to interoperate between these two programming languages. The reticulated python is a species of python found in Southeast Asia. They are the world's longest snakes and longest reptiles. The specific name, reticulatus, is Latin meaning "net-like", or reticulated and is a reference to the complex colour pattern. To access the Python code mentioned above in `R`, we can simply perform the following steps:

```
library(reticulate)

python_file = "ROC_CURVE.py"
source_python(python_file)
```

1.6.3 Recall-Precision & F1-score

Precision and recall are two evaluation metrics commonly used in classification tasks, particularly for binary classification. They provide insights into the performance of a model by examining its ability to correctly identify true positive instances while minimizing false positives and false negatives.

- Precision (also known as Positive Predictive Value): Precision measures the proportion of correctly predicted positive instances out of the total instances predicted as positive. In other words, it evaluates how many of the instances that the model identified as positive are actually positive.

Precision is calculated as:

```
Precision = (True Positives) / (True Positives + False Positives)
```

A high precision indicates that the model has a low rate of false positives, meaning it is good at correctly identifying positive instances and not mislabeling negative instances as positive.

- Recall (also known as Sensitivity or True Positive Rate): Recall measures the proportion of correctly predicted positive instances out of the total actual positive instances. It evaluates how well the model identifies all the positive instances in the dataset.

Recall is calculated as:

```
Recall = (True Positives) / (True Positives + False Negatives)
```

A high recall indicates that the model has a low rate of false negatives, meaning it is good at capturing all the positive instances and not missing any. Additionally, balanced accuracy, useful for imbalanced datasets, is the mean of `TPR` and `TNR`, calculated as:

$$\texttt{Balanced Accuracy} = \texttt{(Sensitivity + Specificity)/2}$$

Both precision and recall are important metrics and their relative importance depends on the specific problem being addressed. In some cases, a high precision might be more critical, such as in spam email filtering, where it's essential not to mislabel non-spam emails as spam. In other cases, a high recall might be more important, such as in medical diagnostics, where it's crucial to identify all positive cases of a disease to avoid missing any patients in need of treatment.

The `F1`-score is another metric that combines both precision and recall into a single value, providing a balance between them. It is the harmonic mean of precision and recall:

$$\texttt{F1-score} = 2 \cdot (\texttt{Precision} \cdot \texttt{Recall})/(\texttt{Precision} + \texttt{Recall})$$

The `F1`-score ranges from 0 to 1, where a higher value indicates better performance. It is particularly useful when dealing with imbalanced datasets or when both precision and recall are important for a specific problem.

1.6.4 Matthews Correlation Coefficient

The Matthews Correlation Coefficient (MCC) is a performance metric for binary classification problems that measures the quality of a model's predictions by considering true and false positives, as well as true and false negatives. It is particularly useful for imbalanced datasets, as it provides a balanced measure of classification performance. MCC ranges from -1 (completely inverse prediction) to 1 (perfect prediction), with 0 representing random predictions.

$$\mathtt{MCC} = \frac{(\mathtt{TP} \times \mathtt{TN} - \mathtt{FP} \times \mathtt{FN})}{\sqrt{(\mathtt{TP} + \mathtt{FP})(\mathtt{TP} + \mathtt{FN})(\mathtt{TN} + \mathtt{FP})(\mathtt{TN} + \mathtt{FN})}}$$

1.6.5 Type I & Type II Error

Type I and Type II errors are terms used in hypothesis testing to describe incorrect conclusions drawn from a statistical test. These errors are related to false positives and false negatives in binary classification problems, which can be linked to the metrics of precision and recall.

- Type I Error (False Positive): A Type I error occurs when the null hypothesis is true, but it is incorrectly rejected. In the context of binary classification, a Type I error corresponds to a false positive, where an instance is wrongly classified as positive when it is actually negative. Type I error is related to the concept of precision, as a high rate of false positives will result in a lower precision value for the classifier.
- Type II Error (False Negative): A Type II error occurs when the null hypothesis is false, but it is incorrectly accepted. In the context of binary classification, a Type II error corresponds to a false negative, where an instance is wrongly classified as negative when it is actually positive. Type II error is related to the concept of recall, as a high rate of false negatives will result in a lower recall value for the classifier.

Both Type I and Type II errors are important to consider when evaluating the performance of a classifier, as they provide insights into the model's ability to correctly identify instances while minimizing misclassification. The relative importance of minimizing Type I and Type II errors depends on the specific problem being addressed. In some cases, minimizing Type I errors might be more important, while in other cases, minimizing Type II errors might be more critical.
To summarize, Type I errors are related to false positives and precision, while Type II errors are related to false negatives and recall. Understanding the implications of these errors is essential in selecting an appropriate classification model for a given problem.

2

Predictive Models

2.1 Regression Models

In this section, we will be discussing about regression models and how they can be applied to a classification problem. We will take a look at general linear models from `R` (`glmnet`), how to tune their hyperparameters using cross-validation and their implementation using R programming. To speed up computations, we will be using the `doMC` package from `R`, which serves as a parallel backend for the `foreach` package. This allows us to run `foreach` loops in parallel, reducing computation time. The `registerDoMC(cores)` function is used to register the multicore parallel backend with the foreach package.

However, it is important to note that the multicore functionality of `doMC` is only compatible with operating systems that support the fork system call and can only run tasks on a single computer, not a cluster of computers. Thus, to see a performance improvement, we need to use `doMC` on a machine with multiple processors, multiple cores, or both.

2.1.1 Ordinary Least Squares using Normal Equations

OLS regression via normal equations involves finding the line of best fit that minimizes the sum of the squared differences between the observed data points and the predicted values of the dependent variable, based on the values of the independent variables. It is a widely used method in fields such as finance, econometrics and other related areas. Let's take a closer look at the mathematics of OLS.

Suppose the data consists of n observations $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$. Each observation i includes a scalar response y_i and a column vector $\mathbf{x}_i$ of p parameters (regres-

sors), i.e.,

$$\mathbf{x}_i = [x_{i1}, x_{i2}, \ldots, x_{ip}]^T.$$

In a linear regression model, the response variable y_i is a linear function of the regressors:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \epsilon_i,$$

where β_0 is the intercept, $\beta_1, \beta_2, \ldots, \beta_p$ are the coefficients of the regressors and ε_i is the error term for observation i. This model can also be written in matrix notation as:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon,$$

where $\mathbf{y}$ and $\boldsymbol{\epsilon}$ are $n \times 1$ vectors of the response variables and the errors of the n observations and $\mathbf{X}$ is an $n \times p$ matrix of regressors.

Consider an overdetermined system:

$$\sum_{j=1}^{p} X_{ij}\beta_j = y_i, \quad i = 1, 2, \ldots, n$$

of n linear equations and p unknown coefficients, $n > p$. This can be written in matrix form as:

$$\mathbf{X}\beta = \mathbf{y},$$

Finding an exact solution for such a system is typically not possible. Therefore, the focus is on identifying the coefficients $\boldsymbol{\beta}$ that provide the best possible fit for the equations. This is accomplished by solving a problem involving the minimization of a quadratic function:

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}}\ S(\beta)$$

where the objective function S is given by:

$$S(\beta) = \sum_{i=1}^{n} \left| y_i - \sum_{j=1}^{p} X_{ij}\beta_j \right|^2 = \|\mathbf{y} - \mathbf{X}\beta\|_2^2$$

To minimize the objective function, we differentiate for each parameter and set the resulting derivative to zero. As a result, we obtain the following estimation:

$$\hat{\beta} = \left(\mathbf{X^T X}\right)^{-1}\mathbf{X^T y},$$

or

$$\hat{\beta} = \beta + \left(\mathbf{X^T X}\right)^{-1}\mathbf{X^T y}\ \epsilon.$$

The `lm` function in `R` uses `QR`[1] decomposition, which is numerically stable.

[1] QR decomposition breaks down a matrix into a product of two simpler matrices: an orthogonal matrix and an upper triangular matrix

The results shown below, indicate that the full model explains about 77% of the variability of the response. The `p-value` of the F-statistic provides strong evidence against the null hypothesis that a model that contains only an intercept term is sufficient. Notably, the only feature for which we have strong indication against the null hypothesis of a zero coefficient is `smoothness_se`.

```
full_lm.model <- lm(y ~. , data = train_data)

summary(full_lm.model)
```

Call:
`lm(formula = y ~ ., data = train_data)`

Residuals:

Min	1Q	Median	3Q	Max
-0.59854	-0.16392	-0.02983	0.14983	0.90751

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-1.864e+00	4.863e-01	-3.832	0.000146 ***
radius_mean	-1.349e-01	2.015e-01	-0.669	0.503669
texture_mean	-2.716e-03	9.178e-03	-0.296	0.767439
perimeter_mean	8.635e-03	2.823e-02	0.306	0.759857
area_mean	3.026e-04	6.190e-04	0.489	0.625245
smoothness_mean	1.205e+00	2.260e+00	0.533	0.594297
compactness_mean	-3.207e+00	1.518e+00	-2.112	0.035280 *
concavity_mean	6.105e-01	1.199e+00	0.509	0.611001
concave.points_mean	4.196e+00	2.234e+00	1.878	0.061104 .
symmetry_mean	3.966e-01	8.816e-01	0.450	0.653081
fractal_dimension_mean	-3.923e+00	6.370e+00	-0.616	0.538255
radius_se	1.975e-01	3.672e-01	0.538	0.590952
texture_se	-5.032e-02	4.733e-02	-1.063	0.288339
perimeter_se	-1.262e-02	4.678e-02	-0.270	0.787470
area_se	-1.925e-04	1.697e-03	-0.113	0.909735
smoothness_se	2.272e+01	7.727e+00	2.941	0.003451 **
compactness_se	2.845e+00	3.185e+00	0.893	0.372286
concavity_se	-3.285e+00	1.810e+00	-1.815	0.070169 .
concave.points_se	5.478e+00	6.824e+00	0.803	0.422527
symmetry_se	3.407e+00	3.313e+00	1.028	0.304369
fractal_dimension_se	-1.052e+01	1.540e+01	-0.683	0.495115
radius_worst	1.814e-01	7.164e-02	2.533	0.011682 *
texture_worst	1.515e-02	8.303e-03	1.825	0.068773 .
perimeter_worst	1.248e-03	6.847e-03	0.182	0.855429
area_worst	-1.025e-03	4.023e-04	-2.549	0.011166 *
smoothness_worst	-9.508e-01	1.636e+00	-0.581	0.561519
compactness_worst	-3.578e-01	5.034e-01	-0.711	0.477608
concavity_worst	4.174e-01	3.413e-01	1.223	0.222046
concave.points_worst	7.653e-01	1.089e+00	0.702	0.482779
symmetry_worst	4.169e-01	5.833e-01	0.715	0.475195
fractal_dimension_worst	5.521e+00	2.839e+00	1.945	0.052461

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.2398 on 425 degrees of freedom
Multiple R-squared: 0.7708, Adjusted R-squared: 0.7546
F-statistic: 47.64 on 30 and 425 DF, p-value: < 2.2e-16

The same analysis was performed on the VIF-filtered features. In the output shown below, one observes that the number of significant features increases dramatically. Thus, the model has more to offer in terms of interpretability. However, one also observes that the R-squared value dropped significantly to approximately 59%. This suggests that the VIF threshold could have been larger than 20 to allow for additional explanatory variables.

```
VIF_lm.model <- lm(y ~. , data = VIF_data)

summary(VIF_lm.model)
```

Call:
`lm(formula = y ~ ., data = train_data)`

Residuals:

Min	1Q	Median	3Q	Max
-0.73796	-0.23663	0.00285	0.24853	0.92599

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-4.1009e-01	1.9327e-01	-2.122	0.034401 *
texture_mean	3.2946e-02	4.2090e-03	7.827	3.71e-14 ***
smoothness_mean	6.2703e+00	2.4573e+00	2.552	0.011054 *
symmetry_mean	2.0592e-01	9.9479e-01	0.207	0.836109
fractal_dimension_mean	-3.2690e+01	4.0180e+00	-8.136	4.16e-15 ***
texture_se	-4.8491e-02	3.8329e-02	-1.265	0.206494
smoothness_se	-2.2067e+01	8.3113e+00	-2.655	0.008216 **
concavity_se	-6.1605e-01	1.0016e+00	-0.615	0.538816
concave.points_se	3.0660e+01	4.7078e+00	6.513	2.01e-10 ***
symmetry_se	-6.2196e+00	3.3265e+00	-1.870	0.062179 .
fractal_dimension_se	3.9706e+01	1.1091e+01	3.580	0.000381 ***
smoothness_worst	5.8585e+00	1.7396e+00	3.368	0.000824 ***
symmetry_worst	2.1770e+00	5.6061e-01	3.883	0.000119 ***

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3137 on 443 degrees of freedom
Multiple R-squared: 0.5911, Adjusted R-squared: 0.5801
F-statistic: 53.38 on 12 and 443 DF, p-value: < 2.2e-16

In `R`, the `'plot'` method provides us with four types of diagnostic plots to

assess the quality of our statistical models.

The "Residuals vs Fitted" plot helps us visualize the difference between the observed and predicted values against the predicted values. A random scatter indicates a good model fit. The "Normal Q-Q" plot compares the distribution of residuals to a standard normal distribution. If the points lie along a diagonal line, it suggests the residuals are normally distributed. The "Scale-Location" plot, also known as the Spread-Location plot, shows the spread of residuals and is used to check if the residuals are spread equally along the ranges of predictors. Finally, the "Residuals vs Leverage" plot helps us identify influential cases, if any, that might be unduly influencing the regression results.

```
par(mfrow=c(2,4))
plot(full_lm.model, main="(Full OLS model)")
plot(vif_lm.model, main="(VIF OLS model)")
```

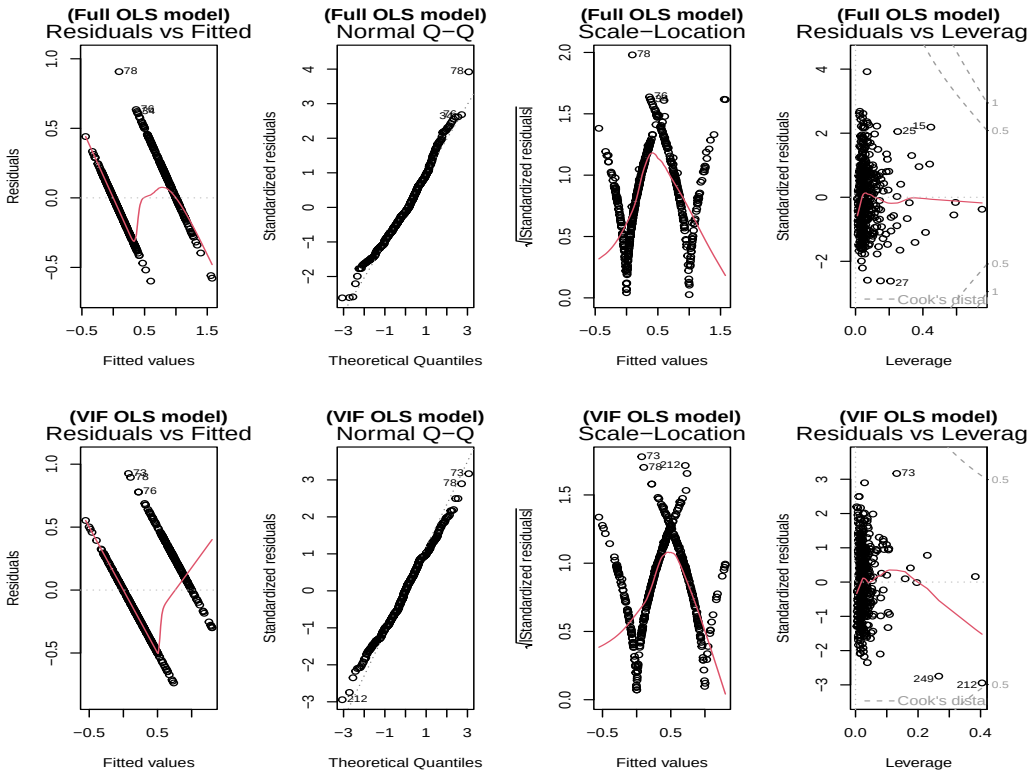
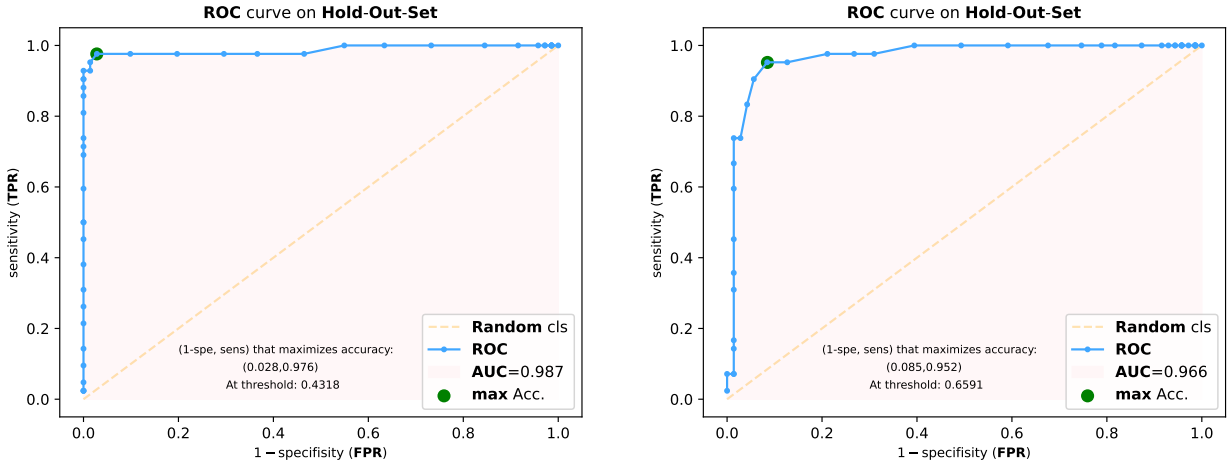


Figure 2.1: OLS residuals diagnostics

Now, we will utilize the functions developed in the previous chapter to plot the ROC curve.

```
y_real <- y_hold_out
ols_scores <- predict(object = full_lm.model,
                      newdata = hold_out_data)

ols_scores_vif <- predict(object = vif_lm.model,
                          newdata = hold_out_data)
#Assuming prior prob of positive class
prob_pos = sum(data$y == 1)/dim(data)[1]

Roc_Auc(score=ols_scores,
        true_class=y_real,
        POSclassProb=prob_pos, #0.3725
        get_max_acc=TRUE)
Roc_Auc(score=ols_scores_vif,
        true_class=y_real,
        POSclassProb=prob_pos,
        get_max_acc=TRUE)
```

The figure shown below depicts ROC curves for the full and the VIF-filtered models. The decision threshold that achieves maximum accuracy is shown with a green dot. In accordance with prior expectations (since the response variable is close to being balanced), the optimal thresholds are close to 0.5.



(a) `Full model`

(b) `VIF model`

Figure 2.2: `OLS` ROC curves

2.1.2 Ordinary Least Squares using SVD Factorization

An alternative approach to solving OLS regression is through the utilization of SVD Factorization. This method offers a numerically stable and computationally efficient way to obtain the pseudoinverse of a matrix. The pseudoinverse is used to solve the normal equations in OLS regression and can be computed using the singular value decomposition of the matrix. This approach avoids the numerical instability and computational inefficiencies that can arise when directly inverting a matrix. Additionally, SVD can help identify and address issues such as multicollinearity in OLS regression.
Singular Value Decomposition (SVD) of an $m \times n$ matrix $\mathbf{X}$ has form:

$$\mathbf{X} = U\Sigma V^T,$$

where U is $m \times m$ orthogonal matrix, V is $n \times n$ orthogonal matrix and Σ is $m \times n$ diagonal matrix, with:

$$\Sigma_{ij} = \begin{cases} 0, & i \neq j \\ \sigma_i, & i = j \end{cases},$$

Diagonal entries σ_i, called **singular values** of $\mathbf{X}$, are usually ordered such that $\sigma_1 > \sigma_2 > \cdots > \sigma_n$.
The columns u_i of U and v_i of V are called left and right **singular vectors**.

The vector

$$\hat{\beta} = \sum_{\sigma_i \neq 0} \frac{u_i^T \mathbf{y}}{\sigma_i} v_i$$

minimizes the norm:

$$\|\mathbf{y} - \mathbf{X}\beta\|_2,$$

if $\text{rank}(\mathbf{X}) = n$, then the solution is unique. For ill-conditioned or rank deficient problems, "small" singular values can be omitted from summation to stabilize solution. SVD can also handle the rank deficient case, if there are only k singular values $\sigma_j > \epsilon$ then take only the first k contributions.

The **natural generalized inverse** (or pseudoinverse) of general real $m \times n$ matrix $\mathbf{X}$ is given by:

$$\mathbf{X}^{-g} = V_p \Sigma_p^{-1} U_p^T,$$

where U_p, V_p consist of the first p columns of U , V and Σ_p is the block diagonal matrix with all the non-zero singular values in the diagonal.

Hence, **minimum-norm** solution (**natural solution**) is given by

$$\hat{\beta} = \mathbf{X}^{-g}\mathbf{y}.$$

Let's now take a look at the `R` implementation of the above[2]:

[2]This is a modified version of the code that can be found *here*.

```
svdOLS <- function (X, y) {
  SVD <- svd(X)
  V <- SVD$v
  U <- SVD$u
  D <- SVD$d
  ## regression coefficients `b`
  ## use `crossprod` for `U'y`
  ## use recycling rule for row rescaling of `U'y` by `D` inverse
  ## use `as.numeric` to return vector instead of matrix
  b <- as.numeric(V %*% (crossprod(U, y) / D))
  ## residuals
  r <- as.numeric(y - X %*% b)
  ## R-squared
  RSS <- crossprod(r)[1]
  TSS <- crossprod(y - mean(y))[1]
  R2 <- 1 - RSS / TSS
  ## multiple return via a list
  list(coefficients = b, residuals = r, R2 = R2)
  }
```

The code below fits an OLS model using the SVD and plots the ROC curve for each model (Full model and VIF-filtered).

```
svd_fit <- svdOLS(as.matrix(X_train), y_train)
svd_fit_vif <- svdOLS(as.matrix(VIF_data[,-1]),
                      VIF_data$y)

predict_svd_ols <- function(data, svd_fit=svd_fit, binary=FALSE) {
  n <- nrow(hold_out_data)
  svd_ols_pred <- numeric(n)

  for (i in 1:n) {
    svd_ols_pred[i] <- sum(as.numeric(data[i,]) *
                           as.vector(svd_fit$coefficients))
  }

  if (binary) {
      return(ifelse(svd_ols_pred > 0.5, 1, 0))
  }

  return(svd_ols_pred)
}

ols_scores.svd <- predict_svd_ols(hold_out_data[,-1],
                                  svd_fit)

ols_scores_vif.svd <- predict_svd_ols(X_hold_out[,vif_selected_vars],
                                      svd_fit_vif)
```

```
roc_ols.svd <- Roc_Auc(score=ols_scores.svd,
                       true_class=y_real,
                       return_=FALSE,
                       model_name="SVD (Full)")

roc_ols_vif.svd <- Roc_Auc(score=ols_scores_vif.svd,
                           true_class=y_real,
                           return_=FALSE,
                           model_name="SVD (VIF)")

mRoc_Auc(roc_ols, roc_ols_vif,
         roc_ols.svd, roc_ols_vif.svd)
```

The Figure shown below, indicates that the above SVD-based implementation of least squares estimates leads to the same results as the `lm` function.

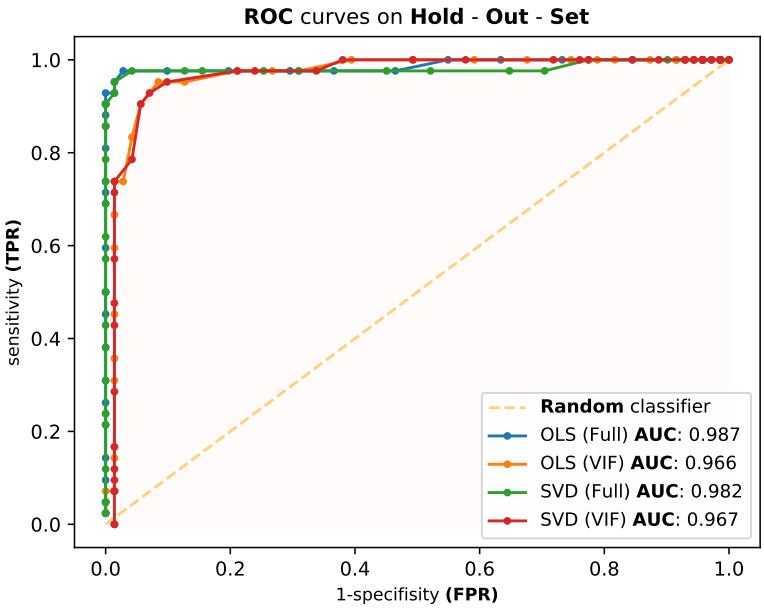


Figure 2.3: `OLS` with normal equations vs `OLS` with `SVD` ROC curves

2.1.3 Supervised Principal Component Analysis

Another approach, which we won't look into deeply, is the Supervised PCA method. This method is useful when dealing with many predictors. Its advantage lies in creating predictors that are orthogonal, implying they're not highly correlated with each other. This is particularly beneficial when our predictors have high correlation. Such high correlation can complicate immediate parameter estimation because it gets tough when a matrix doesn't have an inverse.

Principal Component Analysis (PCA) serves as a dimensionality reduction method that is unsupervised, striving to convert datasets into a lower-dimensional space while retaining the most significant variance from the original data. Through the identification of orthogonal axes, or principal components, PCA uncovers the most important patterns embedded in the data. This technique proves beneficial for decreasing computational complexity, eliminating multicollinearity and visualizing data with high dimensions. For binary classification purposes, PCA can aid in data visualization by projecting it onto a lower-dimensional space; however, it does not directly take class labels into account.

In contrast, Supervised PCA integrates class label information to discover a lower-dimensional representation that enhances the distinction between classes. This approach is more fitting for classification tasks, as it emphasizes the extraction of information that discriminates between classes.

Two influential papers in this field that have served as a source of inspiration are Bair, Tibshirani and Hastie's works [5], [4]. These papers contribute valuable insights into the applications of PCA and its supervised counterpart, showcasing their significance in the topic.

The optimization problem for **PCA** can be defined as follows:

Let $\mathbf{X} \in \mathbb{R}^{n \times p}$ be the data matrix, where n is the number of observations and p is the number of features. Center the data by subtracting the mean of each feature.

Find a projection matrix $\mathbf{W} \in \mathbb{R}^{p \times k}$, with $k \leq p$, such that the total variance of the projected data is maximized:

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmax}} \ \mathtt{Tr}(\mathbf{W}^T \mathbf{C} \mathrm{W})$$

$$\text{subject to } \mathbf{W}^T \mathbf{W} = \mathbf{I}_k.$$

where $\mathbf{C}$ is the covariance matrix of the centered data, $\mathtt{Tr}$ denotes the trace of a matrix and $\mathbf{I}_k$ is the $k \times k$ identity matrix.

The solution to this optimization problem is obtained through eigenvalue decomposition of the covariance matrix $\mathbf{C}$. The columns of the optimal projection

matrix $\mathbf{W}^*$ are the eigenvectors corresponding to the k largest eigenvalues of $\mathbf{C}$. To obtain the lower-dimensional representation of the data, project the centered data onto the principal components (eigenvectors) by calculating: $\mathbf{Z} = \mathbf{X}\mathbf{W}^*$

The optimization problem in PCA focuses on maximizing the total variance of the projected data, while the constraint ensures that the principal components are orthogonal to each other, providing an uncorrelated lower-dimensional representation of the data.

On the other hand for **Supervised PCA**, the optimization problem involves finding the principal components that maximize the separation between classes in a binary classification problem while projecting the data onto a lower-dimensional space. The objective is to find a linear transformation that captures the most discriminative information between the classes.

Let $\mathbf{y} \in \mathbb{R}^n$ be the class label vector. We compute the between-class scatter matrix $\mathbf{S}_B$ and within-class scatter matrix $\mathbf{S}_W$ using the class label information. We then find a projection vector $\mathbf{w} \in \mathbb{R}^p$ that maximizes the following objective function:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmax}} \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

The solution to this optimization problem is the eigenvector corresponding to the largest eigenvalue of the generalized eigenvalue problem: $\mathbf{S}_B\mathbf{w} = \lambda \mathbf{S}_W \mathbf{w}$.

To obtain the lower-dimensional representation of the data, project the original data onto the eigenvector $\mathbf{w}^*$: $\mathbf{Z} = \mathbf{X}\mathbf{w}^*$

The optimization problem in Supervised PCA focuses on maximizing the ratio of the between-class scatter to the within-class scatter, which helps to find a projection that best separates the classes. This is different from the unsupervised PCA, where the objective is to maximize the total variance of the projected data without considering class labels.

The `'nestedcv'` package offers the `supervisedPCA()` function, which allows one to create a supervised PCA plot using `ggplot2`.

```
library(nestedcv)

supervisedPCA(y=data$y, x=X)
```

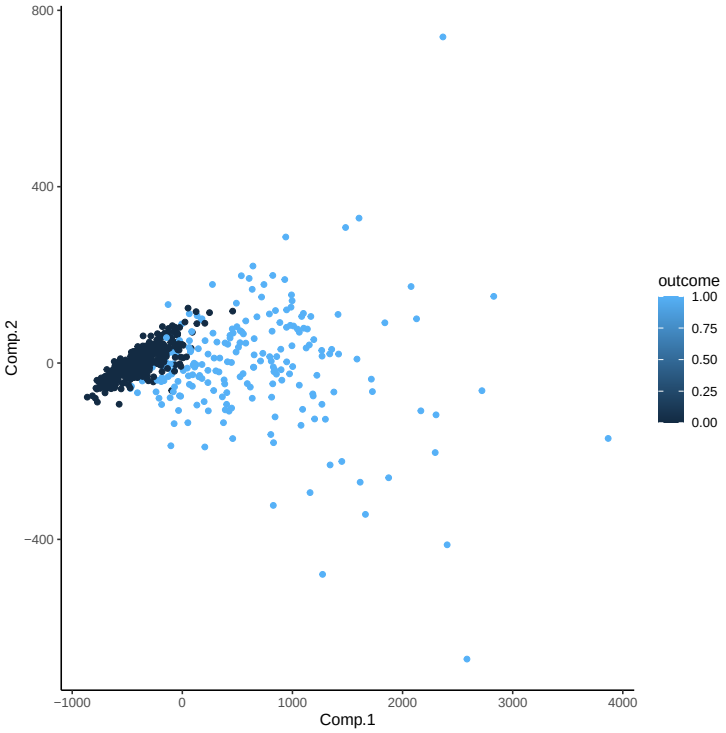


Figure 2.4: Supervised PCA with 2 components plot

2.1.4 MLE for Binary Regression

There are various methods for creating Binary Regression Models (BRM) which are mainly based on the concepts of probability and likelihood. The most commonly used method is the latent variable approach which involves connecting a latent continuous variable to a linear predictor ($\mathbf{x}\beta$) and a binary response variable (y) through a probability distribution.

Suppose there is a binary response variable (y) which takes on values of 0 or 1, indicating a patient's disease status, with 1 indicating the presence of a disease and 0 indicating the absence of the disease. For any binary response variable, it is possible to create a latent continuous variable (y^*) that represents the extent to which the status of 0 or 1 is separated from the cut-off point (τ) where the status changes from 0 to 1 or vice versa. it is usually assumed that:

$$y^* = \mathbf{x}\beta + \epsilon,$$

and if $y^* > \tau$, then $y = 1$, else $y = 0$. Link probability:

$$P(y = 1) = P(y^* > 0) = P(\mathbf{x}\beta + \epsilon > 0) = P(\epsilon < \mathbf{x}\beta) =$$

$$= F(\mathbf{x}\beta),$$

where $F(\mathbf{x}\beta)$ is some generic cumulative density function. Depending on the distributional assumption one makes about ϵ, for example, standard normal or standard logistic, one can have either probit or logit models. For binary logit models,

$$F(\mathbf{x}\beta) = \frac{\exp(\mathbf{x}\beta)}{1 + \exp(\mathbf{x}\beta)}.$$

For probit models:

$$F(\mathbf{x}\beta) = \int_{-\infty}^{x\beta} \frac{1}{2\sqrt{2p}} \exp\left(-\frac{t^2}{2}\right) \mathrm{d}t.$$

To connect $\mathbf{x}\beta$ with the observed binary response variable y, we can use an unobserved continuous variable y^* and the probability distribution of errors. However, the challenge is to find the parameter vector β using an estimation method based on this relationship. Binary regression models (BRM) are a type of Bayesian regression model that can be used to estimate β. Maximum Likelihood Estimation (MLE) is a common method for estimating β in BRM by setting up a likelihood function based on the observed data and parameters. When $y = 1, P(y = 1)$ is equal to $F(\mathbf{x}\beta)$ and when $y = 0, P(y = 0)$ is equal to $1 - F(\mathbf{x}\beta)$. Assuming that the i.i.d. assumption holds, we can calculate the likelihood of observing all the y values as the product of the probabilities of observing each y, given x and β.

$$\mathcal{L}(\beta|\mathbf{x}, y) = \prod_{i=1}^{n} [F(\mathbf{x}\beta)]^y [1 - F(\mathbf{x}\beta)]^{1-y} \Rightarrow$$

$$\mathcal{LL}(\beta|\mathbf{x}, y) = \sum_{y=1} \log F(\mathbf{x}\beta) + \sum_{y=0} \log[1 - F(\mathbf{x}\beta)].$$

The following R code illustrates how to get estimation results using the `logit` link:

```
#Train the MLE models

MLE_reg.model <- glm(y~., data=train_data,
                     family=binomial(link="logit"))

MLE_reg_vif.model <- glm(y~., data=VIF_data,
                         family=binomial(link="logit"))
```

The option, `family=binomial(link="logit")`, requests the `glm` function from the `R` base package stats to set the binary response variable to follow a binomial distribution with the `logit` link (`R` Core Team, 2020). The output shown below presents the outcomes of MLE estimation on the full set of features. The standard errors are very large compared to the coefficient estimates, making

it impossible to identify significant features. In contrast to the previous finding, when logistic regression is estimated using the VIF-filtered set of features, several significant features can be identified.

```
#Results of MLE using all predictors
summary(MLE_reg.model)
```

Call:
`glm(formula=y ~ . , family=binomial(link="logit"), data=train_data)`

Deviance Residuals:

Min	1Q	Median	3Q	Max
-2.250e-04	-2.100e-08	-2.100e-08	2.100e-08	2.633e-04

Coefficients:

Coefficients:	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-2.858e+03	5.563e+05	-0.005	0.996
radius_mean	-1.737e+02	1.455e+05	-0.001	0.999
texture_mean	-9.746e+00	3.982e+03	-0.002	0.998
perimeter_mean	5.925e+01	1.970e+04	0.003	0.998
area_mean	-1.195e+00	7.350e+02	-0.002	0.999
smoothness_mean	2.906e+03	9.501e+05	0.003	0.998
compactness_mean	-4.270e+03	8.010e+05	-0.005	0.996
concavity_mean	-1.573e+03	5.906e+05	-0.003	0.998
concave.points_mean	2.927e+03	6.296e+05	0.005	0.996
symmetry_mean	-4.169e+03	4.008e+05	-0.010	0.992
fractal_dimension_mean	1.055e+04	3.544e+06	0.003	0.998
radius_se	2.485e+03	5.024e+05	0.005	0.996
texture_se	-9.443e+01	1.399e+04	-0.007	0.995
perimeter_se	-7.619e+01	5.611e+04	-0.001	0.999
area_se	-7.600e+00	2.879e+03	-0.003	0.998
smoothness_se	-6.409e+03	2.482e+06	-0.003	0.998
compactness_se	8.832e+03	1.966e+06	0.004	0.996
concavity_se	-8.662e+03	8.981e+05	-0.010	0.992
concave.points_se	2.835e+04	3.603e+06	0.008	0.994
symmetry_se	-7.538e+03	2.117e+06	-0.004	0.997
fractal_dimension_se	-3.424e+04	7.423e+06	-0.005	0.996
radius_worst	-1.937e+02	8.014e+04	-0.002	0.998
texture_worst	2.744e+01	3.436e+03	0.008	0.994
perimeter_worst	8.129e+00	7.287e+03	0.001	0.999
area_worst	1.053e+00	6.482e+02	0.002	0.999
smoothness_worst	1.461e+03	4.578e+05	0.003	0.997
compactness_worst	-2.061e+03	2.844e+05	-0.007	0.994
concavity_worst	2.345e+03	2.917e+05	0.008	0.994

concave.points_worst	8.035e+01	3.645e+05	0.000	1.000
symmetry_worst	2.923e+03	3.855e+05	0.008	0.994
fractal_dimension_worst	2.473e+03	1.127e+06	0.002	0.998

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 6.0231e+02 on 455 degrees of freedom
Residual deviance: 6.4689e-07 on 425 degrees of freedom
AIC: 62

Number of Fisher Scoring iterations: 25

```
#Results of MLE using VIF predictors
summary(MLE_reg_vif.model)
```

Call:
glm(formula=y ∼ . , family=binomial(link="logit"), data=VIF_data)

Deviance Residuals:

Min	1Q	Median	3Q	Max
-2.9732	-0.2823	-0.0426	0.2252	3.2417

Coefficients:

Coefficients:	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-9.23687	2.43994	-3.786	0.000153 ***
texture_mean	0.38932	0.06222	6.257	3.93e-10 ***
smoothness_mean	113.43882	34.97644	3.243	0.001182 **
symmetry_mean	2.72456	13.41155	0.203	0.839018
fractal_dimension_mean	-408.48109	62.23705	-6.563	5.26e-11 ***
texture_se	-0.70861	0.56603	-1.252	0.210610
smoothness_se	-463.42826	113.20816	-4.094	4.25e-05 ***
concavity_se	8.60793	15.88530	0.542	0.587901
concave.points_se	367.64232	74.05979	4.964	6.90e-07 ***
symmetry_se	-90.74090	45.73859	-1.984	0.047267 *
fractal_dimension_se	305.69644	130.67805	2.339	0.019319 *
smoothness_worst	58.61757	22.35495	2.622	0.008738 **
symmetry_worst	23.03410	7.13652	3.228	0.001248 **

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

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 602.31 on 455 degrees of freedom
Residual deviance: 197.75 on 443 degrees of freedom

AIC: 223.75

Number of Fisher Scoring iterations: 7

The full model MLE regression did not converge, likely due to multicollinearity, as evidenced by the high `p-values`. However, MLE regression using the VIF predictors resulted in convergence and significantly lower `p-values`. Some basic residual diagnostics are shown in the figure that follows.

```
par(mfrow=c(2,4))
plot(MLE_reg.model, main="(Full MLE model)")
plot(MLE_reg_vif.model, main="(VIF MLE model)")
```

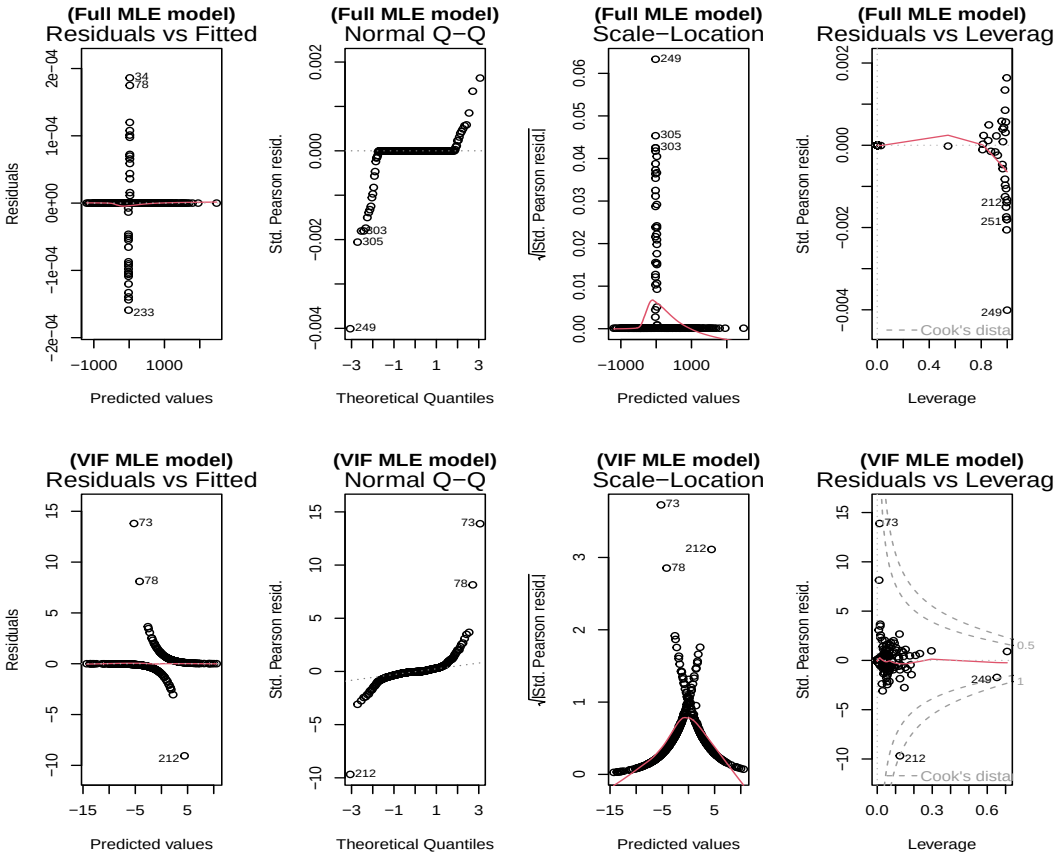


Figure 2.5: Residuals diagnostics for `MLE` regression

```r
#ROC and AUC score on out-of-sample
mle_roc <- Roc_Auc(
         score=as.vector(predict(MLE_reg.model,
         newdata=as.data.frame(X_hold_out))),
         true_class=y_real,
         model_name="MLE Regression (Full)",
         return_=FALSE)
mle_roc_vif <- Roc_Auc(
         score=as.vector(predict(MLE_reg_vif.model,
         newdata=as.data.frame(X_hold_out))),
         true_class=y_real,
         model_name="MLE Regression (VIF)",
         return_=FALSE)
mRoc_Auc(mle_roc, mle_roc_vif, fontsize=6.8, point_size=7)
```

The ROC curves shown below suggest that the predictive performance of logistic regression models is very close to the performance reported previously for conventional linear regressions.

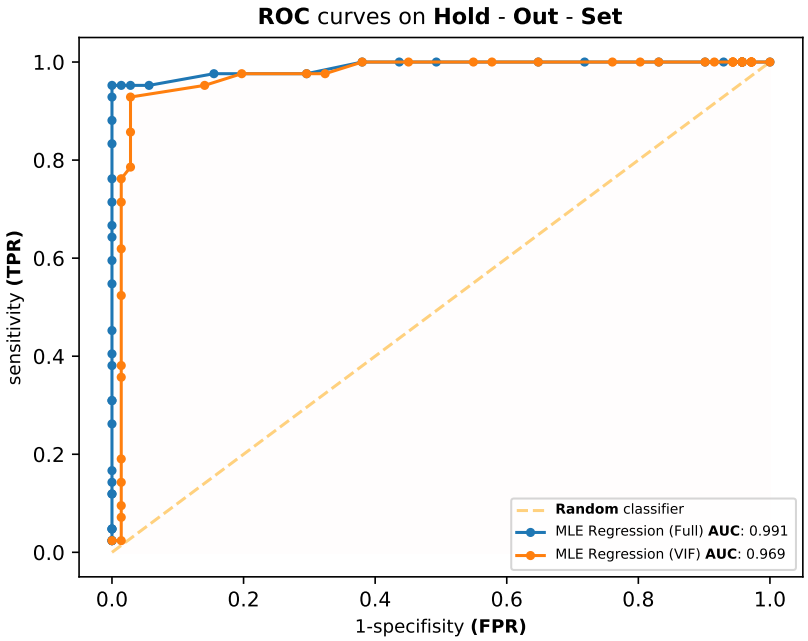


Figure 2.6: MLE regression ROC curve

2.1.5 Logistic Regression

The optimization problem for `GLMNet` logistic regression without the penalty term (i.e., the regularization term) can be defined as follows: Minimize the negative log-likelihood:

$$\underset{\beta_0,\boldsymbol{\beta}}{\text{minimize}} \quad \mathcal{L}(\beta_0,\boldsymbol{\beta}) = -\sum_{i=1}^{n}\Big[y_i \log\big(p(y_i=1|\boldsymbol{x}_i,\beta_0,\boldsymbol{\beta})\big) + (1-y_i)\log\big(1-p(y_i=1|\boldsymbol{x}_i,\beta_0,\boldsymbol{\beta})\big)\Big]$$

$$\text{subject to} \quad p(y_i=1|\boldsymbol{x}_i,\beta_0,\boldsymbol{\beta}) = \frac{1}{1+\exp\big(-(\beta_0+\boldsymbol{x}_i^T\boldsymbol{\beta})\big)}$$

Here, $p(y_i=1|\boldsymbol{x}_i,\beta_0,\boldsymbol{\beta})$ is the probability of $y_i=1$ given $\boldsymbol{x_i}$, $\boldsymbol{\beta_0}$ and $\boldsymbol{\beta}$, which can be computed using the logistic function (or sigmoid). In this optimization problem, the objective is to minimize the negative log-likelihood (L) without any penalty term.

```
unp_log_reg <- glmnet(x=as.matrix(train_data[, -1]),
                      y=as.numeric(train_data$y),
                      family=binomial(),
                      type.measure = "auc",
                      lambda=0) #lambda<-0 : no penalty


unp_log_reg_vif <- glmnet(x=as.matrix(VIF_data[, -1]),
                          y=as.numeric(train_data$y),
                          family=binomial(),
                          type.measure = "auc",
                          lambda=0) #lambda<-0 : no penalty

unp_log_reg$beta
```

Note that the estimates displayed below are derived using the pathwise coordinate descent algorithm rather than the Newton-Raphson method.

```
30 x 1 Sparse Matrix of class "dgCMatrix"
```

Feature	s0
radius_mean	1.506087e+01
texture_mean	-1.567850e+00
perimeter_mean	4.338419e-01
area_mean	7.687174e-03
smoothness_mean	-7.778613e+01
compactness_mean	-1.122636e+03
concavity_mean	-4.273363e+01
concave.points_mean	1.242726e+03
symmetry_mean	-9.125285e+02
fractal_dimension_mean	3.421298e+03
radius_se	3.189368e+02
texture_se	-2.573264e+01
perimeter_se	-8.830741e+00
area_se	1.484723e+00
smoothness_se	-2.808298e+03
compactness_se	3.097810e+02
concavity_se	-1.882358e+03
concave.points_se	8.051497e+03
symmetry_se	-2.035701e+03
fractal_dimension_se	-2.800274e+03
radius_worst	-2.816236e+01
texture_worst	7.151491e+00
perimeter_worst	1.573093e+00
area_worst	1.116984e-01
smoothness_worst	7.313677e+02
compactness_worst	-2.109567e+02
concavity_worst	4.473242e+02
concave.points_worst	5.030202e+01
symmetry_worst	7.733514e+02
fractal_dimension_worst	-1.792663e+02

```
unp_log_reg_vif$beta
```

```
12 x 1 sparse Matrix of class "dgCMatrix"
```

Feature	s0
texture_mean	0.3890011
smoothness_mean	113.1405537
symmetry_mean	2.7183387
fractal_dimension_mean	-408.1093275
texture_se	-0.7079108
smoothness_se	-463.5441376
concavity_se	8.6172685
concave.points_se	367.6025386
symmetry_se	-90.4614211
fractal_dimension_se	305.1297670
smoothness_worst	58.7328776
symmetry_worst	22.9982673

2.1.6 Least Absolute Deviations

Ordinary Least Squares (OLS) and Least Absolute Deviation (LAD) are two common methods used in regression analysis for estimating the parameters of a linear model. Although they are conceptually related, they have distinct historical backgrounds.

The OLS method has a longer history, with roots dating back to the 18th century. It is primarily attributed to two mathematicians: Carl Friedrich Gauss[3] and Adrien-Marie Legendre[4]. Gauss is regarded as one of the greatest mathematicians of all time. In 1809, Gauss published his book [18], in which he introduced the method of least squares to estimate the orbits of celestial bodies. This work is considered to be the first formal introduction of the OLS method. Legendre independently discovered the OLS method around the same time as Gauss. In 1806, he published his book [24], where he introduced the term "least squares" and explained the method for fitting a curve to a set of observed data points.

On the other hand, the LAD method, also known as the `L1-norm` or Least Absolute Errors (LAE), has a rich history that predates OLS. It was first introduced by Tobias Mayer in the 18th century and later formalized by Adrien-Marie Legendre in 1805. The LAD method aims to minimize the sum of the absolute differences between the observed values and the predicted values of the linear model. The table below provides a general comparison between OLS and LAD, highlighting the differences between the two methods.

Table 2.7: `OLS` vs `LAD`

	Ordinary least squares	**Least absolute deviations**
Robustness	Not very robust	Robust
Solution stability	Stable solution	Unstable solution
Number of solutions	One solution* $(n \geq p)$	Possibly multiple solutions

Least Absolute Deviations (LAD) is also referred to as least absolute errors, least absolute residuals, or least absolute values. It is an optimization technique that can be used to determine the best fitting line or curve for a set of data points. Unlike the least squares method, LAD gives equal weight to all data points and is less sensitive to the influence of outliers.

The LAD estimate can also be derived as the maximum likelihood estimate

[3]Carl Friedrich Gauss (1777-1855): A German mathematician, Gauss is regarded as one of the greatest mathematicians of all time

[4]Adrien-Marie Legendre (1752-1833): A French mathematician

if the errors in the data have a Laplace distribution[5]. The method was first introduced by Roger Joseph Boscovich in 1757.
Then LAD problem is equivalent to following optimization problem:

$$\min \sum_{i=1}^{n} |f(x_i) - y_i|.$$

The `lad` function in the `L1pack` package in `R` performs Least Absolute Deviations (LAD) regression, also known as L^1 regression or quantile regression with τ (`tau`) set to 0.5 (the median). The LAD regression model aims to minimize the sum of absolute deviations between the true values and the predicted values.

The optimization problem that the `lad` function solves is:

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \left| y_i - \left(\beta_0 + \sum_{j=1}^{p} \beta_j x_{ij} \right) \right| ,$$

where:

◇ n is the number of observations
◇ y_i is the observed response variable for the i-th observation
◇ x_{ij} is the value of the j-th predictor variable for the i-th observation
◇ β_0 is the intercept term
◇ β_j is the coefficient for the j-th predictor variable

The `lad` function in the `L1pack` library estimates the coefficients that minimize the objective function. Note that the `lad` function does not include any regularization.

```
#LAD without regularization
library(L1pack)
# This function is used to fit linear models
# considering Laplace errors.

lad_full <- lad(y~., data=train_data)
lad_vif <- lad(y~., data=VIF_data)

summary(lad_full)
```

Call:
`lad(formula=y ~ . , data=train_data)`

Residuals:

Min	1Q	Median	3Q	Max
-0.8223	-0.1018	0.0000	0.1348	0.9106

[5]Laplace distribution pdf: $f(x|\,\mu,\beta) = \frac{1}{2\beta}\exp\left(-|x-\mu|/\beta\right)$

Coefficients:

Coefficients:	Estimate	Std.Error	z value	p-value
(Intercept)	-1.1860	0.7047	-1.6828	0.0924
radius_mean	-0.1380	0.2921	-0.4725	0.6366
texture_mean	0.0050	0.0133	0.3733	0.7089
perimeter_mean	-0.0122	0.0409	-0.2980	0.7657
area_mean	0.0013	0.0009	1.4317	0.1522
smoothness_mean	3.4586	3.2757	1.0558	0.2910
compactness_mean	-3.4595	2.2005	-1.5722	0.1159
concavity_mean	3.7083	1.7381	2.1336	0.0329
concave.points_mean	3.4767	3.2382	1.0737	0.2830
symmetry_mean	-0.1140	1.2777	-0.0892	0.9289
fractal_dimension_mean	-2.4102	9.2312	-0.2611	0.7940
radius_se	0.0587	0.5322	0.1103	0.9121
texture_se	0.0279	0.0686	0.4070	0.6840
perimeter_se	-0.0224	0.0678	-0.3310	0.7406
area_se	-0.0008	0.0025	-0.3328	0.7393
smoothness_se	21.2361	11.1982	1.8964	0.0579
compactness_se	0.4642	4.6162	0.1006	0.9199
concavity_se	-4.2071	2.6227	-1.6041	0.1087
concave.points_se	6.6408	9.8889	0.6715	0.5019
symmetry_se	3.5737	4.8015	0.7443	0.4567
fractal_dimension_se	-12.6368	22.3215	-0.5661	0.5713
radius_worst	0.2578	0.1038	2.4830	0.0130
texture_worst	0.0000	0.0120	0.0021	0.9983
perimeter_worst	-0.0012	0.0099	-0.1209	0.9037
area_worst	-0.0012	0.0006	-2.0491	0.0405
smoothness_worst	-2.7339	2.3714	-1.1529	0.2490
compactness_worst	0.4868	0.7295	0.6672	0.5046
concavity_worst	-0.1625	0.4946	-0.3285	0.7426
concave.points_worst	-0.1147	1.5789	-0.0727	0.9421
symmetry_worst	0.4580	0.8453	0.5417	0.5880
fractal_dimension_worst	4.9933	4.1140	1.2137	0.2248

Degrees of freedom: 456 total; 425 residual
Scale estimate: 0.2457347
Log-likelihood: 25.95977 on 32 degrees of freedom

```
summary(lad_vif)
```

Call:
lad(formula=y ∼ . , data=VIF_data)

Residuals:

Min	1Q	Median	3Q	Max
-0.9048	-0.2165	0.0000	0.2395	1.0164

Coefficients:

Coefficients:	Estimate	Std.Error	z value	p-value
(Intercept)	-0.4654	0.3129	-1.4876	0.1369
texture_mean	0.0375	0.0068	5.5069	0.0000
smoothness_mean	-0.4225	3.9778	-0.1062	0.9154
symmetry_mean	1.2019	1.6104	0.7464	0.4555
fractal_dimension_mean	-31.0609	6.5043	-4.7754	0.0000
texture_se	-0.1024	0.0620	-1.6499	0.0990
smoothness_se	-25.1445	13.4543	-1.8689	0.0616
concavity_se	-0.3213	1.6213	-0.1982	0.8429
concave.points_se	32.7416	7.6210	4.2962	0.0000
symmetry_se	-3.2957	5.3849	-0.6120	0.5405
fractal_dimension_se	45.1119	17.9538	2.5127	0.0120
smoothness_worst	9.5804	2.8160	3.4021	0.0007
symmetry_worst	1.5076	0.9075	1.6613	0.0967

Degrees of freedom: 456 total; 443 residual
Scale estimate: 0.3590728
Log-likelihood: -146.9886 on 14 degrees of freedom.

The following code provides the residual diagnostics for the LAD regression. The results are shown in Figure 2.7.

```
par(mfrow=c(2,2))

plot(lad_full, main="(Full model)")
plot(lad_vif, main="(VIF model)")
```

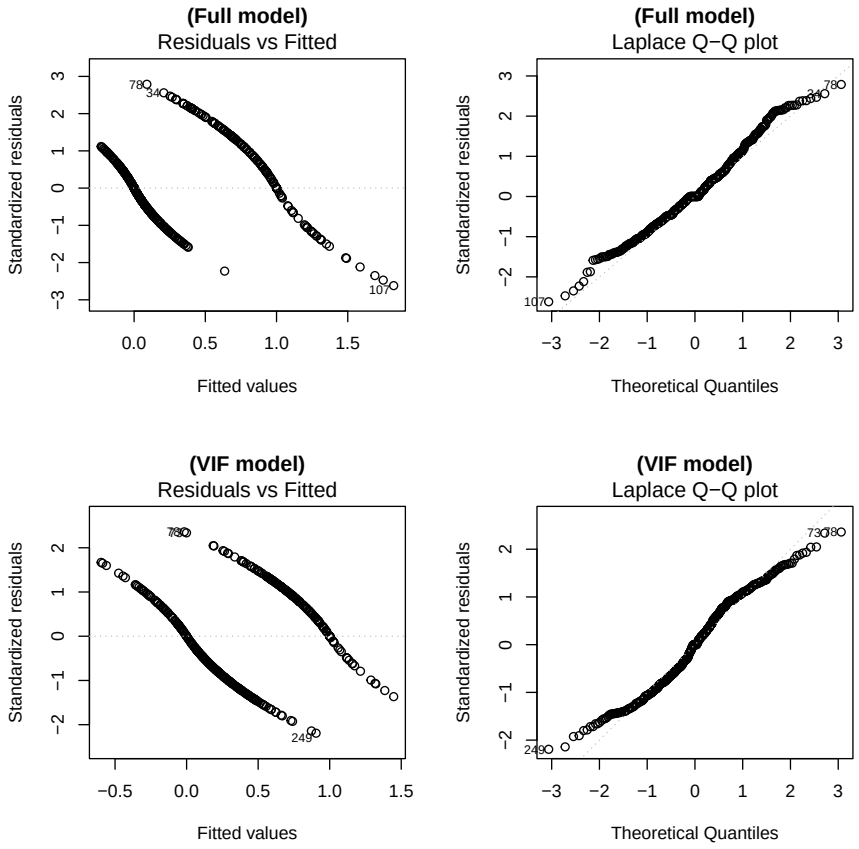


Figure 2.7: `LAD` residuals diagnostics

The `confint()` function in `R` is used to calculate confidence intervals for model parameters. The method used to calculate the confidence intervals depends on the type of model being fitted.

For linear models, `confint()` uses the `t`-distribution to calculate confidence intervals.

```
#95% CI
confint(lad_full)
confint(lad_vif)
```

confint	2.5 %	97.5 %
(Intercept)	$-2.56723649721769e+00$	$1.95316419404602e-01$
radius_mean	$-7.10435825558113e-01$	$4.34427666005796e-01$
texture_mean	$-2.11052890782187e-02$	$3.10363926617581e-02$
perimeter_mean	$-9.23824540396470e-02$	$6.79960355568921e-02$
area_mean	$-4.73877815988548e-04$	$3.04276703073125e-03$
smoothness_mean	$-2.96160883018555e+00$	$9.87887713594703e+00$
compactness_mean	$-7.77227203224185e+00$	$8.53349605649395e-01$
concavity_mean	$3.01730337725473e-01$	$7.11490986757127e+00$
concave.points_mean	$-2.87003681910367e+00$	$9.82337462131443e+00$
symmetry_mean	$-2.61820975305550e+00$	$2.39027277723156e+00$
fractal_dimension_mean	$-2.05031116459563e+01$	$1.56826875555428e+01$
radius_se	$-9.84357541542095e-01$	$1.10179850458944e+00$
texture_se	$-1.06528985357864e-01$	$1.62365153942507e-01$
perimeter_se	$-1.55326237483675e-01$	$1.10442021371689e-01$
area_se	$-5.63811831256398e-03$	$4.00128377555899e-03$
smoothness_se	$-7.12022444736725e-01$	$4.31842888102463e+01$
compactness_se	$-8.58339348899404e+00$	$9.51184546487416e+00$
concavity_se	$-9.34745990427935e+00$	$9.33228288972450e-01$
concave.points_se	$-1.27410723790093e+01$	$2.60226789248297e+01$
symmetry_se	$-5.83698673413914e+00$	$1.29844660584586e+01$
fractal_dimension_se	$-5.63860569041301e+01$	$3.11124822687001e+01$
radius_worst	$5.43043433557552e-02$	$4.61294108777785e-01$
texture_worst	$-2.35585261867779e-02$	$2.36096462378927e-02$
perimeter_worst	$-2.06474963528022e-02$	$1.82474509268581e-02$
area_worst	$-2.33748326063482e-03$	$-5.19683079194162e-05$
smoothness_worst	$-7.38186142359870e+00$	$1.91400243682795e+00$
compactness_worst	$-9.43108451535057e-01$	$1.91665560705454e+00$
concavity_worst	$-1.13189289196784e+00$	$8.06967206090997e-01$
concave.points_worst	$-3.20921501992566e+00$	$2.97979276707505e+00$
symmetry_worst	$-1.19885271377463e+00$	$2.11475867534855e+00$
fractal_dimension_worst	$-3.06997129864694e+00$	$1.30566209010729e+01$

confint	2.5 %	97.5 %
(Intercept)	−1.0785921322683136	0.1477931592656955
texture_mean	0.0241689018557686	0.0508792226110136
smoothness_mean	−8.2188395063679742	7.3738958235261061
symmetry_mean	−1.9543407705816809	4.3581588501712183
fractal_dimension_mean	−43.8091275921968162	−18.3125761725256950
texture_se	−0.2239790652714481	0.0192398572329353
smoothness_se	−51.5143823562969274	1.2253396015534932
concavity_se	−3.4990353974181931	2.8564745942792711
concave.points_se	17.8047014475855789	47.6785048119134061
symmetry_se	−13.8498414587770196	7.2584370171226400
fractal_dimension_se	9.9230236254330535	80.3007975018001616
smoothness_worst	4.0611551369524665	15.0996448549870319
symmetry_worst	−0.2710557406034533	3.2863226953129741

2.2 Logistic vs Conventional Regression

The following table, compares unpenalized logistic regression with ordinary least squares (OLS) regression using a classification threshold of 0.5. Logistic regression clearly outperforms the conventional OLS regression.

Table 2.10: Comparing Logistic Regression and OLS

	Logistic Regression		**OLS Regression**	
	Full	**VIF**	**Full**	**VIF**
Accuracy	0.9823	0.9469	0.9646	0.9292
95% CI	(0.9375, 0.9978)	(0.888, 0.9803)	(0.9118, 0.9903)	(0.8653, 0.9689)
No Information Rate	0.6283	0.6283	0.6283	0.6283
P-Value [Acc > NIR]	<2e-16	1.866e-15	<2e-16	1.372e-13
Kappa	0.9617	0.8852	0.9227	0.8484
Mcnemar's Test P-Value	0.4795	0.6831	0.1336	1.0000
Sensitivity	0.9524	0.9048	0.9048	0.9048
Specificity	1.0000	0.9718	1.0000	0.9437
Pos Pred Value	1.0000	0.9500	1.0000	0.9048
Neg Pred Value	0.9726	0.9452	0.9467	0.9437
Precision	1.0000	0.9500	1.0000	0.9048
Recall	0.9524	0.9048	0.9048	0.9048
F1 Score	0.9756	0.9268	0.9500	0.9048
Prevalence	0.3717	0.3717	0.3717	0.3717
Detection Rate	0.3540	0.3363	0.3363	0.3363
Detection Prevalence	0.3540	0.3540	0.3363	0.3717
Balanced Accuracy	0.9762	0.9383	0.9524	0.9242

One can easily calculate the aforementioned metrics in `R` by using the `confusionMatrix` method from the `caret` package.

```
confusionMatrix(predict,
                actual,
                mode = "everything",
                positive='1')
```

2.3 Full Model vs Second Order Terms vs VIF-Filtered

In this section, we will construct a variety of second-order term models and assess their effectiveness using specific information criteria such as the Corrected Akaike Information Criterion (`AICc`) and the Bayesian Information Criterion

(BIC). We will delve into these criteria in detail in the upcoming chapter. For now, it is important to remember that a lower AICc/BIC score indicates a more effective model.

Let's explore how to do that in R. First, we will construct a function that incorporates all the squared terms and cross-product terms in the form of $X_i \cdot X_j$.

```
second_order_terms <- function(data, symbol="^2") { #data must be as.df
# Loop through each column and create the corresponding 2nd-order term
  for (i in 1:ncol(data)) {
    col_name <- names(data)[i]
    data[paste0(col_name, symbol)] <- data[[col_name]]^2
  }
  return(data)
}
```

The above function returns a `data.frame` that contains all the first order terms, plus, the squared ones. $30 + 30$ in total.
The following function produces all the interactions $X_i \cdot X_j$ (`data.frame` type) $30 \cdot 29$ in total.

```
interactions_xixj <- function(X_train, symbol=":") {

    X_interaction <- data.frame(matrix(ncol = 0, nrow = nrow(X_train)))

    # Iterate through each pair of columns (i not equal to j)
    for (i in 1:ncol(X_train)) {
      for (j in 1:ncol(X_train)) {
        if (i != j) {
         #Calculate the interaction term for the current pair of columns
          interaction_term <- X_train[[i]] * X_train[[j]]

          # Generate the column name for the interaction term
          interaction_name <- paste0(names(X_train)[i],
                                     symbol, names(X_train)[j])

          # Add the interaction term to the 'X_interaction' data frame
          X_interaction[[interaction_name]] <- interaction_term
        }
      }
    }
    return(X_interaction)
}
```

As a result of high multicollinearity, it is not feasible to fit a linear model using all the first and second-order terms. Therefore, we will follow a specific procedure: we will start by creating a data frame with all the first-order and second-order predictors, as well as the label y, resulting in a total of $(30 + 30 + 30 \cdot 29 + 1)$

columns. We will then set various cutoff thresholds and remove any predictors whose correlation exceeds the specified threshold, using the `collinear(data, rsq_cutoff = threshold)` method from the `nestedcv` package. Next, we will fit a linear model using the remaining features and calculate their `VIF` scores. Once this procedure is complete, we can compare the resulting models using information criteria and `adj. r2`. The `'foreach'` package is particularly helpful for performing these calculations since we will be fitting a large number of models. The `foreach` package in `R` is a powerful and flexible tool for performing parallel and distributed loop computations. It provides an alternative to the traditional 'for' loop and 'apply' family functions, making it easier to perform repetitive tasks with greater efficiency.

```
library(foreach)

vif_inter_2nd_X <- second_order_terms(as.data.frame(train_data)[-1])
vif_inter_2nd_train <- vif_inter_2nd_X
vif_inter_2nd_train$y <- train_data$y

xixj_int <- interactions_xixj(as.data.frame(train_data)[-1])

vif_2nd_xixj_full <- cbind(train_data, # 1st order terms
                           vif_inter_2nd_X[31:60], #squared terms
                           xixj_int) #29*30 interaction terms

VIF_steps <- foreach( threshold = seq(0.2, 1, 0.1) )  %do% {

    cat("\n\nfor threshold : ", threshold, "\n\n")

    ### Cut off corr > threshold predictors ###

    system.time(cut_aliased_coeff_2nd <- collinear(
                                          vif_inter_2nd_X[31:60],
                                          rsq_cutoff = threshold))

    df_cut_2nd <- as.data.frame(as.data.frame(vif_inter_2nd_X[31:60])[,
                                         -cut_aliased_coeff_2nd])

    print(paste("Remaining predictors of 2nd order X^2: ",
                dim(df_cut_2nd)[2], "out of:",
                dim(vif_inter_2nd_X)[2]))


    system.time(cut_aliased_coeff_xixj <- collinear(xixj_int,
                                           rsq_cutoff = threshold))

    df_cut_xixj <- as.data.frame(as.data.frame(xixj_int)[,
                                         -cut_aliased_coeff_xixj])

    print(paste("Remaining predictors of 2nd order Xi*Xj: ",
```

```
                dim(df_cut_xixj)[2], "out of:",
                dim(xixj_int)[2]))

    ### Cut off corr > threshold predictors ###

    vif_2nd_xixj <- cbind(VIF_data,
                          df_cut_2nd,
                          df_cut_xixj)

    vif_2nd_xixj_stdr <- as.data.frame(vif_2nd_xixj %>%
       mutate(across(names(vif_2nd_xixj), scale)))



    print(paste("Total predictors remained after the cut off: ",
                dim(vif_2nd_xixj)[2]-1))


    ###### calc. vif values ######

    system.time(vif_values_2nd_xixj <- vif(lm(y~.,
                                    data=vif_2nd_xixj_stdr))) #fit a lm

    vif_df_2nd_xixj <- data.frame(
            Feature = names(vif_values_2nd_xixj),
            VIF_value=vif_values_2nd_xixj[names(vif_values_2nd_xixj)],
            Tolerance=1/vif_values_2nd_xixj[names(vif_values_2nd_xixj)])


    vif_selected_vars_2nd <- c()
    j <- 0

    cat("\n\nVIF predictors selected:\n\n")
    for (i in seq_along(vif_values_2nd_xixj)) {
      vif_val <- vif_values_2nd_xixj[i]
      if (vif_val < 20) {
        j = j+1
        print(paste('feature selected', j, ':',
                    names(vif_values_2nd_xixj)[i], ":",
                    round(vif_val, 2)))
        vif_selected_vars_2nd[j] <- names(vif_values_2nd_xixj)[i]
      }
    }
    if (length(vif_selected_vars_2nd) != 0) {
        return(vif_selected_vars_2nd)
    }
}
```

```
### for threshold :  0.2

### [1] "Remaining predictors of 2nd order X^2:  5 out of: 60"
### [1] "Remaining predictors of 2nd order Xi*Xj:  1 out of: 870"
### [1] "Total predictors remained after the cut off:  18"

### VIF predictors selected:

### [1] "feature selected 1 : symmetry_mean : 3.61"
### [1] "feature selected 2 : fractal_dimension_mean : 5.31"
### [1] "feature selected 3 : texture_se : 17.34"
### [1] "feature selected 4 : smoothness_se : 19.05"
### [1] "feature selected 5 : concavity_se : 3.32"
### [1] "feature selected 6 : concave.points_se : 3.98"
### [1] "feature selected 7 : symmetry_se : 3.86"
### [1] "feature selected 8 : fractal_dimension_se : 3.62"
### [1] "feature selected 9 : smoothness_worst : 8.97"
### [1] "feature selected 10 : symmetry_worst : 6.81"
### [1] "feature selected 11 : `texture_se^2` : 14.63"
### [1] "feature selected 12 : `smoothness_se^2` : 11.41"

### for threshold :  0.3

### [1] "Remaining predictors of 2nd order X^2:  6 out of: 60"
### [1] "Remaining predictors of 2nd order Xi*Xj:  2 out of: 870"
### [1] "Total predictors remained after the cut off:  20"

### VIF predictors selected:

### [1] "feature selected 1 : symmetry_mean : 3.62"
### [1] "feature selected 2 : fractal_dimension_mean : 5.8"
### [1] "feature selected 3 : texture_se : 17.36"
### [1] "feature selected 4 : concavity_se : 3.36"
### [1] "feature selected 5 : concave.points_se : 4.15"
### [1] "feature selected 6 : symmetry_se : 17.92"
### [1] "feature selected 7 : fractal_dimension_se : 3.71"
### [1] "feature selected 8 : smoothness_worst : 9.6"
### [1] "feature selected 9 : symmetry_worst : 7.02"
### [1] "feature selected 10 : `texture_se^2` : 14.68"
### [1] "feature selected 11 : `smoothness_se^2` : 12.14"
### [1] "feature selected 12 : `symmetry_se^2` : 12.08"
### [1] "feature selected 13 : `radius_mean:smoothness_se` : 16.16"

### for threshold :  0.4
.
.
.
```

We keep the predictors mentioned above for each threshold, creating 7 models. Note that the seventh model chosen is the same as the original VIF one.

```
VIF_steps_filt <- VIF_steps[sapply(VIF_steps,
                            function(x) !is.null(x))] #filter out NULL

get_2nd_model <- function (i) { transform(y=train_data$y,
                            as.data.frame(vif_2nd_xixj_full)[,
                            gsub("`", "",
                                unlist(VIF_steps_filt[i]))])
                              }
```

We are now ready to evaluate these models in comparison to one another, as well as against the complete model.

```
# Fit the models
model1 <- lm(y ~ ., data = get_2nd_model(1))
model2 <- lm(y ~ ., data = get_2nd_model(2))
model3 <- lm(y ~ ., data = get_2nd_model(3))
model4 <- lm(y ~ ., data = get_2nd_model(4))
model5 <- lm(y ~ ., data = get_2nd_model(5))
model6 <- lm(y ~ ., data = get_2nd_model(6))

full.model <- lm(y ~ ., data = train_data)
vif.model <- lm(y ~ ., data = VIF_data)


aic_values <- c(AIC(model1), AIC(model2),
                AIC(model3), AIC(model4),
                AIC(model5), AIC(model6),
                AIC(full.model), AIC(vif.model))

aicc_values <- c(AICc(model1), AICc(model2),
                 AIC(model3),  AICc(model4),
                 AICc(model5), AICc(model6),
                 AICc(full.model), AICc(vif.model))

bic_values <- c(BIC(model1), BIC(model2),
                BIC(model3), BIC(model4),
                BIC(model5), BIC(model6),
                BIC(full.model), BIC(vif.model))


### Adjusted R-Squared ###

#If we want to compare nested models, R-squared can
#be problematic because it will ALWAYS favor the larger
#(and therefore more complex) model. Adjusted R-squared is
#an alternative metric that penalizes R-squared for each
#additional predictor. Therefore, larger nested models will
#always have larger R-squared but may have smaller adjusted
#R-squared.
```

```r
adj_r2_values <- c(summary(model1)$adj.r.squared,
                   summary(model2)$adj.r.squared,
                   summary(model3)$adj.r.squared,
                   summary(model4)$adj.r.squared,
                   summary(model5)$adj.r.squared,
                   summary(model6)$adj.r.squared,
                   summary(full.model)$adj.r.squared,
                   summary(vif.model)$adj.r.squared)

predictors <- c(model1$rank - 1, model2$rank - 1,
                model3$rank - 1, model4$rank - 1,
                model5$rank - 1, model6$rank - 1,
                full.model$rank - 1, vif.model$rank - 1)

# Compare info. crit. values
aic_comparison <- data.frame(Model = c(1,2,3,4,5,6,'Full',
                                       'Original VIF'),
                             AIC = aic_values,
                             AICc = aicc_values,
                             BIC = bic_values,
                             adj_r2 = adj_r2_values,
                             Predictors = predictors)
aic_comparison <- aic_comparison[order(aic_comparison$AICc), ]
## View the comparison table ##
aic_comparison
```

Model	AIC	AICc	BIC	adj_r2	Predictors
Full	23.66706	28.65997	155.5868	0.7546140	30
2	219.16694	220.25784	281.0043	0.6097058	13
Original VIF	251.56759	252.51997	309.2825	0.5800729	12
1	305.21809	306.17047	362.9330	0.5276427	12
5	357.15763	358.11001	414.8725	0.4706562	12
3	362.92569	362.92569	424.7631	0.4650564	13
4	396.48534	397.18963	445.9553	0.4205088	10
6	418.73516	419.05731	451.7151	0.3862797	6

Table 2.11: Comparing `Full`, `VIF-filtered` & `second-order-terms` models, based on information criteria and adjusted R^2

Predictors	Model
symmetry_mean, fractal_dimension_mean, texture_se, smoothness_se, concavity_se, concave.points_se, symmetry_se, fractal_dimension_se, smoothness_worst, symmetry_worst, texture_se.2, smoothness_se.2	1
symmetry_mean, fractal_dimension_mean, texture_se, concavity_se, concave.points_se, symmetry_se, fractal_dimension_se, smoothness_worst, symmetry_worst, texture_se.2, smoothness_se.2, symmetry_se.2, radius_mean:smoothness_se	2
symmetry_mean, concavity_se, concave.points_se, fractal_dimension_se, smoothness_worst, symmetry_worst, compactness_mean.2, texture_se.2, smoothness_se.2, compactness_se.2, concavity_se.2, concave.points_se.2, symmetry_se.2	3
concavity_se, concave.points_se, fractal_dimension_se, smoothness_worst compactness_mean.2 texture_se.2, smoothness_se.2, compactness_se.2, concavity_se.2, concave.points_se.2	4
concavity_se, fractal_dimension_se, smoothness_worst, compactness_mean.2, radius_se.2, texture_se.2, smoothness_se.2, compactness_se.2, concavity_se.2, concave.points_se.2, fractal_dimension_se.2, compactness_worst.2	5
radius_se.2, compactness_se.2, concavity_se.2, fractal_dimension_se.2, compactness_worst.2, concavity_worst.2	6

Table 2.12: Second order interactions, predictors survived VIF filtering

Note that texture_se.2 represents the square of texture_se (texture_se^2), while radius_mean:smoothness_se represents the product of radius_mean and smoothness_se (radius_mean×smoothness_se). According to Table 2.11, it is evident that Model 2 performs well and even outperforms the original VIF model. The following code offers a visualization of residual diagnostics (Figure 2.8) for the second-order model (model 2) using MLE regression.

```
#fit a mle model
so_mle.fit <- glm(y~. , data=sec_ord_train_data,
                  family=binomial(link='logit'))

par(mfrow=c(2,2))
plot(so_mle.fit)
```

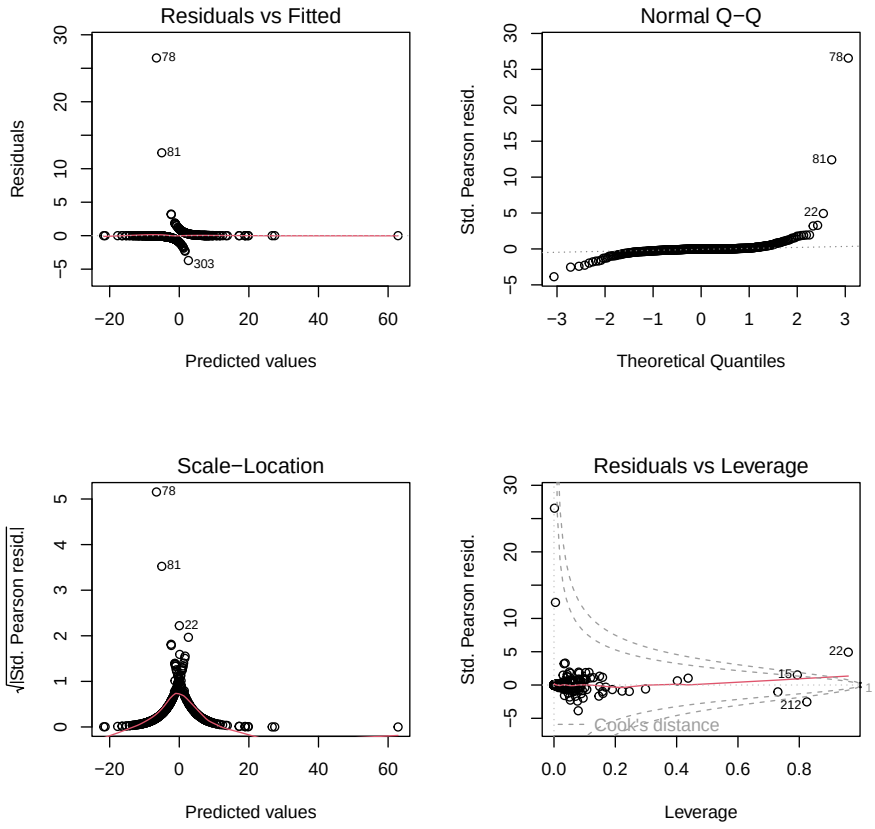


Figure 2.8: Residuals diagnostics for 2nd order (`model 2`) `MLE` regression

The code provided below combines the ROC curves of three models: the full model (which uses all predictors), the VIF-filtered model and the second-order model (`model 2`, with 13 features). These models are based on MLE regression. As depicted in Figure 2.9, the second-order model's performance closely mirrors that of the full model, boasting an AUC of 0.99, almost identical to the full model's AUC.

```
hold_out_so <- cbind(
         interactions_xixj(as.data.frame(X_hold_out),
                           symbol="."),
         second_order_terms(as.data.frame(X_hold_out),
                            symbol=".2"))[,
                            names(sec_ord_train_data)[-1]]

y_pred_MLE_2ord <- predict(so_mle.fit,
```

```
                        newdata=as.data.frame(hold_out_so),
                        type="response")


mle_roc_2o <- Roc_Auc(
         score=y_pred_MLE_2ord,
         true_class=y_real,
         model_name="MLE Regression (2nd order)",
         return_=FALSE)

mRoc_Auc(mle_roc,
         mle_roc_vif,
         mle_roc_2o,
         fontsize=8.2,
         point_size=7)
```

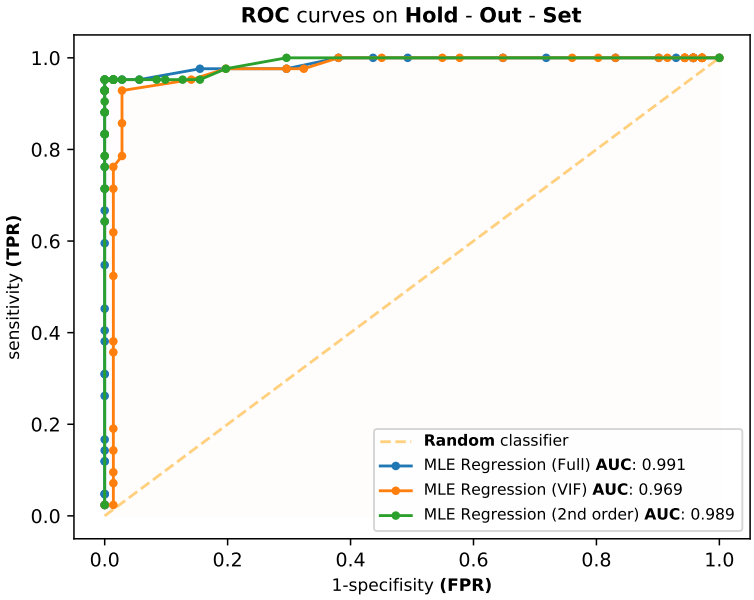


Figure 2.9: `Second-order, VIF-filtered` and `Full-model MLE` ROC curves

2.3.1 Second-Order Stepwise Model Selection

In this approach, we use a different process than the previous one. Instead of removing related predictors, we start with the complete model using all 30 predictors. Then, at each step, we add the best interaction term to improve

the model's fit. We can use `AICc`, `BIC`, or `adjusted R-squared` to choose the best term. We test all possible second-order interactions $X_i \cdot X_j$, where i and j can be the same. This process continues until it reaches a stable number, which usually takes about $35-40$ steps. To improve speed, we use the `'foreach'` and `'doParallel'`[6] packages. `'doParallel'` acts as a parallel support system for `'foreach'`, allowing it to run loops simultaneously. To execute code in parallel, `'foreach'` must be combined with a package like `'doParallel'`.

```
library(doParallel)
library(foreach)
registerDoParallel(cores = 2)


stepwise_interactions_selection <- function (interactions_2nd,
                                             train_data,
                                             criterion="AICc",
                                             iters=35,
                                             show_steps=TRUE) {

    best <- ''
    prev_AICc <- AICc(full.model)
    prev_BIC <- BIC(full.model)
    prev_adk.r2 <- summary(full.model)$adj.r.squared

    system.time({
        best_interactions_stepwise <- foreach (i=1:iters)  %do% {

            #Check if X^2 interaction fits the best
            if (substr(best, nchar(best)-1, nchar(best)) == '^2') {
                x_k <- best_interaction_term(interactions_2nd,
                                             train_data,
                                             next_interaction=best)

                best <- paste(best, '+', "I(", x_k[1], ")")

                best_term_step <- x_k[1]
                new_aicc <- round(as.numeric(x_k[2]),3)
                new_bic <- round(as.numeric(x_k[3]),3)
                new_adj.r2 <- round(as.numeric(x_k[4]),3)

            }else { # Xi*Xj, i!=j interaction
                x_k <- best_interaction_term(interactions_2nd,
                                             train_data,
                                             next_interaction=best)

                best <- paste(best, '+', x_k[1])
```

[6]Steve Weston wrote the original version of this vignette for the `doMC` package. Rich Calaway adapted the vignette for `doParallel`

```
                best_term_step <- x_k[1];
                new_aicc <- round(as.numeric(x_k[2]),3);
                new_bic <- round(as.numeric(x_k[3]),3);
                new_adj.r2 <- round(as.numeric(x_k[4]),3);
                    }

            prev_AICc = new_aicc

            if (show_steps) {

            cat("loop: ", i, ": Interaction added:",
                best_term_step, "\nAICc: ", new_aicc,
                "\nBIC: ", new_bic,
                "\nadj. R2: ", new_adj.r2, '\n\n\n')
                }
            return(x_k[1])
            }
        })
    return (best_interactions_stepwise)
    }
```

What is the time complexity of the above? Let n denote the number of interactions (in our case 30×30) and k the iterations it took to converge.

- 1st. iteration: n
- 2nd. iteration: $n-1$
- 3rd. iteration: $n-2$

  $\vdots$
- Final iteration: $n-k$

Hence, the time complexity is given by:

$$T(n,k) = \sum_{i=0}^{k}(n-i) = \frac{1}{2}(k+1)(2n-k).$$

The code below employs the `AICc` criterion for model selection.

```
## Takes around 6-7 mins. for each info criterion.
stepwise_int_aicc <- stepwise_interactions_selection(
                                    interactions_2nd,
                                    train_data,
                                    criterion="AICc",
                                    iters=45)
```

`Output:`

```
## loop:  1 : Interaction added: radius_worst:area_worst
## AICc:  -42.21
## BIC:  88.515
## adj. R2:  0.791


## loop:  2 : Interaction added: smoothness_mean:area_se
## AICc:  -72.819
## BIC:  61.692
## adj. R2:  0.805


## loop:  3 : Interaction added: smoothness_worst:concave.points_worst
## AICc:  -83.593
## BIC:  54.694
## adj. R2:  0.81

.     .     .

## loop:  32 : Interaction added: concavity_mean:fractal_dimension_mean
## AICc:  -177.071
## BIC:  64.721
## adj. R2:  0.858


## loop:  33 : Interaction added: smoothness_mean:concavity_mean
## AICc:  -177.193
## BIC:  65.489
## adj. R2:  0.859


## loop:  34 : Interaction added: texture_mean:fractal_dimension_se
## AICc:  -177.193
## BIC:  68.769
## adj. R2:  0.859
```

The `AICc` convergence is achieved on the 34th iteration. We now use the `adj r2`:

```
## Same result as the AICc one.
stepwise_int_adjr2 <- stepwise_interactions_selection(
                                    interactions_2nd,
                                    train_data,
                                    criterion="adj.r2",
                                    iters=35)
```

`BIC` differs from `AICc` in that it penalizes the number of predictors more heavily.

```
stepwise_int_bic <- stepwise_interactions_selection(
                                    interactions_2nd,
                                    train_data,
                                    criterion="BIC",
                                    iters=45)
```

Output:

```
## loop: 1 : Interaction added: radius_worst:area_worst
## AICc: -42.21
## BIC: 88.515
## adj. R2: 0.791

## loop: 2 : Interaction added: smoothness_mean:area_se
## AICc: -72.819
## BIC: 61.692
## adj. R2: 0.805

##l oop: 3 : Interaction added: smoothness_worst:concave.points_worst
## AICc: -83.593
## BIC: 54.694
## adj. R2: 0.81

.     .     .

## loop: 15 : Interaction added: texture_mean:texture_worst
## AICc: -145.51
## BIC: 37.189
## adj. R2: 0.84

## loop: 16 : Interaction added: radius_se:compactness_se
## AICc: -146.072
## BIC: 37.189
## adj. R2: 0.84

## loop: 17 : Interaction added: area_se:concave.points_worst
## AICc: -146.548
## BIC: 40.25
## adj. R2: 0.841
```

After the 15th iteration, the BIC increases, indicating that we should stop there.

```
best_interactions_names_aicc <- unlist(
                            stepwise_int_aicc[1:33])
best_interactions_names_bic <- unlist(
                            stepwise_int_bic[1:15])
best_interactions_names_adjr2 <- unlist(
                            stepwise_int_adjr2[1:33])
```

```r
# Based on AICc (same as adj.r2)
aicc_names <-  c(names(train_data),
           best_interactions_names_aicc)
sec_ord_aicc <- as.data.frame(cbind(train_data,
      interactions_xixj(X_train)))[, aicc_names]
# Based on BIC
bic_names <-  c(names(train_data),
           best_interactions_names_bic)

sec_ord_bic <- as.data.frame(cbind(train_data,
      interactions_xixj(X_train)))[, bic_names]

aicc_model <- lm(y~., data=sec_ord_aicc)
bic_model <- lm(y~., data=sec_ord_bic)

adj_r2_values <- c(summary(aicc_model)$adj.r.squared,
                   summary(bic_model)$adj.r.squared,
                   summary(full.model)$adj.r.squared,
                   summary(vif.model)$adj.r.squared)
aicc_values <- c(AICc(aicc_model),
                 AICc(bic_model),
                 AICc(full.model),
                 AICc(vif.model))
bic_values <- c(BIC(aicc_model),
                BIC(bic_model),
                BIC(full.model),
                BIC(vif.model))
predictors <- c(aicc_model$rank - 1, bic_model$rank - 1,
                full.model$rank - 1, vif.model$rank - 1)
# Compare info. crit. values
aic_comparison <- data.frame(Model = c('AICc 2nd order',
                                       'BIC 2nd order',
                                       'Full',
                                       'Original VIF'),
                             AICc = aicc_values,
                             BIC = bic_values,
                             adj_r2 = adj_r2_values,
                             Predictors = predictors)
aic_comparison <- aic_comparison[order(aic_comparison$AICc),]
## View the comparison table ##
aic_comparison
```

The table shown below demonstrates that second-order terms provide very useful information, which dramatically improves model performance according to all three information criteria. The number of predictors in the model also increases significantly.

Model	AICc	BIC	adj_r2	Predictors
AICc 2nd order	−177.19305	68.76898	0.8587976	63
BIC 2nd order	-145.50973	37.18861	0.8395919	45
Full	28.65997	155.58683	0.7546140	30
Original VIF	252.51997	309.28249	0.5800729	12

Table 2.13: Comparing models via forward selection and information criteria (Stepwise second-order method)

The following code provides the corresponding ROC curve for each OLS model.

```
hold_out_so_aicc <- cbind(
         interactions_xixj(as.data.frame(X_hold_out)),
         second_order_terms(as.data.frame(hold_out_data)))[,
                                              aicc_names]

hold_out_so_bic <- cbind(
         interactions_xixj(as.data.frame(X_hold_out)),
         second_order_terms(as.data.frame(hold_out_data)))[,
                                              bic_names]

y_pred_aicc_2ord <- predict(aicc_model,
                            newdata=hold_out_so_aicc,
                            type="response")

y_pred_bic_2ord <- predict(bic_model,
                            newdata=hold_out_so_bic,
                            type="response")
#ROCS
roc_2oaicc_ols <- Roc_Auc(score=y_pred_bic_2ord,
                   true_class=y_real,
                   return_=FALSE,
                   model_name="OLS 2nd order (AICc)")

roc_2obic_ols <- Roc_Auc(score=y_pred_bic_2ord,
                        true_class=y_real,
                        return_=FALSE,
                        model_name="OLS 2nd order (BIC)")

mRoc_Auc(roc_ols, roc_ols_vif,
         roc_2oaicc_ols, roc_2obic_ols)
```

Similar findings are observed when alternative specifications are evaluated on a hold-out set using AUC.

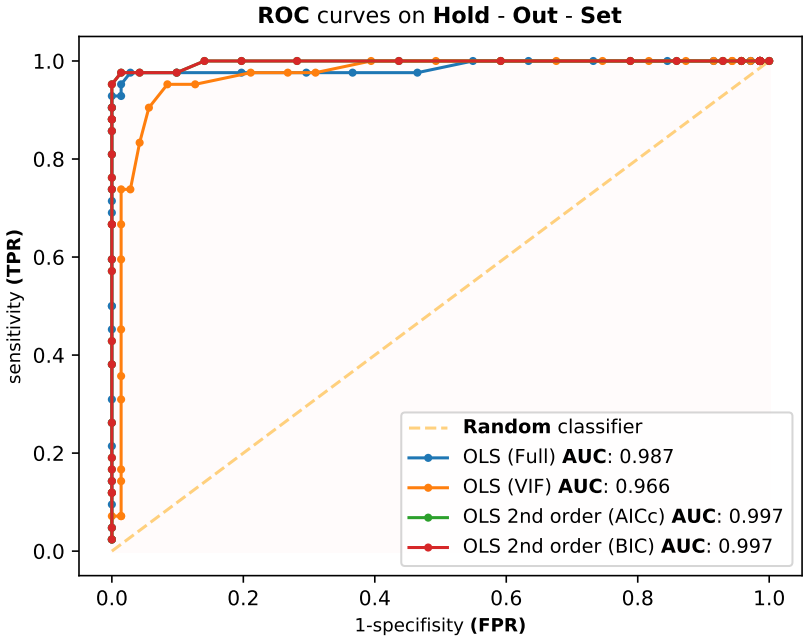


Figure 2.10: 2nd order `OLS` models (`AICc`/`BIC`) ROC curves

2.4 Improving the Use of `P-values`

`P-values` are commonly used by researchers to assess the strength of evidence against a null hypothesis, but they can be misinterpreted. This recommendation [7], can help guard against harmful `p-value` misinterpretations and move away from reliance on `'p < 0.05'`.

The `p-value` quantifies the discrepancy between the data and a null hypothesis of interest, usually the assumption of no difference or no effect. A Bayesian approach allows the calibration of `p-values` by transforming them to direct measures of the evidence against the null hypothesis, so-called `Bayes factors`.

$$\textbf{BF} = \frac{\text{Average likelihood of the observed data under the alternative hypothesis}}{\text{Likelihood of the observed data under the null hypothesis}}$$

Unfortunately, there is no unique mapping between `p-values` and `Bayes factors` because, unlike calculating the `p-value`, calculating the `Bayes factor` requires specifying an alternative hypothesis (more specifically, a prior distribution for the parameter values under the alternative hypothesis).
What can we do? Several methods have been developed for using the `p-value` to

calculate an upper bound on BF, called the Bayes factor bound (BFB), namely:

$$\mathbf{BF} \leq \mathbf{BFB} = \frac{1}{-e\ \mathtt{p} \log \mathtt{p}} \quad .$$

"Use 0.005 Instead of 0.05 as a Threshold" approach. The following table shows the value of BFB for a wide range of p-values. The table also gives the corresponding upper bound on the posterior probability of H_1. When the prior probabilities of H_0 and H_1 are equal, this upper bound on the posterior probability of H_1 is given by $P^U(H_1 \mid p) = \mathtt{BFB}/(1 + \mathtt{BFB})$.

Table 2.14: Evidence against the null hypothesis

p-value	0.1	0.05	0.01	0.005	0.001	0.0001	1e-05
BFB	1.6	2.46	7.99	13.89	53.26	399.42	3195.36
$\mathtt{P}^U(H_1 \mid \mathtt{p})$	0.615	0.7107	0.8887	0.9328	0.9816	0.9975	0.9997

Table 2.14 is taken from [7]. For example, a p-value of 0.05 corresponds to a Bayes factor of at most 2.46:1. That is, the data imply odds in favor of the alternative hypothesis relative to the null hypothesis of at most 2.46 to 1. So if the null and alternative hypotheses were originally equally likely, there remains, at least, a 29% chance that the null hypothesis is true. On the other hand, a p-value of **0.005** indicates the alternative hypothesis is at most **≈14** times as likely as the null hypothesis.

2.5 Optimization Techniques for Estimation

2.5.1 Newton-Raphson Method

Newton's method, or the Newton-Raphson method, is an optimization approach for finding roots of nonlinear equations. It is well-regarded for its rapid quadratic convergence. The technique involves iteratively approximating the function with a Taylor expansion and refining the estimate until it reaches the best solution. Let θ^* denote the true/optimal value of the parameter and let $\theta^{(t)}$ represent the parameter value at iteration t, then, as $t \to \infty$:

$$\frac{\|\theta^{(t+1)} - \theta^*\|}{\|\theta^{(t)} - \theta^*\|^2} \to \ \mathtt{Constant}.$$

Taylor expansion around the current iterate $\theta^{(t)}$:

$$L(\theta) \approx L(\theta^{(t)}) + [\nabla L(\theta^{(t)})]^T(\theta - \theta^{(t)}) + \frac{1}{2}(\theta - \theta^{(t)})^T[\nabla^2 L(\theta^{(t)})]^T(\theta - \theta^{(t)})$$

To get the maximum of the above function, we equate its gradient to zero:

$$\nabla L(\theta^{(t)}) + [\nabla^2 L(\theta^{(t)})]^T(\theta - \theta^{(t)}) = \mathbf{0},$$

which leads to the next iterate (**Naive Newton's method**):

$$\theta^{(t+1)} = \theta^{(t)} - \left([\nabla L(\theta^{(t)})]^T\right)^{-1}\nabla L(\theta^{(t)}) \ ,$$

or

$$\theta^{(t+1)} = \theta^{(t)} - \underbrace{H^{-1}(\theta^{(t)})}_{\texttt{Hessian}}\nabla L(\theta^{(t)}) \ .$$

However, naive Newton's method presents some challenges, such as the requirement to derive, evaluate and invert the observed information matrix, which can be computationally demanding. Several solutions exist for these problems, including leveraging the Hessian's structure, applying automatic differentiation, numerical differentiation, or using quasi-Newton[7] techniques. Additionally, the naive method doesn't ensure an ascent algorithm, meaning it could go in an undesired direction. To tackle this, one can use a positive definite approximation of the Hessian along with line search methods.

It is important to mention that Raphson[8], independently, developed a similar algorithm for finding roots of equations around 1690, which he published in his book [27]. His work simplified Newton's original formulation, making it more accessible and widely known.

2.5.2 Fisher Scoring

Fisher scoring is a variant of Newton's method, specifically adapted to handle statistical models. The main idea behind Fisher scoring is to approximate the Hessian matrix (second derivative) in Newton's method with the expected or average curvature of the function, known as the Fisher information matrix. This approximation often results in a more stable and accurate estimation of the parameters compared to the naive Newton's method. Fisher information matrix:

$$\mathbf{I}(\theta) = \mathbb{E}\big\{ -\nabla^2 L(\theta)\big\} \geq \mathbf{0_{n\times n}} \ .$$

Hence, the Fisher's scoring algorithm iterates according to:

$$\theta^{(t+1)} = \theta^{(t)} + s[\mathbf{I}(\theta^{(t)})]^{-1}\nabla L(\theta^{(t)}) \ .$$

Fisher scoring is widely used in statistical estimation, especially in the context of generalized linear models, logistic regression and other problems where maximum likelihood estimation is the objective.

[7] quasi-Newton methods more computationally efficient and suitable for high-dimensional problems

[8] English mathematician Joseph Raphson (1648-1715)

Sir Ronald Aylmer Fisher[9] is widely regarded as one of the key pioneers of modern statistics and, for many, he is considered the father of statistics. His groundbreaking contributions to the field have shaped the way we approach statistical analysis and data interpretation to this day.

Furthermore, the `glm()` function in `R` uses the iteratively reweighted least squares (`IRLS`) algorithm for estimation, which is a form of Fisher scoring.

2.5.3 Stochastic Gradient Descent

Gradient Descent is a widely used optimization technique. Generally, it aims to minimize a loss function. The update formula for Gradient Descent is as follows:

$$\theta^{(n+1)} = \theta^{(n)} - \alpha \nabla L\left(\theta^{(n)}\right)$$

Here, θ represents the model parameters and α is the learning rate or step size.

Clearly, Gradient Descent is a modified version of Newton's method, obtained by simplifying the Hessian matrix approximation. When we replace the Hessian matrix with an identity matrix and incorporate a scalar learning rate, Gradient Descent transforms into a first-order optimization technique that depends exclusively on the gradient of the loss function, while still preserving the iterative structure of Newton's method.

Note that, in classical Gradient Descent, the gradient is computed using the entire dataset, which means that it takes the average of the gradients for all data points before updating the model parameters. This can be computationally expensive, especially for large datasets, as it requires processing all data points before making a single update.

Stochastic Gradient Descent (often abbreviated SGD) is an iterative optimization technique for objective functions with appropriate smoothness properties (such as differentiable or subdifferentiable functions). It can be viewed as a stochastic version of gradient descent optimization, as it substitutes the true gradient (computed using the whole dataset) with an estimation (computed from a randomly chosen subset of the data). This approach significantly reduces the computational burden, particularly in high-dimensional optimization problems, offering quicker iterations at the expense of a slower convergence rate.

Although the fundamental concept of stochastic approximation can be traced back to the Robbins-Monro[10] algorithm from the 1950s, SGD has gained significant importance as an optimization method in the field of machine learning. This method is particularly effective in addressing high-dimensional problems by enabling faster iterations while maintaining reasonable convergence rates.

[9] British statistician and geneticist (1890-1962)

[10] A stochastic optimization method that was introduced in 1951, named after Herbert Robbins and Sutton Monro

3

Stepwise Feature Selection

In this chapter, we will discuss the concept of stepwise feature selection. This method incorporates information criteria that we've previously introduced, such as the Akaike information criterion (`AIC`), corrected Akaike information criterion (`AICc`) and Bayesian information criterion (`BIC`). We will then evaluate the models produced using this method in comparison to the models we created in earlier chapters.
`R` provides the `stepAIC` method in the `MASS` package, which allows for stepwise selection using `AIC` or `BIC` criteria. In addition, the `boot.stepAIC` method in the `bootStepAIC` package, developed by Dimitris Rizopoulos[1] in 2022, will be examined in the upcoming Bootstrap chapter.

3.1 Information Criteria

An information criterion is a statistical measure used to compare the goodness of fit of different models while accounting for the complexity of each model. The goal is to identify the model that best balances the fit to the data with the simplicity of the model, thereby avoiding overfitting. Information criteria often involve a trade-off between the likelihood of the model given the data and a penalty term for the number of parameters used.

3.1.1 Akaike Information Criterion (AIC)

The Akaike Information Criterion (`AIC`) is a mathematical measure used for model selection in statistical analysis. It was developed by the Japanese statistician Hirotugu Akaike in 1973. The `AIC` helps to determine the best-fitting model among a set of candidate models by estimating the relative amount of information lost when a given model is used to approximate the underlying process that generates the data. The criterion balances the goodness-of-fit and

[1] Professor of Biostatistics at the Erasmus Medical Center Rotterdam

model complexity, aiming to prevent overfitting.

`AIC` is defined as:

$$\texttt{AIC} = 2k - 2\log(\mathcal{L}),$$

where: k is the number of estimated parameters in the model and $\mathcal{L}$ is the maximum likelihood of the model.

The model with the lowest `AIC` value is considered to be the best fit. The `AIC` is useful in situations where there is no single "true" model and the goal is to find the most parsimonious and informative model for the given data.

Historically, the `AIC` was introduced by Akaike[2] in a 1974 paper [2]. Akaike's contribution to statistics [1] was recognized with several awards, including the Kyoto Prize in 2006.
In comparison to `p-values`, `AIC` offers several advantages for model selection:

- Parsimony: `AIC` encourages the selection of simpler models by penalizing model complexity. `p-values`, on the other hand, are more focused on testing individual hypotheses and do not account for model complexity.
- Relativity: `AIC` allows for the comparison of multiple candidate models, even if they are not nested. `p-values` are typically used for hypothesis testing, making them less suitable for model comparison.
- Less reliance on sample size: `AIC` is less sensitive to sample size than `p-values`. As the sample size increases, `p-values` tend to become smaller, which can lead to the rejection of null hypotheses even when the effect size is trivial. `AIC`, by contrast, remains relatively stable across different sample sizes.

Is important to note that `AIC` is not a perfect metric and may not always lead to the best model selection, especially when the sample size is small or the models being compared are very different.

3.1.2 Corrected Akaike Information Criterion (AICc)

The corrected Akaike (`AICc`) is a modification of the original `AIC`, introduced to improve its performance when dealing with small sample sizes. The `AICc` was first proposed by Satoshi Sakamoto, Mitsuhiro Sato and Genshiro Kitagawa in their 1986 paper [28]

`AICc` is defined as:

$$\texttt{AICc} = \texttt{AIC} + \frac{2k(k+1)}{n-k-1} = -2\log(\mathcal{L}) + 2k + \frac{2k(k+1)}{n-k-1}$$

[2] Japanese statistician in the early 1970s

The main difference between AIC and AICc is the additional term in the AICc formula that accounts for the finite sample size. This term acts as a correction factor, which becomes more significant as the sample size decreases. When the sample size is large, the correction term converges to zero and AICc approaches the original AIC.

The AICc is useful because it provides a more accurate measure of the relative information lost when a given model is used to approximate the underlying process that generates the data, particularly when the sample size is small. In such cases, AIC may underestimate the true information loss, leading to overfitting and the selection of overly complex models.

AICc should be used over AIC when the sample size is small or when the ratio of the sample size to the number of parameters is low (less than 40, as a rule of thumb). By accounting for the sample size in its calculation, AICc helps to prevent overfitting and select more parsimonious models, particularly when dealing with limited data.

3.1.3 Bayesian Information Criterion (BIC)

The Bayesian Information Criterion (BIC), also known as the Schwarz Information Criterion (SIC), is another model selection criterion used in statistical analysis. It was developed by the German statistician Gideon Schwarz in 1978. Like the AIC, the BIC aims to balance the goodness-of-fit and model complexity, but it does so with a different penalty term for the number of parameters. The BIC is derived from a Bayesian perspective and is asymptotically consistent, meaning that as the sample size grows, the BIC will almost surely select the true model if it is among the candidate models. BIC is defined as:

$$\texttt{BIC} = -2\log(\mathcal{L}) + k \cdot \log(n) \ .$$

The model with the lowest BIC value is considered to be the best fit. The main difference between AIC and BIC lies in the penalty term for the number of parameters. The BIC has a stronger penalty for model complexity, which increases logarithmically with the sample size. This can make BIC more conservative than AIC in terms of selecting simpler models.

Some advantages of using BIC over AIC include:

- Asymptotic consistency: BIC is asymptotically consistent, which means that as the sample size grows, it will almost surely select the true model if it is among the candidate models. AIC does not have this property.
- Stronger penalty for complexity: BIC penalizes model complexity more heavily than AIC, leading to the selection of simpler models. This can be beneficial in situations where overfitting is a concern.

- Bayesian interpretation: `BIC` has a Bayesian foundation and its penalty term can be interpreted as a prior distribution on model complexity. This makes `BIC` more appealing to researchers who prefer a Bayesian approach to model selection.

However, `BIC` also has some limitations. For example, `BIC` is derived under the assumption of a large sample size, which may not always be the case in practice. When dealing with small sample sizes, `AICc` may be a more appropriate choice.

3.2 Forward-Backward Selection

Stepwise selection offers a more computationally efficient approach compared to best subset selection [23]. Rather than examining all 2^p potential models comprised of various subsets of the p predictors, as in best subset selection, Stepwise selection evaluates a significantly smaller collection of models.

Algorithm 1 Forward Stepwise Selection

Input: Dataset **D**, predictors **p**
Output: Selected Variables

1: Let $\mathbf{M}_0$ denote the null model, which contains no predictors.
2: **for** $k = 0, ..., p-1$ **do**
3: Consider all `p - k` models that augment the predictors in $\mathbf{M_k}$ with one additional predictor.
4: Choose the best among these `p - k` models and call it $\mathbf{M_{k+1}}$ //best is defined as having highest `ROC AUC, F1-score,` ...
5: **end for**
6: Select a single best model from among $\mathbf{M}_0, \ldots, \mathbf{M}_p$ using `AICc`/`BIC`

This amounts to a total of $1 + \sum_{k=0}^{p-1}(p-k) = 1 + p(p+1)/2$ models. This is a substantial difference: i.e. if $p = 30$, best subset selection requires fitting 1.073.741.824 models, whereas forward stepwise requires fitting only 466 models.

The `stepAIC` is a function available in the `MASS` package, used for performing stepwise model selection based on `AIC` (or `BIC`).

The main arguments for the `stepAIC` function are:

- `object`: A fitted model object, usually a starting model (for `"backward"`), null model (for `"forward"`) intermediate model (for `"both"`.
- `scope`: A list of lower and upper limits for the set of models being considered.
- `direction`: Specifies the search direction, which can be `"forward"`, `"backward"`, or `"both"`.
- `k`: A constant used for `AIC` calculation (for `BIC` set `k`=log(n)).

3.2.1 Forward

When conducting forward search, it is crucial to appropriately set the scope argument. The scope should specify the "minimum" (lower) and "maximum" (upper) models that encompass the range of models to be explored. Failing to provide the scope argument will result in the maximum model being assumed to be the starting model, which will cause `stepAIC` to skip the search process entirely.

```
# Set k to log(n) to use BIC criterion
n <- nrow(train_data)
logn <- log(n)

suppressWarnings({

    # Fit an empty model with only the intercept
    null.model <- glm(y~ 1, data=train_data,
                      family=binomial(link='logit'))

    # fit a full MLE model
    MLE_full.model <- glm(y~., data=train_data,
                      family=binomial(link='logit'))

    ## Forward Selection ##

    system.time(forward_aic <- stepAIC(null.model,
                        scope=list(lower=null.model,
                                   upper=MLE_full.model),
                        direction="forward",
                        trace=0))

    # Using BIC
    system.time(forward_bic <- stepAIC(null.model,
                        scope=list(lower=null.model,
                                   upper=MLE_full.model),
                        direction="forward",
                        trace=0,
                        k=logn))
    })
```

Let's view every step:

```
forward_aic$anova[, c("Step", "AIC")] #AIC
forward_bic$anova[, c("Step", "AIC")] #BIC
```

The tables shown below present the results of the forward selection procedure based on `AIC` and `BIC`. It is worth noting that the number of predictors is substantially reduced relative to the OLS models discussed previously. `AIC` and `BIC` values are not comparable to the ones from OLS regression as they are

derived using different likelihoods.

Table 3.1: Based on `AIC`

	Step	AIC
1	Null model	604.31
2	+ perimeter_worst	172.46
3	+ smoothness_worst	114.72
4	+ texture_worst	85.12
5	+ concave.points_mean	77.37
6	+ area_worst	73.17
7	+ perimeter_mean	64.55
8	+ symmetry_worst	63.58
9	+ compactness_mean	61.50
10	+ concave.points_se	58.86

Table 3.2: Based on `BIC`

	Step	BIC
1	Null model	608.44
2	+ perimeter_worst	180.71
3	+ smoothness_worst	127.08
4	+ texture_worst	101.61
5	+ concave.points_mean	97.98
6	+ area_worst	97.91
7	+ perimeter_mean	93.40

Table 3.3: `Forward procedure`

3.2.2 Backward

The same analysis was performed using backward instead of forward selection. The outcomes depicted in the tables that follow, suggest that substansially larger number of predictors is included in logistic regressions when one utilizes backward selection. Furthermore, the values of the information criteria are substansially improved. Hence, one would expect that backward selection will produce specificaitions with improved generalization capability relative to forward selection.

```
suppressWarnings({
    ## Backward Selection ##
    system.time(backward_aic <- stepAIC(MLE_full.model,
                          scope=list(lower=null.model,
                                     upper=MLE_full.model),
                          direction="backward", trace=0))
    # Using BIC
    system.time(backward_bic <- stepAIC(MLE_full.model,
                          scope=list(lower=null.model,
                                     upper=MLE_full.model),
                          direction="backward", trace=0, k=logn))
    })
```

Let's take a closer look at each individual step:

```
backward_aic$anova[, c("Step", "AIC")] #AIC
backward_bic$anova[, c("Step", "AIC")] #BIC
```

Table 3.4: Based on AIC

	Step	AIC
1	Full model	62.00
2	- concave.points_worst	60.00
3	- radius_mean	58.00
4	- perimeter_worst	56.00
5	- perimeter_se	54.00
6	- smoothness_mean	52.00
7	- fractal_dimension_mean	50.00
8	- area_se	48.00
9	- concavity_mean	46.00
10	- area_mean	44.00
11	- texture_mean	42.00
12	- perimeter_mean	40.00
13	- fractal_dimension_worst	38.00
14	- fractal_dimension_se	36.00
15	- compactness_se	34.00
16	- smoothness_se	32.00
17	- texture_se	30.00

Table 3.5: Based on BIC

	Step	BIC
1	Full model	189.80
2	- concave.points_worst	183.67
3	- radius_mean	177.55
4	- perimeter_worst	171.43
5	- perimeter_se	165.31
6	- smoothness_mean	159.18
7	- fractal_dimension_mean	153.06
8	- area_se	146.94
9	- concavity_mean	140.82
10	- area_mean	134.69
11	- texture_mean	128.57
12	- perimeter_mean	122.45
13	- fractal_dimension_worst	116.33
14	- fractal_dimension_se	110.20
15	- compactness_se	104.08
16	- smoothness_se	97.96
17	- texture_se	91.84

Table 3.6: Backward procedure

3.2.3 Both

The "both" selection is useful if starting from an intermediate model. It removes the problems associated with "backward" and "forward" selections and starts searching from an intermediate model. The tables shown below illustrate that such an approach provides inferior performance in terms of AIC and BIC relative to backward selection.

```
correlations <- cor(data)[, 'y'][-1]

# Set a correlation threshold (e.g., 0.5 or -0.5)
threshold <- 0.5

# Identify predictors with correlation above
# the threshold or below the negative threshold
selected_predictors <- names(correlations[abs(correlations)>threshold])

# Create a formula for the initial intermediate model
model_formula <- as.formula(paste("y ~",
                                  paste(selected_predictors,
                                        collapse = '+')))

# Fit the initial intermediate model (MLE fit <- binary response)
intermediate.model <- glm(model_formula, data = train_data,
                          family=binomial(link='logit'))
```

```r
length(names(intermediate.model$coeff[-1])) ## 15 predictors

### "Both" selection: ###
suppressWarnings({

    system.time(both_aic <- stepAIC(intermediate.model,
                          scope=list(lower=null.model,
                                     upper=MLE_full.model),
                          direction="both",
                          trace=0))

    # Using BIC

    system.time(both_bic <- stepAIC(intermediate.model,
                          scope=list(lower=null.model,
                                     upper=MLE_full.model),
                          direction="both",
                          trace=0,
                          k=logn))
    })
```

All steps for `'both'` direction are shown below:

```r
both_aic$anova[, c("Step", "AIC")] #AIC
both_bic$anova[, c("Step", "AIC")] #BIC
```

Table 3.7: Based on `AIC`

	Step	AIC
1	Intermediate model	99.07
2	+ texture_worst	76.40
3	- area_se	74.40
4	- perimeter_mean	72.43
5	- compactness_worst	70.51
6	- radius_worst	68.62
7	- perimeter_se	67.08
8	- perimeter_worst	65.58
9	+ symmetry_worst	63.85
10	- concavity_mean	62.07
11	- area_mean	60.19
12	- concavity_worst	59.26
13	+ smoothness_worst	58.78

Table 3.8: Based on `BIC`

	Step	BIC
1	Intermediate model	165.03
2	+ texture_worst	146.48
3	- area_se	140.36
4	- perimeter_mean	134.27
5	- compactness_worst	128.22
6	- radius_worst	122.22
7	- perimeter_se	116.55
8	- perimeter_worst	110.92
9	- area_mean	105.46
10	- concavity_mean	100.15
11	- radius_se	95.03
12	- concave.points_worst	90.01
13	- concavity_worst	88.63
14	- compactness_mean	84.56

Table 3.9: `Both procedure`

3.3 Model Selection Bias

In 2010, Gergely Hegyi and László Zsolt Garamszegi published a paper [21] on stepwise selection methods. In the context of stepwise feature selection, the authors suggest moving away from traditional stepwise regression methods towards the information theoretic (IT) approach. Stepwise regression methods, which involve adding or removing predictors based on their statistical significance, have been criticized for their subjectivity, model uncertainty and potential for parameter estimation bias.

This paper indicates that in stepwise regression, the model is simplified according to the estimates derived from the dataset, which means adjusting the model to fit the data. This process can increase the chances of overestimated effect sizes in the final model, a phenomenon known as "data dredging" (Copas and Long 1991; Chatfield 1995; Forstmeier and Schielzeth 2010). Studies have revealed that overestimation is most significant in weak predictors (Sauerbrei 1999), which constitute the majority of biological predictors (Møller and Jennions 2002). To mitigate estimation bias in stepwise regression, one can test the final model on a separate, independent dataset (Harrell et al. 1984; Moonesinghe et al. 2007) or employ bootstrap resampling on the same sample (Sauerbrei 1999). Parameter estimation bias, also known as "model selection bias", is among the most severe drawbacks of stepwise regression (Whittingham et al. 2006).

Conversely, information theory has been suggested as a means to circumvent the "model selection bias" inherent in stepwise methods by comparing a fixed set of models simultaneously using a single information criterion. As a result, the model selection outcome reflects the entire candidate model set and the set is not tailored to the data (Burnham and Anderson 2002). However, information theory is not entirely devoid of bias in model selection.

3.4 Stepwise Regression Results

In the tables presented below (Table 3.10 & Table 3.11), we can observe the selected predictors from the `StepAIC()` regression (based on `AIC` and `BIC` respectively), along with their respective coefficients, standard errors and `p-values` determined by a `z-test`. In the tables below, we can observe that the backward regression might need more iterations to converge due to high `p-values`.

	Forward			Backward			Both		
	Est.	Std. Error	Pr(>|z|)	Est.	Std. Error	Pr(>|z|)	Est.	Std. Error	Pr(>|z|)
(Intercept)	-31.49393	14.06607	0.02516	-7540.76	78724.85	0.924	-35.70254	16.94450	0.035115
perimeter_worst	0.13001	0.18545	0.48327						
smoothness_worst	68.24384	40.47271	0.09176 .				62.86661	42.08121	0.135192
texture_worst	0.47886	0.12735	0.00017	167.87	1662.93	0.920	0.50705	0.14744	0.000584
concave.points_mean	217.16801	74.39950	0.00351	42207.74	423394.54	0.921	186.39959	70.62414	0.008307
area_worst	0.03917	0.01446	0.00675	16.46	183.38	0.928	0.03424	0.01256	0.006412
perimeter_mean	-0.54316	0.22035	0.01370						
symmetry_worst	24.07281	11.98382	0.04456	38367.56	377857.77	0.919	25.23948	12.22855	0.039020
compactness_mean	-60.82872	28.79925	0.03467	-20587.88	209488.25	0.922	-68.69845	28.75932	0.016906
concave.points_se	265.94916	151.12133	0.07844	276314.90	2914921.83	0.924			
symmetry_mean				-46018.70	463326.91	0.921			
radius_se				11891.20	117976.82	0.920	11.27836	6.69524	0.092078
concavity_se				-62426.49	688511.38	0.928			
concave.points_se									
symmetry_se				-119222.56	1182816.53	0.920			
radius_worst				-1243.48	14261.03	0.931			
area_worst									
smoothness_worst				13215.29	135200.17	0.922			
compactness_worst				-7778.18	78824.12	0.921			
concavity_worst				14444.18	157291.67	0.927			
concave.points_worst							38.02533	28.64887	0.184413

Table 3.10: `StepAIC` summary based on `AIC`

	Forward			Backward			Both		
	Est.	Std. Error	Pr(>|z|)	Est.	Std. Error	Pr(>|z|)	Est.	Std. Error	Pr(>|z|)
(Intercept)	-13.83188	9.53444	0.146856	-7540.76	78724.85	0.924	-3.438918	5.379468	0.522649
perimeter_worst	0.02685	0.13388	0.841042						
smoothness_worst	37.73704	29.93624	0.207460						
texture_worst	0.33069	0.07882	2.72e-05	167.87	1662.93	0.920	0.317122	0.075956	2.98e-05
concave.points_mean	157.25246	41.93646	0.000177	42207.74	423394.54	0.921	155.451964	33.920791	4.59e-06
area_worst	0.03858	0.01238	0.001824	16.46	183.38	0.928	0.040174	0.009337	1.69e-05
perimeter_mean	-0.44934	0.15388	0.003498						
compactness_mean				-20587.88	209488.25	0.922			
symmetry_mean				-46018.70	463326.91	0.921			
radius_se				11891.20	117976.82	0.920			
concavity_se				-62426.49	688511.38	0.928			
concave.points_se				276314.90	2914921.83	0.924			
symmetry_se				-119222.56	1182816.53	0.920			
radius_worst				-1243.48	14261.03	0.931			
compactness_worst				-7778.18	78824.12	0.921			
concavity_worst				14444.18	157291.67	0.927			
symmetry_worst				38367.56	377857.77	0.919			
radius_mean							-3.163829	0.876025	0.000304

Table 3.11: `StepAIC` summary based on `BIC`

```r
#Compare StepAIC/BIC models via AICc/BIC

aicc_values <- c(AICc(forward_aic), AICc(backward_aic),
                 AICc(both_aic), AICc(forward_bic),
                 AICc(backward_bic), AICc(both_bic))

bic_values <- c(BIC(forward_aic), BIC(backward_aic),
                BIC(both_aic), BIC(forward_bic),
                BIC(backward_bic), BIC(both_bic))

predictors <- c(dim(forward_aic$model[-1])[2],
                dim(backward_aic$model[-1])[2],
                dim(both_aic$model[-1])[2],
                dim(forward_bic$model[-1])[2],
                dim(backward_bic$model[-1])[2],
                dim(both_bic$model[-1])[2])
# Compare info. crit. values
stepAICcBICtable <- data.frame(Model = c('forward/AIC',
                                         'backward/AIC',
                                         'both/AIC',
                                         'forward/BIC',
                                         'backward/BIC',
                                         'both/BIC'),
                               AICc = aicc_values,
                               BIC = bic_values,
                               Predictors = predictors)
## View the comparison table ##
stepAICcBICtable #best AICc -> backward.(aic/bic)
                 #best BIC  -> both.bic
```

	StepAIC model	AICc	BIC	Predictors
1	forward/AIC	59.36	100.09	9
2	backward/AIC	31.09	91.84	14
3	both/AIC	59.27	100.00	9
4	forward/BIC	64.80	93.40	6
5	backward/BIC	31.09	91.84	14
6	both/BIC	64.08	84.56	4

Table 3.12: StepAIC AICc/BIC/#Predictors

```r
best_stepAIC.data <- backward_aic$model
best_stepBIC.data <- both_bic$model
```

3.5 StepAIC for Second Order Models

We will implement the StepAIC regression for each second-order model selected by the stepwise method (based on both AICc and BIC criteria). To begin, we will establish an intermediate model for each criterion:

```
# Compute correlations between y and other predictors
correlations_aicc <- cor(sec_ord_aicc)[, 'y'][-1]
correlations_bic <- cor(sec_ord_bic)[, 'y'][-1]

# Set a correlation threshold (e.g., 0.5 or -0.5)
threshold <- 0.5

# Identify predictors with correlation above
# the threshold or below the negative threshold
selected_predictors_aicc <- names(correlations_aicc[
                  abs(correlations_aicc)>threshold])
selected_predictors_bic <- names(correlations_bic[
                 abs(correlations_bic)>threshold])

# Create a formula for the initial intermediate model
model_formula_aicc <- as.formula(paste("y ~",
                                paste(selected_predictors_aicc,
                                      collapse = '+')))
model_formula_bic <- as.formula(paste("y ~",
                                paste(selected_predictors_bic,
                                      collapse = '+')))

suppressWarnings({
# Fit the initial intermediate model
intermediate_bic.model <- glm(model_formula_bic,
                              data=sec_ord_bic,
                              family=binomial(link='logit'))

intermediate_aicc.model <- glm(model_formula_aicc,
                              data=sec_ord_aicc,
                              family=binomial(link='logit'))
})
```

For second order model, based on AICc:

```
### For second order model based on stepwise using AICc

suppressWarnings({

    # Fit an empty model with only the intercept
    null.model <- glm(y~ 1, data=sec_ord_aicc,
                      family=binomial(link='logit'))
```

```
    # fit a full MLE model
    MLE_full.model <- glm(as.formula(paste("y ~",
                          paste(names(sec_ord_aicc)[-1],
                          collapse = '+'))),
                          data=sec_ord_aicc,
                          family=binomial(link='logit'))

    ## Stepwise Selection ##

    system.time(sord_forward_aic <- stepAIC(null.model,
                         scope=list(lower=null.model,
                                    upper=MLE_full.model),
                         direction="forward",
                         trace=0))

    system.time(sord_backward_aic <- stepAIC(MLE_full.model,
                         scope=list(lower=null.model,
                                    upper=MLE_full.model),
                         direction="backward",
                         trace=0))

    system.time(sord_both_aic <- stepAIC(intermediate_aicc.model,
                         scope=list(lower=null.model,
                                    upper=MLE_full.model),
                         direction="both",
                         trace=0))
    })
```

For second order model, based on BIC:

```
### For second order model based on stepwise using BIC
n <- nrow(sec_ord_bic)
logn <- log(n)

suppressWarnings({
    # Fit an empty model with only the intercept
    null.model <- glm(y~ 1, data=sec_ord_bic,
                      family=binomial(link='logit'))
    # fit a full MLE model
    MLE_full.model <- glm(as.formula(paste("y ~",
                          paste(names(sec_ord_bic)[-1],
                          collapse = '+'))),
                          data=sec_ord_bic,
                          family=binomial(link='logit'))

    ## Stepwise Selection ##

    # Using BIC
    system.time(sord_forward_bic <- stepAIC(null.model,
                         scope=list(lower=null.model,
```

```
                                    upper=MLE_full.model),
                          direction="forward",
                          trace=0,
                          k=logn))
    system.time(sord_backward_bic <- stepAIC(MLE_full.model,
                          scope=list(lower=null.model,
                                    upper=MLE_full.model),
                          direction="backward",
                          trace=0,
                          k=logn))
    system.time(sord_both_bic <- stepAIC(intermediate_bic.model,
                          scope=list(lower=null.model,
                                    upper=MLE_full.model),
                          direction="both",
                          trace=0,
                          k=logn))
    })
```

The code below compares the above models (based on `AICc` & `BIC`).

```
aicc_values <- c(AICc(sord_forward_aic), AICc(sord_backward_aic),
                 AICc(sord_both_aic), AICc(sord_forward_bic),
                 AICc(sord_backward_bic), AICc(sord_both_bic))

bic_values <- c(BIC(sord_forward_aic), BIC(sord_backward_aic),
                BIC(sord_both_aic), BIC(sord_forward_bic),
                BIC(sord_backward_bic), BIC(sord_both_bic))
predictors <- c(sord_forward_aic$rank-1, sord_backward_aic$rank-1,
                sord_both_aic$rank-1, sord_forward_bic$rank-1,
                sord_backward_bic$rank-1, sord_both_bic$rank-1)
# Compare info. crit. values
stepAICBICtable <- data.frame(Model = c(
                                    '2nd Order (AICc) StepAIC Forward',
                                    '2nd Order (AICc) StepAIC Backward',
                                    '2nd Order (AICc) StepAIC Both',
                                    '2nd Order (BIC) StepBIC Forward',
                                    '2nd Order (BIC) StepBIC Backward',
                                    '2nd Order (BIC) StepBIC Both'),
                              AICc = aicc_values,
                              BIC = bic_values,
                              Predictors = predictors)
## View the comparison table ##
stepAICBICtable #backward for AICc 2o model seems the best fit.
```

Table 3.13 presents a comparison of various models. It reveals that the backwards stepwise regression (which is based on the `AIC`), yields the lowest `AIC` and `BIC`. Thus, it seems that the stepwise second-order model, which is based on `AICc`, offers a better fit than the other models.

	Model	AICc	BIC	Predictors
1	2nd Order (AICc) StepAIC Forward	59.36	100.09	9
2	2nd Order (AICc) StepAIC Backward	24.70	73.47	11
3	2nd Order (AICc) StepAIC Both	26.82	79.59	12
4	2nd Order (BIC) StepBIC Forward	64.80	93.40	6
5	2nd Order (BIC) StepBIC Backward	26.82	79.59	12
6	2nd Order (BIC) StepBIC Both	33.24	97.96	15

Table 3.13: AICc/BIC for StepAIC second order models

We will now save the best model observed, which is determined by the minimum values of both AICc and BIC.

```
best_sec_order.model <- as.matrix(sec_ord_aicc[,
                    c('y',names(coef(sord_backward_aic)[-1]))])
names(as.data.frame(best_sec_order.model)[-1])
```

compactness_mean · concavity_mean · fractal_dimension_mean · radius_se · compactness_se · texture_worst · area_worst · concave.points_worst · symmetry_worst · compactness_mean:compactness_se · concavity_mean:fractal_dimension_mean

3.6 stepAIC using Corrected Akaike Information Criterion

In the MASS package, the stepAIC function employs the formula k· p - 2log($\mathcal{L}$) to evaluate the quality of a model, where p represents the number of estimated parameters in the model, $\mathcal{L}$ represents the likelihood and k is set to 2 for AIC or log(n) for BIC. In this section, we present an alternative approach using the corrected Akaike Information Criterion (AICc) instead of AIC or BIC. We have developed the following functions to achieve this:

```
library(MASS)
library(AICcmodavg)

step <- function (init.model, data,
                  direction="forward",
                  trace=FALSE) {

    ## full forward:  init-> null.model ##
    ## full backward: init-> full.model ##

    # define the response var. given init.model:
    y <- as.character(terms(init.model)[[2]])

    AICcs <- list(AICc(init.model))
```

```
    predictors <- colnames(data)
    predictors <- predictors[!predictors %in% y]

    best_aicc <- AICcs[[1]];
    loop <- 1;

        if (direction == "forward") {
            best_predictor <- '';
     #Leave out the predictors in init.model (if not null) for searching
            if (length(names(init.model$coef))!=1){
                leave_out_predictors <- match(names(init.model$coef)[-1],
                                                    predictors)
                predictors <- predictors[-leave_out_predictors]
            }
            # Loop through (remaining) predictors
            for (predictor in predictors) {
              # Add predictor to the null model
              model <- suppressWarnings(
                        update(init.model,
                            as.formula(paste("~ . +",
                                        predictor)))
                       )

              # Calculate AICc
              aicc <- AICc(model)

              if (trace) {
                  loop = loop + 1
                  cat("iter. ", loop-1, ': model: ')
                  print(model$call)
                  cat(" -> AICc: ", aicc, '\n\n')
              }

              # Check if AICc is lower than the best AICc
              if (aicc < best_aicc) {
                best_aicc <- aicc
                best_predictor <- predictor
              }
            }

            return (list(best_pred=best_predictor,
                         aicc=best_aicc))
    }

    else if (direction == "backward") {

          worst_predictor <- ""

          # Use predictors already in the model instead
```

```
          predictors <- names(init.model$coef)[-1]

          # Loop through predictors
          for (predictor in predictors) {
            # Remove predictor from the model
            reduced_formula <- as.formula(paste(y, "~ . -",
                                                predictor))
            reduced_model <- suppressWarnings(
                                      update(init.model,
                                             reduced_formula,
                                             data = data))
            aicc <- AICc(reduced_model)

            if (trace) {
                  loop = loop + 1
                  cat("iter. ", loop-1, ': model: ')
                  print(reduced_model$call)
                  cat(" -> AICc: ", aicc, '\n\n')
              }

            if (aicc < best_aicc) {
              worst_predictor <- predictor
              best_aicc <- aicc
            }
          }
          # Return the worst predictor and its AICc value
          return(list(worst_pred=worst_predictor,
                      aicc=best_aicc))
        }
    }
```

Inputs of `step()`:

- ◇ `init.model`: Initial model `c(lm, glm, aov, ...)`
- ◇ `data`: training data: `as.data.frame, as.matrix`
- ◇ `direction`: `c('forward', 'backward')`
- ◇ `trace`: bool

Output: `c(best predictor, AICc)` for `'forward'`, `c(worst predictor, AICc)` for `'backward'`, given the initial model. The following function utilizes the `step()` function by beginning with the initial model and then, depending on the direction, adds or subtracts the output predictor of the `step()` method. This procedure is known as stepwise selection.

```
stepAICc <- function(init.model, data,
                     direction = "forward",
                     trace = FALSE) {
```

```
y <- as.character(terms(init.model)[[2]])
predictors <- colnames(data)
predictors <- predictors[!predictors %in% y]
anova <- data.frame(step=character(),
                    AICc=numeric())
#add the AICc of the init model:
anova <- rbind(anova,
               data.frame(step="Initial model",
                          AICc=AICc(init.model)))
if (direction == "forward") {
  while (length(predictors) > 0) {
    step_result <- step(init.model,
                        data, direction="forward",
                        trace)
    best_predictor <- step_result$best_pred

    if (best_predictor == "") {
      break
    }
    init.model <- suppressWarnings(
                        update(init.model,
                               as.formula(paste("~ . +",
                                          best_predictor))))
    predictors <- predictors[!predictors %in% best_predictor]
    anova <- rbind(anova,
                   data.frame(step=paste("+", best_predictor),
                              AICc=step_result$aicc))
  }
} else if (direction == "backward") {
  predictors <- names(init.model$coef)[-1]
  while (length(predictors) > 0) {
    step_result <- step(init.model, data,
                        direction="backward")
    worst_predictor <- step_result$worst_pred

    if (worst_predictor == "") {
      break
    }
    init.model <- suppressWarnings(
                        update(init.model,
                               as.formula(paste("~ . -",
                                          worst_predictor))))
    predictors <- predictors[!predictors %in% worst_predictor]
    anova <- rbind(anova,
                   data.frame(step=paste("-",
                                         worst_predictor),
                              AICc=step_result$aicc))
  }
} else  if (direction == "both") {
```

```
    while (TRUE) {
      step_forward <- step(init.model,
                           data, direction="forward",
                           trace)
      step_backward <- step(init.model,
                            data, direction="backward",
                            trace)
      if (step_forward$aicc < step_backward$aicc) {
        best_predictor <- step_forward$best_pred

        if (best_predictor == "") {
          break
        }
        init.model <- suppressWarnings(
                        update(init.model,
                               as.formula(paste("~ . +",
                                                best_predictor))))
        predictors <- predictors[!predictors %in% best_predictor]

        anova <- rbind(anova,
                    data.frame(step=paste("+",
                                          best_predictor),
                               AICc=step_forward$aicc))
      } else {
        worst_predictor <- step_backward$worst_pred

        if (worst_predictor == "") {
          break
        }
        init.model <- suppressWarnings(
                           update(init.model,
                                  as.formula(paste("~ . -",
                                                   worst_predictor))))
        predictors <- predictors[!predictors %in% worst_predictor]

        anova <- rbind(anova,
                       data.frame(
                           step=paste("-", worst_predictor),
                           AICc=step_backward$aicc))
      }
    }
  }

  return(list(model=init.model, anova=as.matrix(anova)))
}
```

Notably, the `'direction'` parameter can accept a `'both'` input.

3.6.1 Comparing stepAIC with stepAICc

There is no difference between the final predictors selected and those chosen by the original stepAIC process.

The following R code implements StepAICc using MLE regression.

```
# Forward stepwise selection based on AICc
null.model <- glm(y ~ 1,
                  data = train_data,
                  family = binomial(link = "logit"))

stepAICc.forward <- stepAICc(null.model,
                             train_data,
                             direction = "forward")
# Backward stepwise selection based on AICc
full.model <- suppressWarnings(glm(y ~.,
                     data=train_data,
                     family=binomial(link="logit")))

stepAICc.backward <- stepAICc(full.model,
                             train_data,
                             direction = "backward")

## Both forward and backward stepwise selection
correlations <- cor(data)[, 'y'][-1]
threshold <- 0.5
selected_predictors <- names(correlations[
                          abs(correlations)>threshold])
model_formula <- as.formula(paste("y ~",
                             paste(selected_predictors,
                             collapse = '+')))
# Fit the initial intermediate model
intermediate.model <- suppressWarnings(
                        glm(model_formula,
                            data=train_data,
                        family=binomial(link='logit')))
stepAICc.both <- stepAICc(intermediate.model,
                          train_data,
                          direction = "both")
### Let's view every step ###

stepAICc.forward$anova
stepAICc.backward$anova
stepAICc.both$anova
```

	step	AICc
1	Initial model (null)	604.32347
2	+ perimeter_worst	172.49026
3	+ smoothness_worst	114.77028
4	+ texture_worst	85.20568
5	+ concave.points_mean	77.50306
6	+ area_worst	73.35974
7	+ perimeter_mean	64.79537
8	+ symmetry_worst	63.90520
9	+ compactness_mean	61.90645
10	+ concave.points_se	59.35550

Table 3.14: `forward stepAICc`

	step	AICc
1	Initial model (full)	66.67925
2	- concave.points_worst	64.37647
3	- radius_mean	62.08451
4	- perimeter_worst	59.80328
5	- perimeter_se	57.53271
6	- smoothness_mean	55.27273
7	- fractal_dimension_mean	53.02326
8	- area_se	50.78422
9	- concavity_mean	48.55556
10	- area_mean	46.33718
11	- texture_mean	44.12903
12	- perimeter_mean	41.93104
13	- fractal_dimension_worst	39.74312
14	- fractal_dimension_se	37.56522
15	- compactness_se	35.39726
16	- smoothness_se	33.23919
17	- texture_se	31.09096

Table 3.15: `backward stepAICc`

	step	AICc
1	Initial model (intermediate)	100.31031
2	+ texture_worst	77.79680
3	- area_se	75.63919
4	- perimeter_mean	73.51922
5	- compactness_worst	71.45791
6	- radius_worst	69.44623
7	- perimeter_se	67.78257
8	- perimeter_worst	66.17131
9	+ symmetry_worst	64.55230
10	- concavity_mean	62.66473
11	- area_mean	60.68662
12	- concavity_worst	59.65936
13	+ smoothness_worst	59.27316

Table 3.16: `both stepAICc`

We observe that the results of `stepAICc` method we developed correspond well with those of `stepAIC`. However, it's important to remember that when dealing with smaller datasets, `stepAICc` can produce significantly different and possibly more accurate or preferred outcomes, in comparison to the original `stepAIC` method within the `MASS` package.

4

Penalized Estimation

In general, the goal of a regression model is to find the best-fitting function that minimizes the discrepancy between the predicted values and the actual values. In unpenalized regression, the focus is entirely on minimizing this error term. However, penalized estimation techniques introduce an additional term in the objective function to account for the complexity of the model. The main differences between penalized and unpenalized estimation methods are as follows:

- ◇ Regularization: Penalized estimation methods (ridge, lasso and elastic net) incorporate regularization techniques to avoid overfitting by adding a penalty term to the objective function. This penalty term constrains the magnitude of the coefficients, which helps in reducing model complexity and improving generalization to new data.
- ◇ Coefficient shrinkage: Penalized methods shrink the coefficients towards zero. Ridge regression uses L^2 regularization, which shrinks coefficients evenly, while lasso uses L^1 regularization, which can shrink some coefficients to exactly zero. Elastic net combines both L^1 and L^2 regularization, allowing for a balance between ridge and lasso.
- ◇ Feature selection: Lasso and elastic net can perform feature selection by setting some coefficients to zero, effectively eliminating the corresponding features from the model. This can result in simpler and more interpretable models. Ridge regression, on the other hand, does not set coefficients to zero, but it does shrink them, which can help in multicollinearity situations.
- ◇ Multicollinearity: Penalized methods are more robust to multicollinearity, a situation where predictor variables are highly correlated. Ridge regression is particularly helpful in this case, as it distributes the effect of correlated variables across the coefficients, reducing the impact of multicollinearity on the model.

◇ Bias-variance trade-off: Penalized methods introduce a bias in the estimates to reduce variance, improving the overall prediction accuracy. By controlling the amount of penalty, one can strike a balance between the bias and variance in the model, achieving better generalization.

4.1 Penalized Models (Lasso, Ridge, Elastic-Net)

In this section, we will look at penalized model estimation by using Lasso, Ridge and Elastic Net penalties. Also, we will apply the `10-fold` cross-validation technique to fine-tune the hyperparameters for each model. We will focus on penalized Linear Regression, Logistic Regression and Least Absolute Deviations.

4.1.1 Penalized Linear Regression via `glmnet`

• Linear Regression: family $\sim$ Gaussian: "`gaussian`" is the default family argument for the function glmnet. Suppose we have observations $x_i \in \mathbb{R}^p$ and the responses $y_i \in \mathbb{R}, i = 1, \ldots, N$. The objective function for the Gaussian family is:

$$\min_{(\beta_0,\beta)\in\mathbb{R}^{p+1}} \frac{1}{2N}\sum_{i=1}^{N}(y_i - \beta_0 - x_i^T\beta)^2 + \lambda\big[(1-a)\|\beta\|_2^2/2 + a\|\beta\|_1\big],$$

where $\lambda \geq 0$ is a hyperparameter and $0 \leq a \leq 1$ is a compromise between ridge regression ($\alpha = 0$) and lasso regression ($\alpha = 1$). glmnet applies coordinate descent[1] to solve the problem. The following `R` code implements penalized linear regression (Gaussian family) by fine-tuning the hyperparameter λ through 10-fold cross-validation. It also utilizes parallel computing.

```
set.seed(666)
registerDoMC(cores = 2)

### 10-fold cross-validation fit ###

# Gaussian

system.time(glm_gaussian_ridgeCV <- cv.glmnet(
                                          x=as.matrix(X_train),
                                          y=as.numeric(train_data$y),
                                          type.measure="mse",
                                          alpha=0,
                                          nfolds=10,
                                          parallel = TRUE))
```

[1]Coordinate descent is an optimization algorithm that successively minimizes along coordinate directions to find the minimum of a function

```
system.time(glm_gaussian_lassoCV <- cv.glmnet(
                                          x=as.matrix(X_train),
                                          y=as.numeric(train_data$y),
                                          type.measure="mse",
                                          alpha=1,
                                          nfolds=10,
                                          parallel = TRUE))
### VIF data ###

# Gaussian

system.time(glm_gaussian_ridgeCV_vif <- cv.glmnet(
                                          x=as.matrix(VIF_data[,-1]),
                                          y=as.numeric(train_data$y),
                                          type.measure="mse",
                                          alpha = 0,
                                          nfolds=10,
                                          parallel = TRUE))

system.time(glm_gaussian_lassoCV_vif <- cv.glmnet(
                                          x=as.matrix(VIF_data[,-1]),
                                          y=as.numeric(train_data$y),
                                          type.measure="mse",
                                          alpha = 1,
                                          nfolds = 10,
                                          parallel = TRUE))
```

4.1.2 Penalized Logistic Regression via `glmnet`

- Logistic Regression: family $\sim$ Binomial

The objective function for the Binomial family is:

$$\min_{(\beta_0,\beta)\in\mathbb{R}^{p+1}} -\left[\frac{1}{N}\sum_{i=1}^{N} y_i \cdot (\beta_0 + x_i^T\beta_0) - \log\left(1+e^{(\beta_0+x_i^T\beta)}\right)\right] + \lambda\left[(1-a)\|\beta\|_2^2/2 + a\|\beta\|_1\right]$$

The `R` code below, implements penalized logistic regression (Binomial family) by fine-tuning the hyperparameter λ.

```
set.seed(666)
registerDoMC(cores = 2)

### 10-fold cross-validation fit ###

# Binomial

system.time(glm_binomial_ridgeCV <- cv.glmnet(
                                          x=as.matrix(X_train),
```

```
                                        y=as.numeric(train_data$y),
                                        family='binomial',
                                        type.measure="auc",
                                        alpha=0,
                                        nfolds = 10,
                                        parallel = TRUE))

system.time(glm_binomial_lassoCV <- cv.glmnet(
                                        x=as.matrix(X_train),
                                        y=as.numeric(train_data$y),
                                        family='binomial',
                                        type.measure="auc",
                                        alpha=1,
                                        nfolds=10,
                                        parallel = TRUE))
### VIF data ###

system.time(glm_binomial_ridgeCV_vif <- cv.glmnet(
                                        x=as.matrix(VIF_data[,-1]),
                                        y=as.numeric(train_data$y),
                                        family='binomial',
                                        type.measure = "auc",
                                        alpha = 0,
                                        nfolds = 10,
                                        parallel = TRUE))

system.time(glm_binomial_lassoCV_vif <- cv.glmnet(
                                        x=as.matrix(VIF_data[,-1]),
                                        y=as.numeric(train_data$y),
                                        family='binomial',
                                        type.measure = "auc",
                                        alpha = 1,
                                        nfolds = 10,
                                        parallel = TRUE))
```

It's important to mention that generally, applying a Lasso or Elastic-Net regularization on a stepwise regression model isn't considered appropriate. These regularization methods are designed to perform their own variable selection, and this also applies to Ridge regularization. However, for the purpose of this analysis, we have fitted the stepwise model to observe its behavior. The code provided below refers to the complete second-order models based on the `AICc` and `BIC`.

```
## Second order data | based on "stepwise" AICc/BIC :
# Note: FULL second order models NOT the stepAIC one.
### Sec. ord. AICc ###
# Gaussian
system.time(glm_gaussian_ridgeCV_2oaicc <- cv.glmnet(
                                    x=as.matrix(sec_ord_aicc[,-1]),
```

```
                                        y=as.numeric(sec_ord_aicc$y),
                                        type.measure="mse",
                                        alpha = 0,
                                        nfolds=10,
                                        parallel = TRUE))
system.time(glm_gaussian_lassoCV_2oaicc <- cv.glmnet(
                                        x=as.matrix(sec_ord_aicc[,-1]),
                                        y=as.numeric(sec_ord_aicc$y),
                                        type.measure="mse",
                                        alpha = 1,
                                        nfolds = 10,
                                        parallel = TRUE))
# Binomial
system.time(glm_binomial_ridgeCV_2oaicc <- cv.glmnet(
                                        x=as.matrix(sec_ord_aicc[,-1]),
                                        y=as.numeric(sec_ord_aicc$y),
                                        family='binomial',
                                        type.measure = "auc",
                                        alpha = 0,
                                        nfolds = 10,
                                        parallel = TRUE))
system.time(glm_binomial_lassoCV_2oaicc <- cv.glmnet(
                                        x=as.matrix(sec_ord_aicc[,-1]),
                                        y=as.numeric(sec_ord_aicc$y),
                                        family='binomial',
                                        type.measure = "auc",
                                        alpha = 1,
                                        nfolds = 10,
                                        parallel = TRUE))
### Sec. ord. BIC ###
# Gaussian
system.time(glm_gaussian_ridgeCV_2obic <- cv.glmnet(
                                        x=as.matrix(sec_ord_bic[,-1]),
                                        y=as.numeric(sec_ord_bic$y),
                                        type.measure="mse",
                                        alpha = 0,
                                        nfolds=10,
                                        parallel = TRUE))
system.time(glm_gaussian_lassoCV_2obic <- cv.glmnet(
                                       x=as.matrix(sec_ord_bic[,-1]),
                                       y=as.numeric(sec_ord_bic$y),
                                       type.measure="mse",
                                       alpha = 1,
                                       nfolds = 10,
                                       parallel = TRUE))
# Binomial
system.time(glm_binomial_ridgeCV_2obic <- cv.glmnet(
                                        x=as.matrix(sec_ord_bic[,-1]),
                                        y=as.numeric(sec_ord_bic$y),
```

```
                                        family='binomial',
                                        type.measure = "auc",
                                        alpha = 0,
                                        nfolds = 10,
                                        parallel = TRUE))
system.time(glm_binomial_lassoCV_2obic <- cv.glmnet(
                                       x=as.matrix(sec_ord_bic[,-1]),
                                       y=as.numeric(sec_ord_bic$y),
                                       family='binomial',
                                       type.measure = "auc",
                                       alpha = 1,
                                       nfolds = 10,
                                       parallel = TRUE))
```

The models have been successfully fitted. Now, let's view the plot displaying the relationship between the lambda values (in logarithmic scale) and their corresponding performance metrics:

```
par(mfrow=c(3,4), cex.main = 0.87)

plot(glm_gaussian_ridgeCV_2oaicc)
title("Gaussian-Ridge (sec.ord AICc)",adj=0.5,line=2.3)
plot(glm_binomial_ridgeCV_2oaicc)
title("Binomial-Ridge (sec.ord AICc)",adj=0.5,line=2.3)
plot(glm_gaussian_ridgeCV_2obic)
title("Gaussian-Ridge (sec.ord BIC)",adj=0.5,line=2.3)
plot(glm_binomial_ridgeCV_2obic)
title("Binomial-Ridge (sec.ord BIC)",adj = 0.5,line=2.3)
plot(glm_gaussian_lassoCV_2oaicc)
title("Gaussian-Lasso (sec.ord AICc)",adj=0.5,line=2.3)
plot(glm_binomial_lassoCV_2oaicc)
title("Binomial-Lasso (sec.ord AICc)",adj=0.5,line=2.3)
plot(glm_gaussian_lassoCV_2obic)
title("Gaussian-Lasso (sec.ord BIC)",adj=0.5,line=2.3)
plot(glm_binomial_lassoCV_2obic)
title("Binomial-Lasso (sec.ord BIC)",adj=0.5,line=2.3)
plot(glm_binomial_ridgeCV)
title(main = "Binomial-Ridge (Full)",line=2.3)
plot(glm_binomial_lassoCV)
title(main = "Binomial-Lasso (Full)",line=2.3)
plot(glm_binomial_ridgeCV_vif)
title(main = "Binomial glm ridge (VIF)",line=2.3)
plot(glm_binomial_lassoCV_vif)
title(main = "Binomial glm lasso (VIF)",line=2.3)
```

Let's now fit the `stepAIC` models, 3 in total (best based on: `AIC`, `BIC` and best second-order).

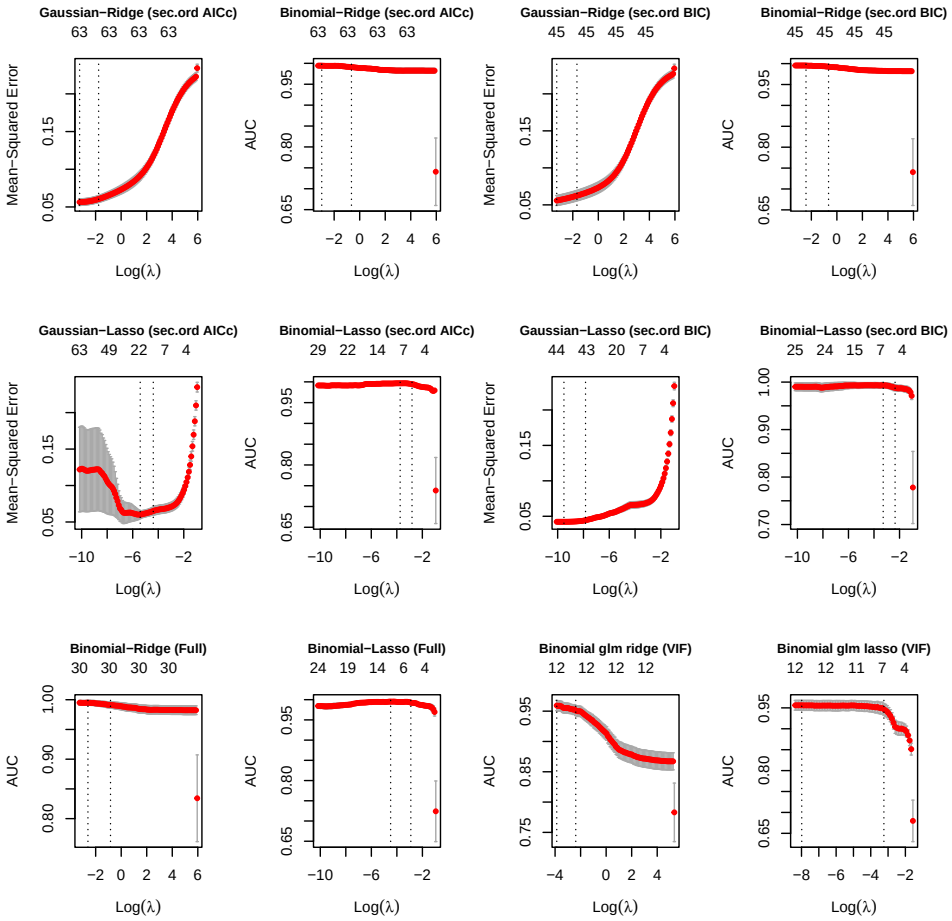


Figure 4.1: `10-Fold CV glmnet` metric-Log(λ) plots

```
#Second order data | based on "stepwise" AICc/BIC:

### Best (stepwise) Sec. ord. ###
# Gaussian

y_2o <- as.numeric(best_sec_order.model[,1])

system.time(glm_gaussian_ridgeCV_best2o <- cv.glmnet(
                                          x=best_sec_order.model[,-1],
                                            y=y_2o,
                                            type.measure="mse",
                                            alpha = 0,
                                            nfolds=10,
```

```
                                            parallel = TRUE))


system.time(glm_gaussian_lassoCV_best2o <- cv.glmnet(
                                            x=best_sec_order.model[,-1],
                                              y=y_2o,
                                              type.measure="mse",
                                              alpha = 1,
                                              nfolds = 10,
                                              parallel = TRUE))


# Binomial
system.time(glm_binomial_ridgeCV_best2o <- cv.glmnet(
                                            x=best_sec_order.model[,-1],
                                              y=y_2o,
                                              family='binomial',
                                              type.measure = "auc",
                                              alpha = 0,
                                              nfolds = 10,
                                              parallel = TRUE))


system.time(glm_binomial_lassoCV_best2o <- cv.glmnet(
                                            x=best_sec_order.model[,-1],
                                              y=y_2o,
                                              family='binomial',
                                              type.measure = "auc",
                                              alpha = 1,
                                              nfolds = 10,
                                              parallel = TRUE))
```

The provided R code generates a plot (Figure 4.2) with the logarithm of lambda (λ) values on the x-axis and the y-axis represents the Mean Squared Error (MSE) for the Gaussian family and Area Under Curve (ROC AUC) for the Binomial family.

```
par(mfrow=c(2,2), cex.main = 0.95)
plot(glm_gaussian_ridgeCV_best2o)
title("Gaussian-Ridge (Best Sec. Order)", adj = 0.5, line = 2.3)
plot(glm_binomial_ridgeCV_best2o)
title("Binomial-Ridge (Best Sec. Order)", adj = 0.5, line = 2.3)
plot(glm_gaussian_lassoCV_best2o)
title("Gaussian-Lasso (Best Sec. Order)", adj = 0.5, line = 2.3)
plot(glm_binomial_lassoCV_best2o)
title("Binomial-Lasso (Best Sec. Order)", adj = 0.5, line = 2.3)
```

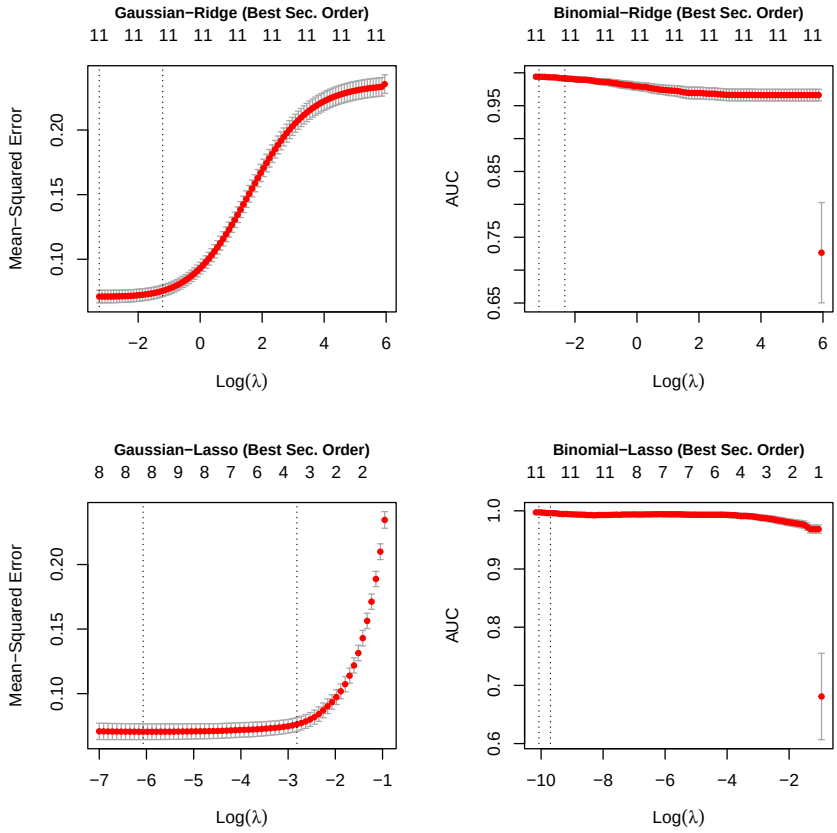
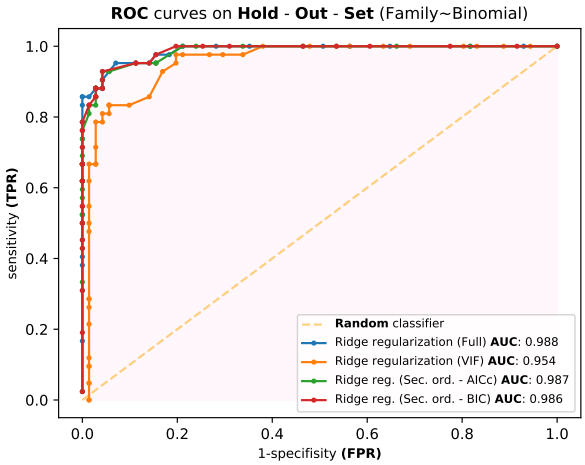


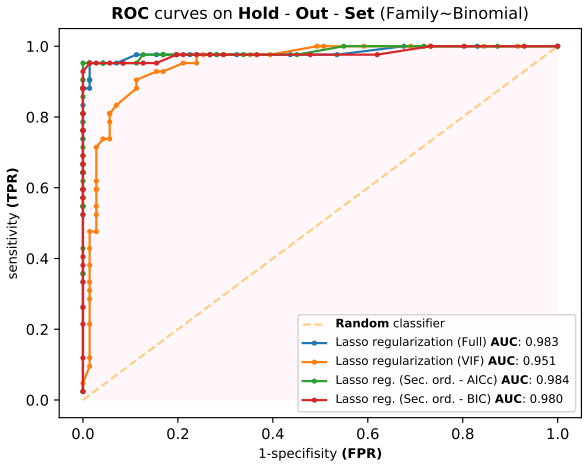
Figure 4.2: `10-Fold CV second order glmnet` metric-Log(λ) plots

4.1.3 `cv.glmnet` ROC Perfomance

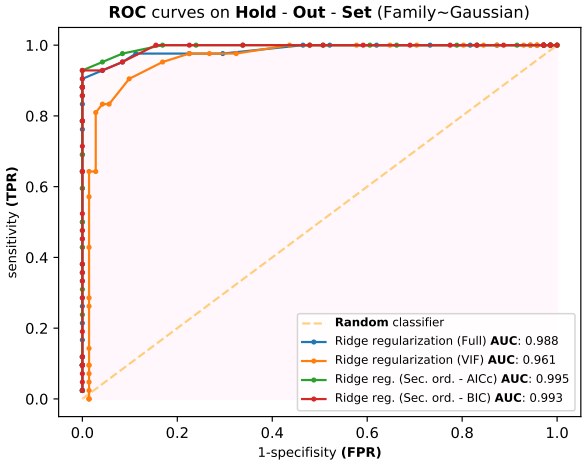
The following figures (Figure 4.3, Figure 4.4) visualize the ROC curves on the out-of-sample data for each trained model presented below (as legends).



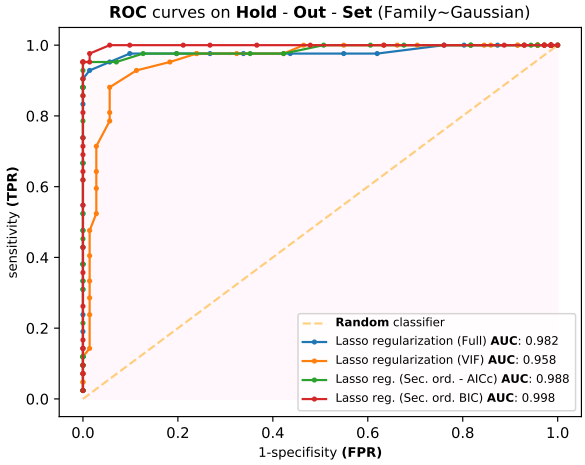
(a) `Ridge-Logistic-Regression`



(b) `Lasso-Logistic-Regression`



(c) `Ridge-Linear-Regression`

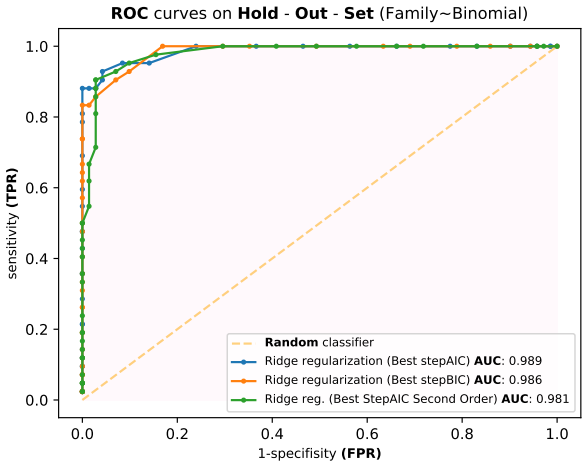


(d) `Lasso-Linear-Regression`

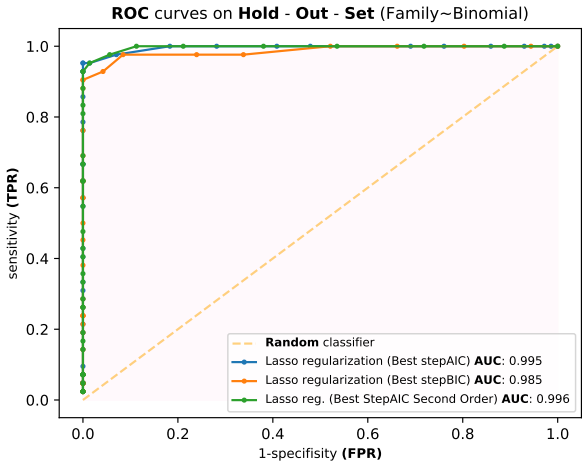
Figure 4.3: `10-Fold CV glmnet` models `ROC` Curves

4.1.4 Penalized Least Absolute Deviations

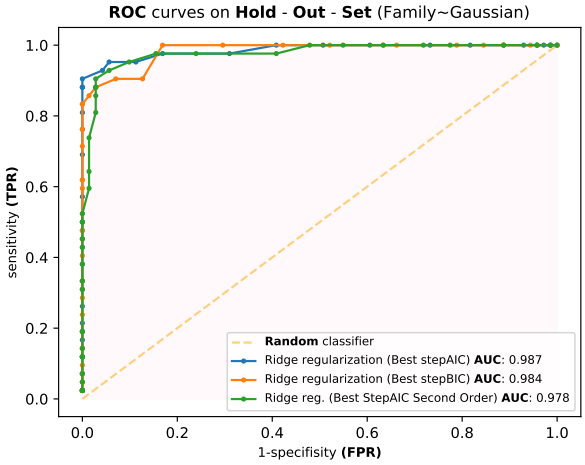
The `'hqreg'` package in `R` includes tools for Lasso, Ridge and Elastic-Net penalized regression models. While it supports both Quantile and Huber loss regression, in our discussion, we will focus more on Quantile regression (LAD loss, shown in Figure 4.5). However, we'll also take a look at how Huber loss regression performs. One important feature of this package is the `cv.hqreg` method. This method uses K-fold cross-validation to adjust the HP λ.



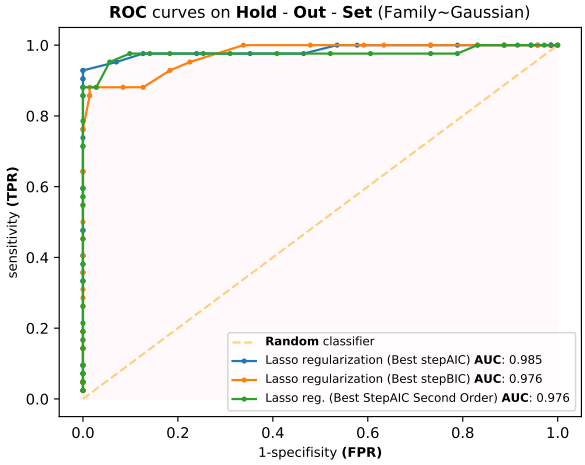
(a) `Ridge-Logistic-Regression`



(b) `Lasso-Logistic-Regression`



(c) `Ridge-Linear-Regression`



(d) `Lasso-Linear-Regression`

Figure 4.4: `10-Fold CV glmnet StepAIC` models `ROC` curves

When using the Huber loss and the elastic-net penalty, we obtain the elastic-net penalized Huber regression:

$$\min_{\beta} f_H(\beta) = \frac{1}{N}\sum_{i}^{N} h_\gamma\big(y_i - \beta_0 - \mathbf{x_i}^T\beta\big) + \lambda\big(a\|\beta\| + (1-a)^{1/2}\|\beta\|_2^2\big),\ 0 \le y \le 1$$

$\diamond$ $h_\delta(u)$ is the Huber loss function, which is defined as:

$$h_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| \leq \delta \\ \delta(|x| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

$\diamond$ δ (delta) is a tuning parameter that controls the transition between squared loss and absolute loss. Figure 4.5 illustrates various types of loss functions. In our analysis, we focus on the LAD (or Quantile loss) and also explore the Huber loss.

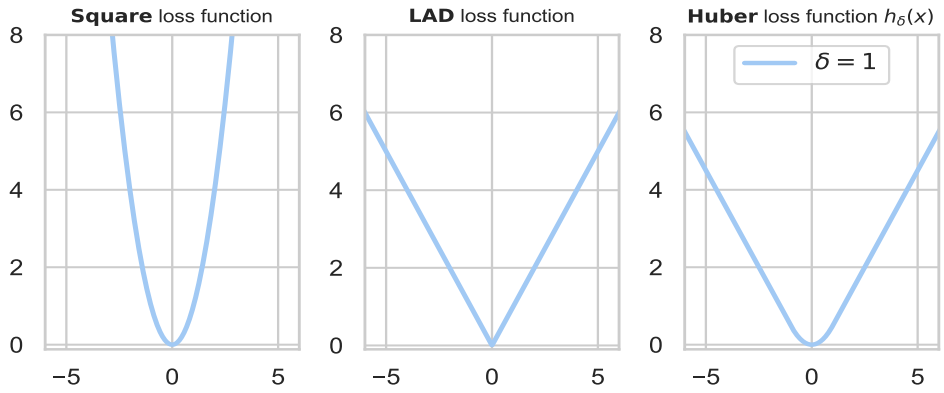


Figure 4.5: Loss functions

The Elastic-Net penalized Quantile Regression optimization problem can be formulated as:

$$\min_{\boldsymbol{\beta}} \; Q(\tau \,|\, \mathbf{X}, \mathbf{y}, \boldsymbol{\beta}) + \lambda \cdot \left(\alpha \cdot \|\boldsymbol{\beta}\|_1 + (1 - \alpha) \cdot \frac{\|\boldsymbol{\beta}\|_2^2}{2} \right),$$

where:

- $Q(\tau \,|\, \mathbf{X}, \mathbf{y}, \boldsymbol{\beta})$ is the quantile loss function for a given quantile τ $(0 < \tau < 1)$ and it is defined as:

$$Q(\tau \,|\, \mathbf{X}, \mathbf{y}, \boldsymbol{\beta}) = \frac{1}{n} \sum_{i=1}^{n} \left[\tau \cdot \max(y_i - \mathbf{x}_i^T \boldsymbol{\beta}, 0) + (1 - \tau) \cdot \max(\mathbf{x}_i^T \boldsymbol{\beta} - y_i, 0) \right]$$

- $\lambda \geq 0$ is the regularization parameter, which controls the strength of the penalty term. A higher λ value leads to more regularization and sparser coefficient estimates.

- α is the mixing parameter $(0 \leq \alpha \leq 1)$ that balances the contributions of `L1` (`Lasso`) and `L2` (`Ridge`) penalties. When $\alpha = 0$, the penalty term is equivalent to Ridge Regression and when $\alpha = 1$, the penalty term is equivalent to Lasso Regression.

The solution can be found using various optimization algorithms, such as `gradient descent` or `coordinate gradient descent`. The code below utilizes the `'cv.hqreg'` function for penalized Huber loss and Quantile regression. The method also performs hyperparameter tuning (λ) by employing 10-fold cross-validation.

```
library(hqreg)
registerDoMC(cores = 2)

### We are gonna fit 5 models ###:
#1.(full), 2.(VIF), 3.(bestStepAIC)
#4.(bestStepBIC), 5.(best 2ord stepmodel)
### Ridge , Lasso penalty
### Huber, Quantile Loss

### Total cv.hqreg models: 5x2x2=20.

cv_huberreg <- function (X, y=y_train, alpha) {
    return (
            cv.hqreg(
                X=as.matrix(X),
                y=as.numeric(y),
                nfolds = 10,
                alpha=alpha,
                method='huber',
                seed=5666))
}

cv_quantilereg <- function (X, y=y_train, alpha) {
    return (
            cv.hqreg(
                X=as.matrix(X),
                y=as.numeric(y),
                nfolds = 10,
                alpha=alpha,
                method='quantile',
                seed=5666))
}


system.time({
  ##### Huber loss #####
  #Ridge, lasso
  hcvLAD_ridge.full <- cv_huberreg(X_train, alpha=0)
  hcvLAD_lasso.full <- cv_huberreg(X_train, alpha=1)
  hcvLAD_ridge.vif <- cv_huberreg(VIF_data[,-1], alpha=0)
  hcvLAD_lasso.vif <- cv_huberreg(VIF_data[,-1], alpha=1)
  hcvLAD_ridge.stepAIC <- cv_huberreg(best_stepAIC.data[-1],
                                      alpha=0)
  hcvLAD_lasso.stepAIC <- cv_huberreg(best_stepAIC.data[-1],
```

```
                                    alpha=1)
  hcvLAD_ridge.stepBIC <- cv_huberreg(best_stepBIC.data[-1],
                                    alpha=0)
  hcvLAD_lasso.stepBIC <- cv_huberreg(best_stepBIC.data[-1],
                                    alpha=1)
  hcvLAD_ridge.step2o <- cv_huberreg(best_sec_order.model[,-1],
                                   alpha=0)
  hcvLAD_lasso.step2o <- cv_huberreg(best_sec_order.model[,-1],
                                   alpha=1)

  ##### quantile loss #####
  #Ridge, lasso
  qcvLAD_ridge.full <- cv_quantilereg(X_train, alpha=0)
  qcvLAD_lasso.full <- cv_quantilereg(X_train, alpha=1)
  qcvLAD_ridge.vif <- cv_quantilereg(VIF_data[,-1], alpha=0)
  qcvLAD_lasso.vif <- cv_quantilereg(VIF_data[,-1], alpha=1)
  qcvLAD_ridge.stepAIC <- cv_quantilereg(best_stepAIC.data[-1],
                                      alpha=0)
  qcvLAD_lasso.stepAIC <- cv_quantilereg(best_stepAIC.data[-1],
                                     alpha=1)
  qcvLAD_ridge.stepBIC <- cv_quantilereg(best_stepBIC.data[-1],
                                      alpha=0)
  qcvLAD_lasso.stepBIC <- cv_quantilereg(best_stepBIC.data[-1],
                                      alpha=1)
  qcvLAD_ridge.step2o <- cv_quantilereg(best_sec_order.model[,-1],
                                     alpha=0)
  qcvLAD_lasso.step2o <- cv_quantilereg(best_sec_order.model[,-1],
                                     alpha=1)
})
```

The 20 models have been fitted, with the optimal lambda selected for each penalty term and loss function through cross-validation. Figure 4.6 displays the results of cross-validation. The x-axis represents various values of the tuning parameter, lambda (λ), while the y-axis shows the corresponding Deviance, which is a measure of error or difference between the predicted and actual values. This plot helps in understanding how the model's deviance changes with different lambda values.

```
par(mfrow=c(4,5), cex.main = 0.75)

################
## Huber loss ##
################

plot(hcvLAD_ridge.full)
title(main = "huber-ridge (Full)",line=1.6)
plot(hcvLAD_lasso.full)
title(main = "huber-lasso (Full)",line=1.6)
plot(hcvLAD_ridge.vif)
```

```
title(main = "huber-ridge (vif)",line=1.6)
plot(hcvLAD_lasso.vif)
title(main = "huber-lasso (vif)",line=1.6)
plot(hcvLAD_ridge.stepAIC)
title(main = "huber-ridge (stepAIC)",line=1.6)
plot(hcvLAD_lasso.stepAIC)
title(main = "huber-lasso (stepAIC)",line=1.6)
plot(hcvLAD_ridge.stepBIC)
title(main = "huber-ridge (stepBIC)",line=1.6)
plot(hcvLAD_lasso.stepBIC)
title(main = "huber-lasso (stepBIC)",line=1.6)
plot(hcvLAD_ridge.step2o)
title(main = "huber-ridge (step sec. ord.)",line=1.6)
plot(hcvLAD_lasso.step2o)
title(main = "huber-lasso (step sec. ord.)",line=1.6)

#########################
## Quantile loss (LAD) ##
#########################

plot(qcvLAD_ridge.full)
title(main = "huber-ridge (Full)",line=1.6)
plot(qcvLAD_lasso.full)
title(main = "huber-lasso (Full)",line=1.6)
plot(qcvLAD_ridge.vif)
title(main = "huber-ridge (vif)",line=1.6)
plot(qcvLAD_lasso.vif)
title(main = "huber-lasso (vif)",line=1.6)
plot(qcvLAD_ridge.stepAIC)
title(main = "huber-ridge (stepAIC)",line=1.6)
plot(qcvLAD_lasso.stepAIC)
title(main = "huber-lasso (stepAIC)",line=1.6)
plot(qcvLAD_ridge.stepBIC)
title(main = "huber-ridge (stepBIC)",line=1.6)
plot(qcvLAD_lasso.stepBIC)
title(main = "huber-lasso (stepBIC)",line=1.6)
plot(qcvLAD_ridge.step2o)
title(main = "huber-ridge (step sec. ord.)",line=1.6)
plot(qcvLAD_lasso.step2o)
title(main = "huber-lasso (step sec. ord.)",line=1.6)
```

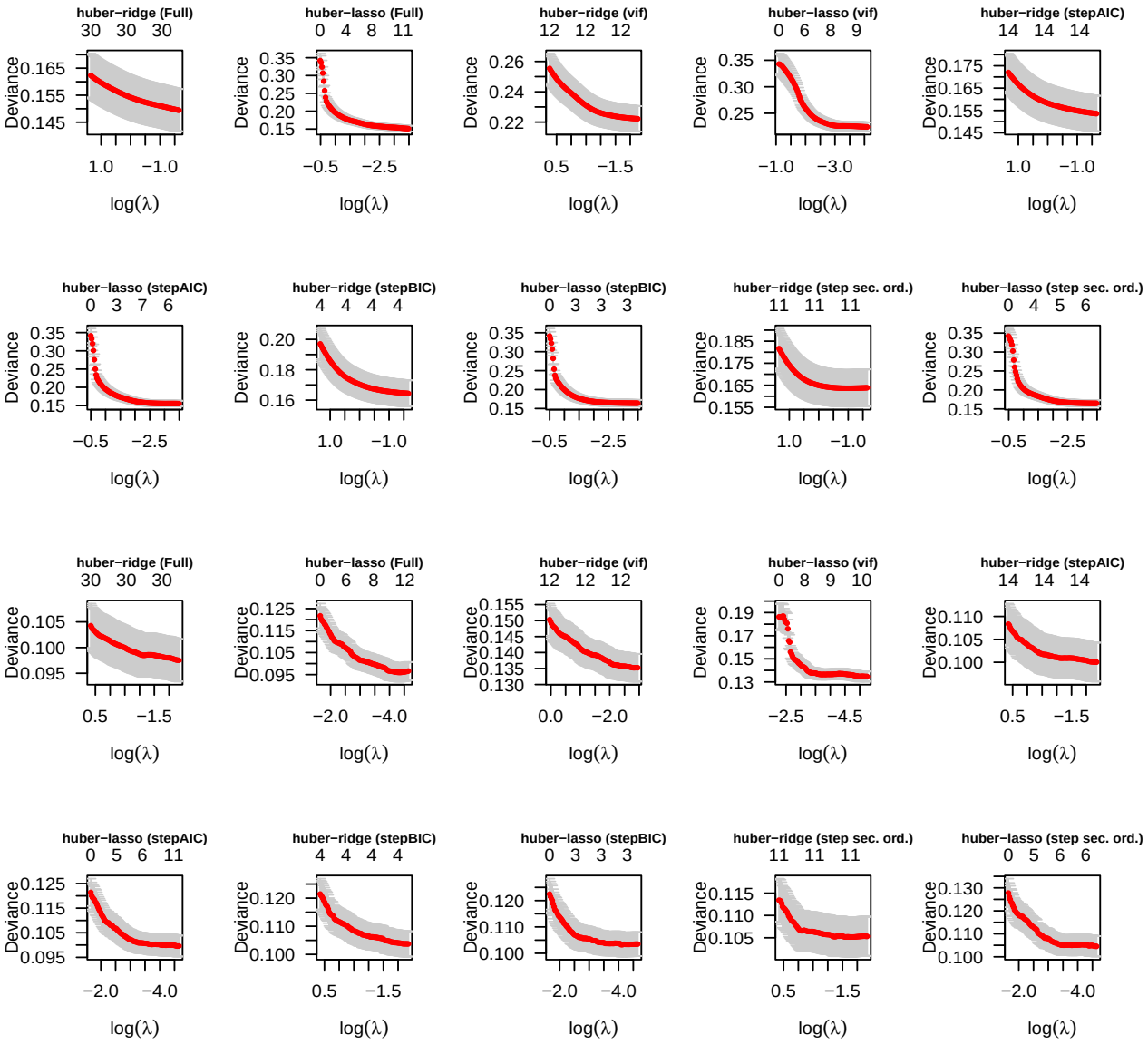
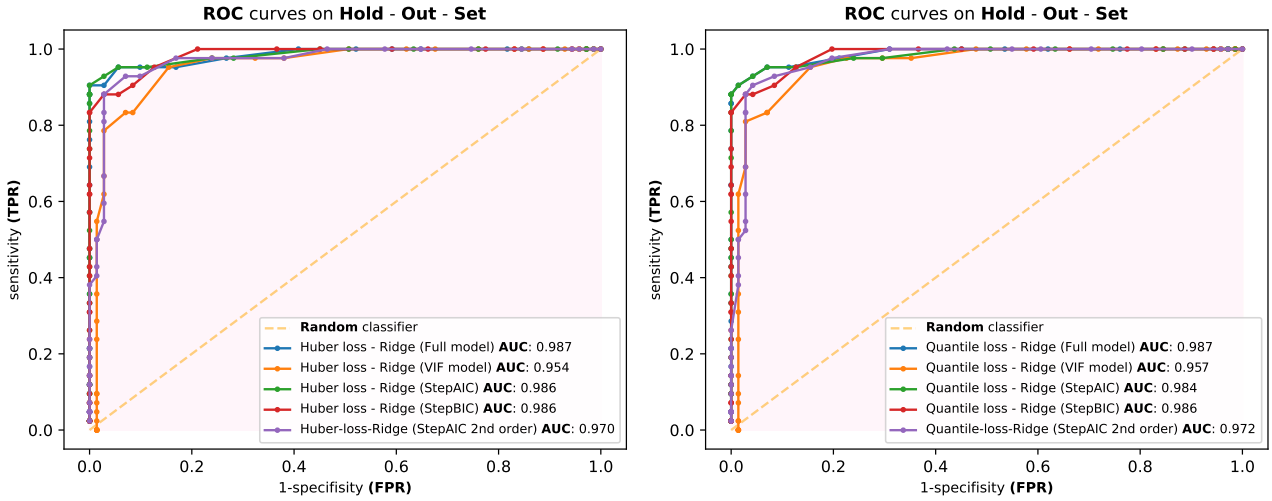


Figure 4.6: `10-Fold CV hqreg` metric-Log(λ) plots

4.1.5 `cv.hqreg` ROC Perfomance

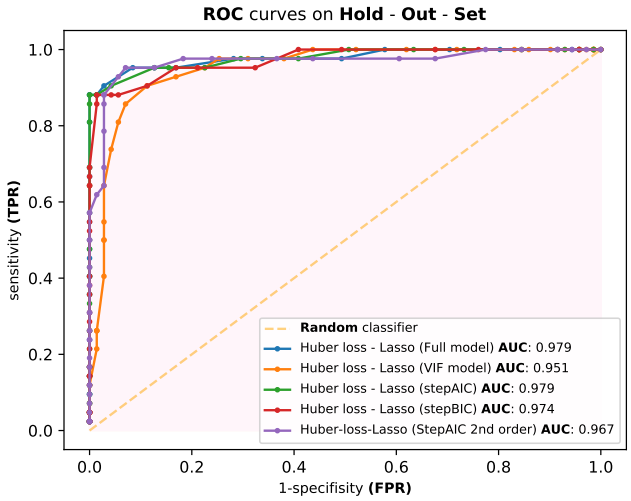
The Figure 4.7 present visualization of the ROC curves on the out-of-sample data for each trained model (shown as legends). This visualization allows for the

comparison of model performance under two different loss functions: Quantile loss (LAD) and Huber loss.

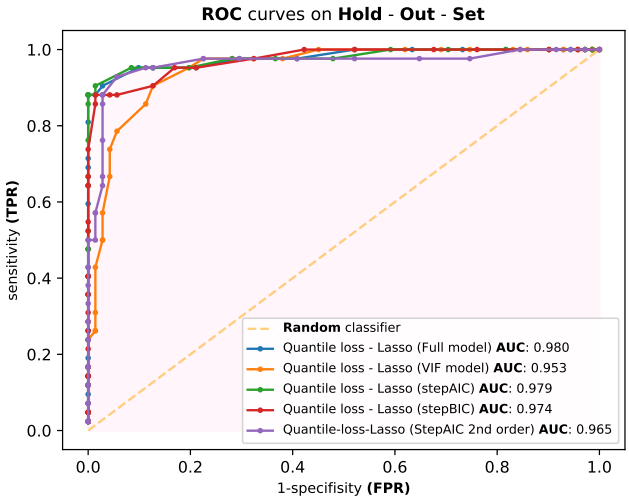


(a) `Ridge-huber-loss`

(b) `Ridge-quantile-loss`



(c) `Lasso-huber-loss`



(d) `Lasso-quantile-loss`

Figure 4.7: `10-Fold CV hqreg` models ROC Curves

4.2 Pathwise Coordinate Descent Optimization

Introduced in 2007 by Friedman, Hastie, Höfling and Tibshirani, the `pathwise coordinate optimization` method [15], provides an effective way to solve problems commonly encountered in regularized statistical models such as Lasso and Elastic Net.

The core principle of this technique involves tracing the evolution of parameter estimates as a function of the regularization parameter. Rather than solving the optimization problem anew for each value of the regularization parameter, this technique smartly uses the solution at one point as a stepping stone for the next, saving computational time.

The unique aspect of this method is its coordinate-wise operation. It improves the objective function one feature at a time, keeping all other features constant and continues to loop through all the features until the adjustment in the solution becomes negligible. This characteristic makes it an especially powerful tool when dealing with high-dimensional data.

One of the key outcomes of this optimization process is a piecewise-linear solution path. By conservatively storing and reusing past solutions, this algorithm can drastically cut down computation time. It swiftly provides a range of solutions over a series of regularization parameter values, which proves extremely helpful when the optimal regularization parameter isn't predetermined and must be established through techniques like cross-validation.

The optimization problem for Lasso is:

$$\min_{\beta} \frac{1}{2n} \sum_{i=1}^{n} (y_i - x_i^T \beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

where

- y_i is the response variable.
- x_i is the $i\text{-}th$ observation vector.
- β is the vector of coefficients we want to estimate.
- λ is the regularization parameter.
- n is the number of observations.
- p is the number of features.

To solve this with pathwise coordinate optimization, we would iteratively optimize one β_j at a time, holding the other coefficients constant, until convergence. The updating rule for the $j\text{-}th$ coordinate, in the case of Lasso, is:

$$\beta_j \leftarrow S\left(\frac{1}{n}x_j^T(y - X_{-j}\beta_{-j}), \lambda\right),$$

where X_{-j} denotes the matrix X with the $j\text{-}th$ column removed, β_{-j} is the coefficient vector without the $j - th$ coefficient and S is the soft-thresholding operator, defined as:

$$S(z, \gamma) = sign(z)(|z| - \gamma)_+ \ ,$$

where $(a)_+ = \max(a, 0)$.

This algorithm continues until the changes in the coefficients become small enough, indicating that the solution has converged. The method is efficient and produces a path of solutions as λ varies, which is helpful in scenarios where we need to tune the regularization parameter.

The objective function for the Binomial family is:

$$\min_{(\beta_0,\beta)\in\mathbb{R}^{p+1}} -\left[\frac{1}{N}\sum_{i=1}^{N} y_i \cdot (\beta_0 + x_i^T\beta_0) - \log\left(1 + e^{(\beta_0 + x_i^T\beta)}\right)\right] + \lambda\big[(1-a)\|\beta\|_2^2/2 + a\|\beta\|_1\big]$$

The objective function for the Gaussian family is:

$$\min_{(\beta_0,\beta)\in\mathbb{R}^{p+1}} \frac{1}{2N}\sum_{i=1}^{N}(y_i - \beta_0 - x_i^T\beta)^2 + \lambda\big[(1-a)\|\beta\|_2^2/2 + a\|\beta\|_1\big].$$

Coordinate descent is applied to solve the problem. Specifically, suppose we have current estimates $\tilde{\beta}_0$ and $\tilde{\beta}_\ell \ \forall \ j \in 1, \ldots, p$. By computing the gradient at $\beta_j = \tilde{\beta}_j$ and simple calculus, the update is:

$$\tilde{\beta}_j \leftarrow \frac{S\left(\frac{1}{N}\sum_{i=1}^{N} x_{ij}\big(y_i - \tilde{y}_i^{(j)}\big), \lambda a\right)}{1 + \lambda(1-a)},$$

where $\tilde{y}_i^{(j)} = \tilde{\beta}_0 + \sum_{\ell\neq j} x_{i\ell}\tilde{\beta}_\ell$ and $S(z, \gamma)$ is the soft-thresholding operator.

Logistic regression (Binomial family) can encounter instability and erratic results when the number of predictors (p) exceeds the number of observations (N), or even when they are nearly equal. The elastic-net penalty serves as a solution to these challenges, enhancing stability and conducting variable selection and regularization effectively.

5

Cross-Validation & Bootstrapping

5.1 K-Fold Cross-Validation

K-fold cross-validation is a resampling technique used in machine learning and statistical modeling to evaluate the performance of a predictive model. It involves dividing the dataset into K equal-sized subsamples, or 'folds', where K is a positive integer. The model is then trained and evaluated K times, with each of the K folds used exactly once as the validation data, while the remaining $K-1$ folds are used as the training data.

In each fold, the model is trained on the training data and evaluated on the validation data. The performance metric of interest, such as accuracy or mean squared error, is recorded for each fold. The K performance metrics are then averaged to obtain a single overall estimate of the model's performance. This estimate can be used to compare the performance of different models, or to tune the hyperparameters of the model.

K-fold cross-validation is a popular method for estimating the performance of a model because it can help to reduce the impact of sampling variability and overfitting. It can also be used to identify instances of bias or variance in the model, as well as to detect potential problems with the data, such as outliers or class imbalance. How many models are trained by the K-Fold CV?

$$K \times C + 1,$$

C: number of configurations and the "plus one" is the final model that is trained using all data.

The figure provided below presents the procedure of cross-validation for $K = 5$.

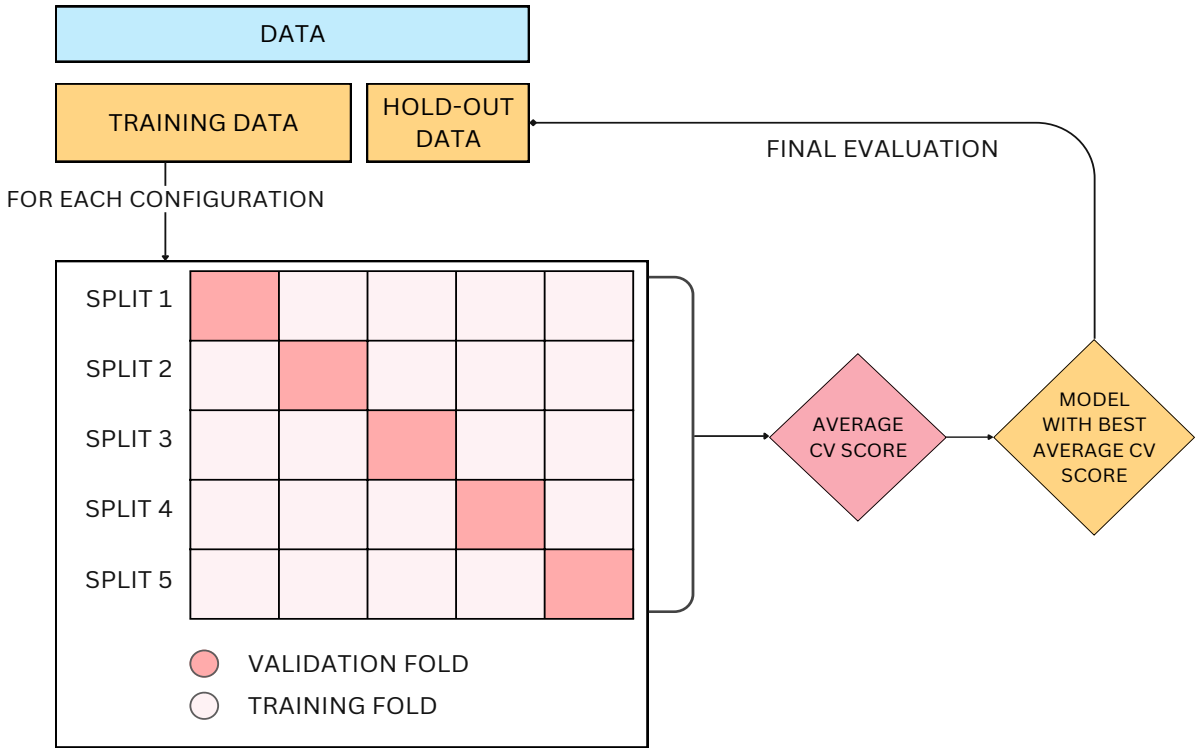


Figure 5.1: K-Fold Cross-Validation procedure (K=5)

Keep in mind that each split should be stratified (Figure 5.2). Stratification is important to maintain the balance of class proportions in the subsets. Stratified sampling ensures that each fold has a similar proportion of each class as the entire dataset.

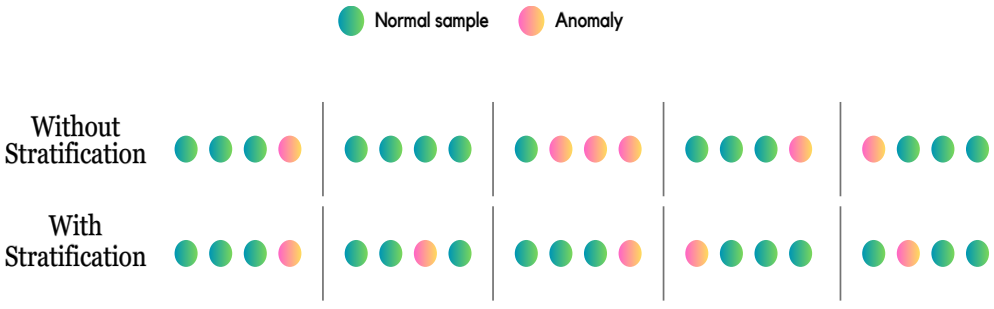


Figure 5.2: Stratified vs non-stratified data

5.2 Nested Cross-Validation

Nested cross-validation is a technique used in machine learning to evaluate and optimize the performance of a model by using multiple rounds of cross-validation.

In nested cross-validation, there are two levels of cross-validation loops. The outer loop is used to evaluate the performance of the model using K-fold cross-validation, where the data is divided into K equal parts and the model is trained on $K-1$ parts and tested on the remaining part. This process is repeated K times and the performance of the model is averaged across the K folds.

The inner loop is used to select the best hyperparameters for the model. The data is further divided into multiple folds and the model is trained and evaluated on these folds using different hyperparameters. This process is repeated for each fold of the outer loop and the best hyperparameters are selected based on the average performance across all the folds.

The purpose of using nested cross-validation is to prevent overfitting and obtain a more reliable estimate of the model's performance on new, unseen data.

Nested CV is unbiased because even though we pick a configuration from the inner CV, we then evaluate this on new test data on the outer loop.

How many models are trained by the nested CV?

$$\underbrace{K}_{\text{outer loop}} \times \Big(\underbrace{(K-1) \times C + 1}_{\text{inner loop}} \Big).$$

The figure below illustrates the procedure of nested cross-validation for $K = 3$.

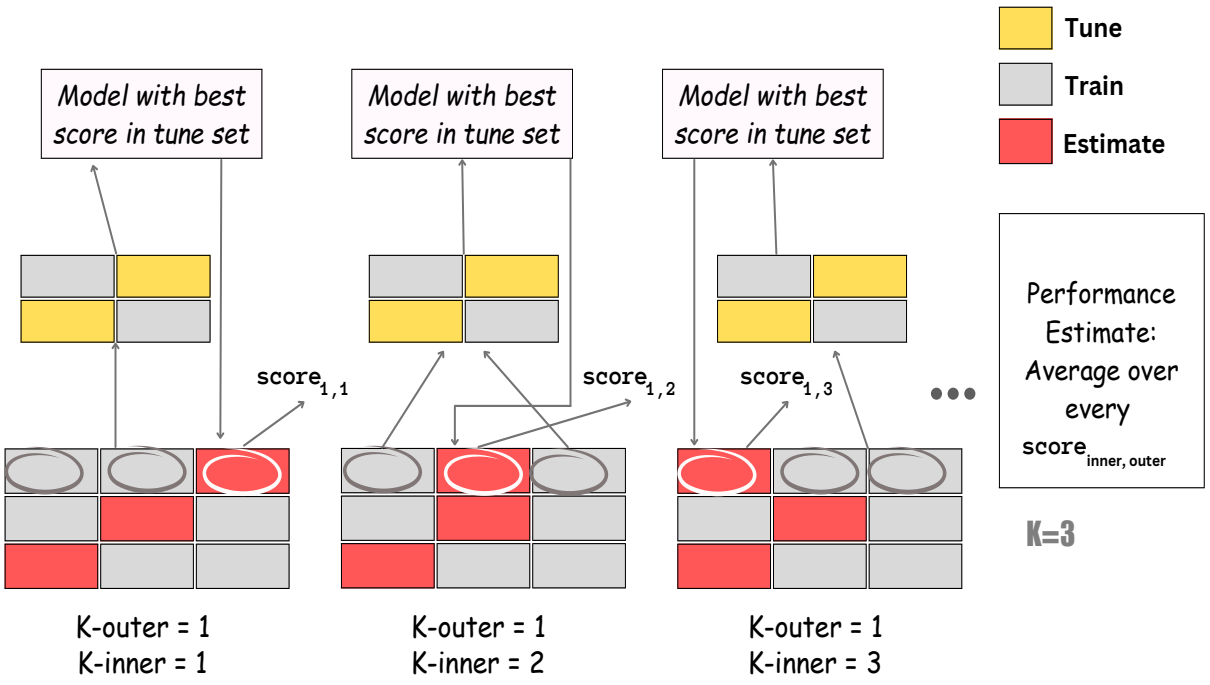


Figure 5.3: Nested cross-validation procedure (K=3)

5.2.1 Nested Cross-Validation for `glmnet` Models

`nestedcv.glmnet` is a function provided by the `nestedcv` package in `R`, designed specifically to perform nested cross-validation for models created using the `glmnet` package. The `glmnet` package is a popular `R` package for fitting generalized linear models (`GLMs`) with regularization (`LASSO, Ridge, or Elastic Net`).

The `nestedcv.glmnet` function simplifies the process of performing nested cross-validation for `glmnet` models and helps in estimating the model's performance more accurately while also optimizing the hyperparameters.

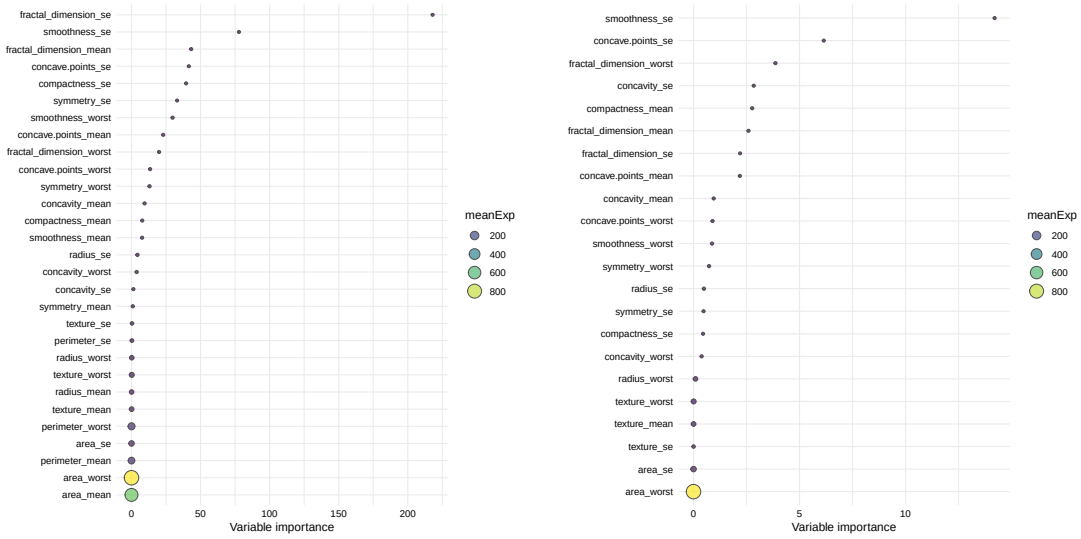
```
library(nestedcv)

X <- data[-1]

data_2o <- cbind(interactions_xixj(X),
     data[-1])[, names(as.data.frame(
     best_sec_order.model))[-1]]

nestedCV_binomial <- function (X, y=data$y) {
```

```
        return(nestcv.glmnet(
                        y=as.factor(y),
                        x=as.matrix(X),
                        min_1se=0,
                        family="binomial",
                        cv.cores=10,
                        alphaSet=seq(0, 1, 0.05),
                        parallel = TRUE))
}
nestedCV_gaussian <- function (X, y=data$y) {
        return(nestcv.glmnet(
                        y=as.numeric(y),
                        x=as.matrix(X),
                        min_1se=0,
                        family="gaussian",
                        cv.cores=10,
                        alphaSet=seq(0, 1, 0.05),
                        parallel = TRUE))
}

set.seed(69)
registerDoMC(cores = 2)

system.time({
    ## Binomial ##
    ncv_bin.full <- nestedCV_binomial(X)
    ncv_bin.vif <- nestedCV_binomial(X[,
                        names(VIF_data[,-1])])
    ncv_bin.stepAIC <- nestedCV_binomial(X[,
                        names(best_stepAIC.data)[-1]])
    ncv_bin.stepBIC <- nestedCV_binomial(X[,
                        names(best_stepBIC.data)[-1]])
    ncv_bin.step2o <- nestedCV_binomial(data_2o)

    ## Gaussian ##
    ncv_gauss.full <- nestedCV_gaussian(X)
    ncv_gauss.vif <- nestedCV_gaussian(X[,
                        names(VIF_data[,-1])])
    ncv_gauss.stepAIC <- nestedCV_gaussian(X[,
                        names(best_stepAIC.data)[-1]])
    ncv_gauss.stepBIC <- nestedCV_gaussian(X[,
                        names(best_stepBIC.data)[-1]])
    ncv_gauss.step2o <- nestedCV_gaussian(data_2o)
})
```

5.2.2 Variable Importance

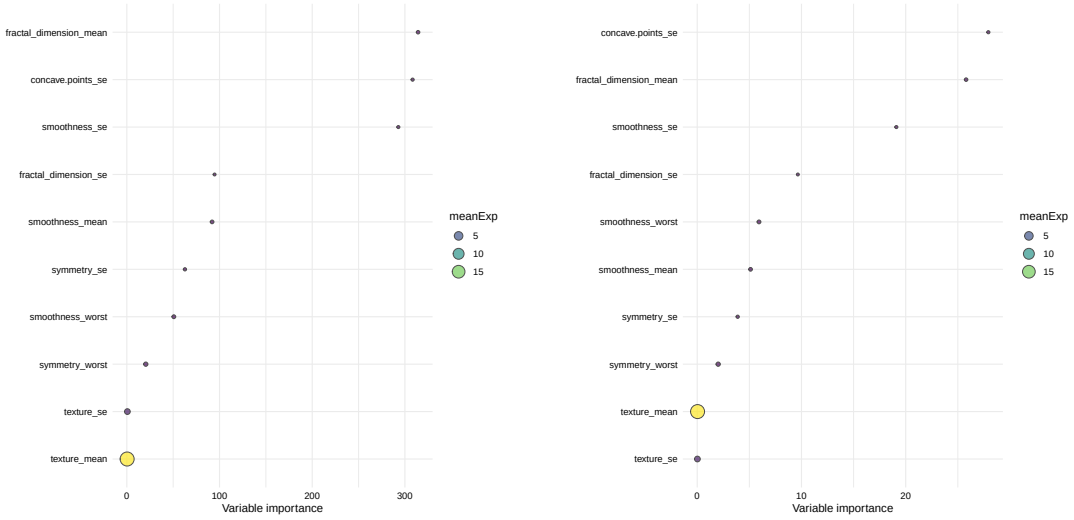
The package also offers a method for visualizing the variable importance plot (Figures 5.4, 5.5, 5.6, 5.7, 5.8). We can see on the x-axis the measure of 'Variable Importance'. This measure typically represents the contribution of each feature, or predictor, to the model. The higher the value, the more important the feature is in making accurate predictions according to the model. On the y-axis, we have the features (predictors) themselves. The `'meanExp'` in the legend typically represents the mean expected value or contribution of each feature. This can be an average over multiple runs of the model, providing a more robust estimate of each feature's importance.

```
plot_varImp(ncv_bin.full, abs=TRUE, size=TRUE)
plot_varImp(ncv_bin.vif, abs=TRUE, size=TRUE)
plot_varImp(ncv_bin.stepAIC, abs=TRUE, size=TRUE)
plot_varImp(ncv_bin.stepBIC, abs=TRUE, size=TRUE)
plot_varImp(ncv_bin.step2o, abs=TRUE, size=TRUE)
plot_varImp(ncv_gauss.full, abs=TRUE, size=TRUE)
plot_varImp(ncv_gauss.vif, abs=TRUE, size=TRUE)
plot_varImp(ncv_gauss.stepAIC, abs=TRUE, size=TRUE)
plot_varImp(ncv_gauss.stepBIC, abs=TRUE, size=TRUE)
plot_varImp(ncv_gauss.step2o, abs=TRUE, size=TRUE)
```



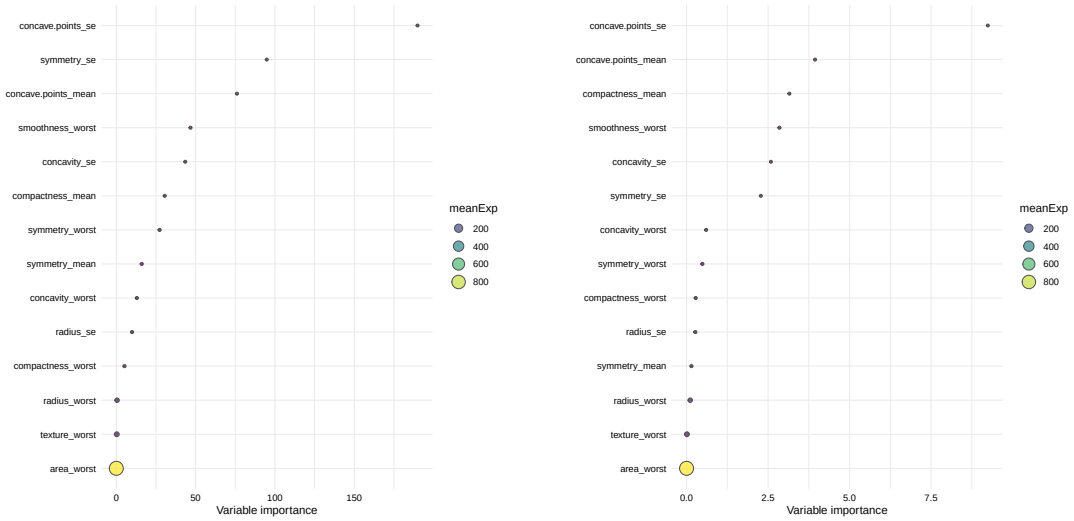
(a) `NCV Binomial family` (b) `NCV Gaussian famiy`

Figure 5.4: Variable Importance of `glmnet full model` Nested Cross-Validation Plot



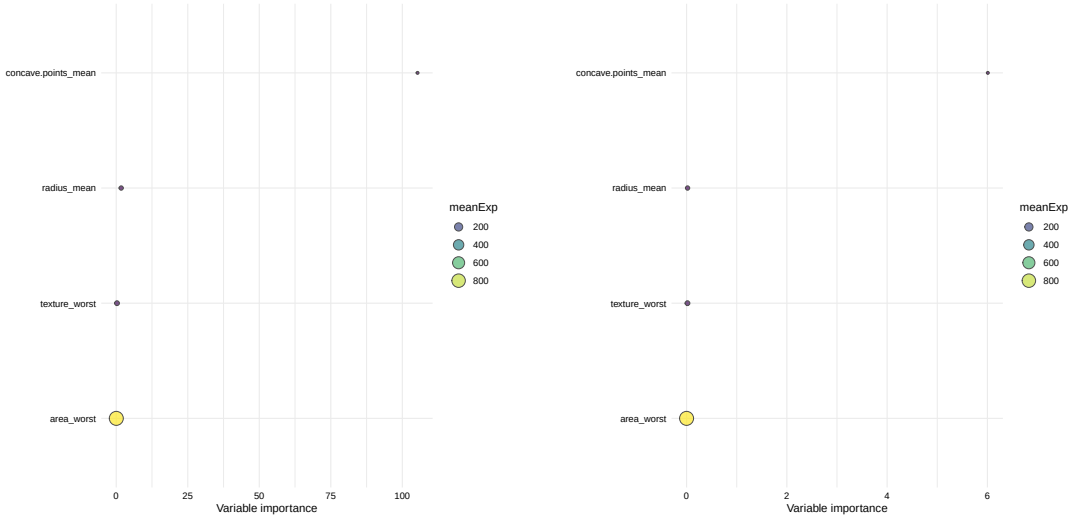
(a) `NCV Binomial family` (b) `NCV Gaussian family`

Figure 5.5: Variable Importance of `glmnet VIF model` Nested Cross-Validation Plot



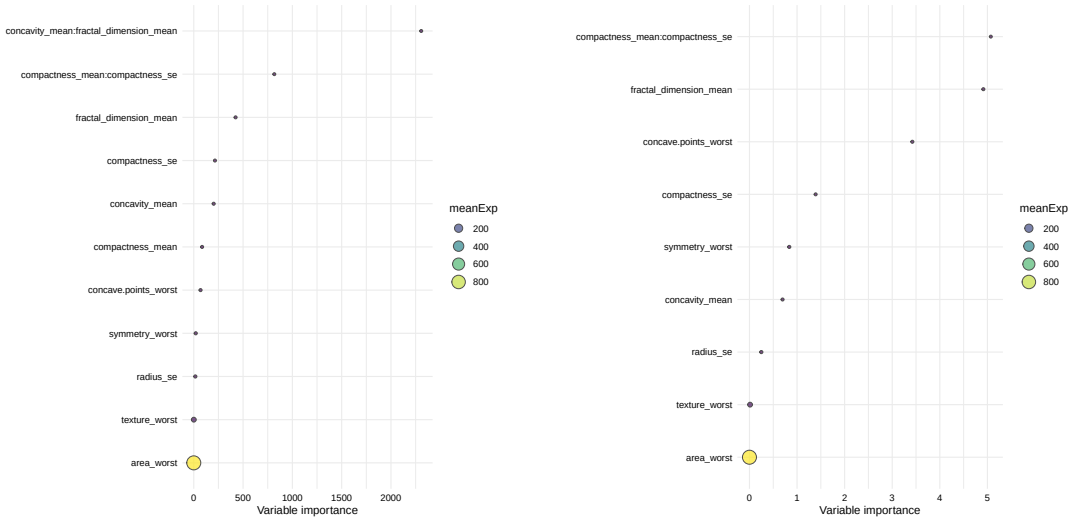
(a) `NCV Binomial family` (b) `NCV Gaussian family`

Figure 5.6: Variable Importance of `glmnet best stepAIC model` Nested Cross-Validation Plot



(a) `NCV Binomial family` (b) `NCV Gaussian family`

Figure 5.7: Variable Importance of `glmnet best stepBIC model` Nested Cross-Validation Plot



(a) `NCV Binomial family` (b) `NCV Gaussian family`

Figure 5.8: Variable Importance of `glmnet best second order stepAIC model` Nested Cross-Validation

The following R code provides us with a visualization of the performance of a generalized linear model (glmnet) under different hyperparameters. The results are shown in Figures 5.9 and 5.10. On the x-axis, we have the logarithm of the lambda hyperparameter. Lambda is a regularization parameter in glmnet models, controlling the amount of shrinkage. On the y-axis, we have the measure of model performance. For a binomial glmnet model, this is the binomial deviance, which is a measure of the difference between the predicted probabilities and the actual outcomes. For a Gaussian glmnet model, this is the mean squared error (MSE). The different lines in the plot represent different values of the alpha hyperparameter. Alpha controls the type of regularization used in the glmnet model: an alpha of 0 corresponds to ridge regression, an alpha of 1 corresponds to lasso regression and values in between correspond to a mix of the two, known as elastic net.

```
par(mfrow=c(3,5))

#The tuning of alpha for each outer fold can be plotted.
plot(ncv_bin.full$outer_result[[1]]$cvafit, main="(full model)")
plot(ncv_bin.vif$outer_result[[1]]$cvafit, main="(VIF model)")
plot(ncv_bin.stepAIC$outer_result[[1]]$cvafit, main="(StepAIC model)")
plot(ncv_bin.stepBIC$outer_result[[1]]$cvafit, main="(StepBIC model)")
plot(ncv_bin.step2o$outer_result[[1]]$cvafit, main="(Step2o model)")
# scatter plot
plot(ncv_bin.full$outer_result[[1]]$cvafit,
     type = 'p', main="(full model)")
plot(ncv_bin.vif$outer_result[[1]]$cvafit,
     type = 'p', main="(VIF model)")
plot(ncv_bin.stepAIC$outer_result[[1]]$cvafit,
     type = 'p', main="(StepAIC model)")
plot(ncv_bin.stepBIC$outer_result[[1]]$cvafit,
     type = 'p', main="(StepBIC model)")
plot(ncv_bin.step2o$outer_result[[1]]$cvafit,
     type = 'p', main="(Step2o model)")
# number of non-zero coefficients
plot(ncv_bin.full$outer_result[[1]]$cvafit,
     xaxis = 'nvar', main="(full model)")
plot(ncv_bin.vif$outer_result[[1]]$cvafit,
     xaxis = 'nvar', main="(VIF model)")
plot(ncv_bin.stepAIC$outer_result[[1]]$cvafit,
     xaxis = 'nvar', main="(StepAIC model)")
plot(ncv_bin.stepBIC$outer_result[[1]]$cvafit,
     xaxis = 'nvar', main="(StepBIC model)")
plot(ncv_bin.step2o$outer_result[[1]]$cvafit,
     xaxis = 'nvar', main="(Step2o model)")
### Repeat the same for gaussian NCV ###
```

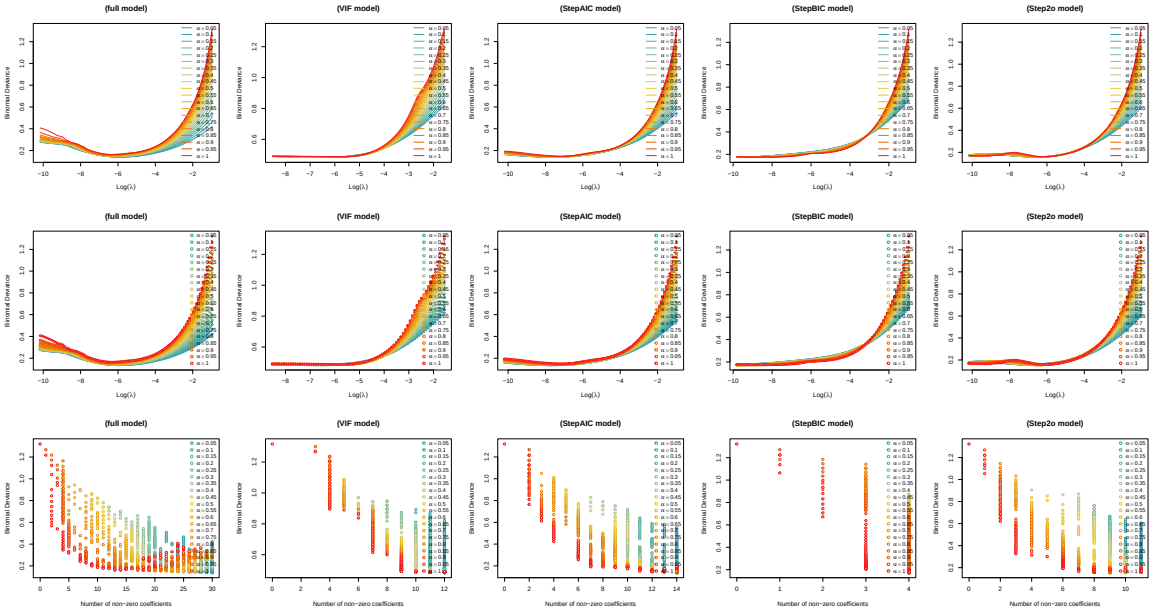


Figure 5.9: Binomial `nested CV outer fold 1`

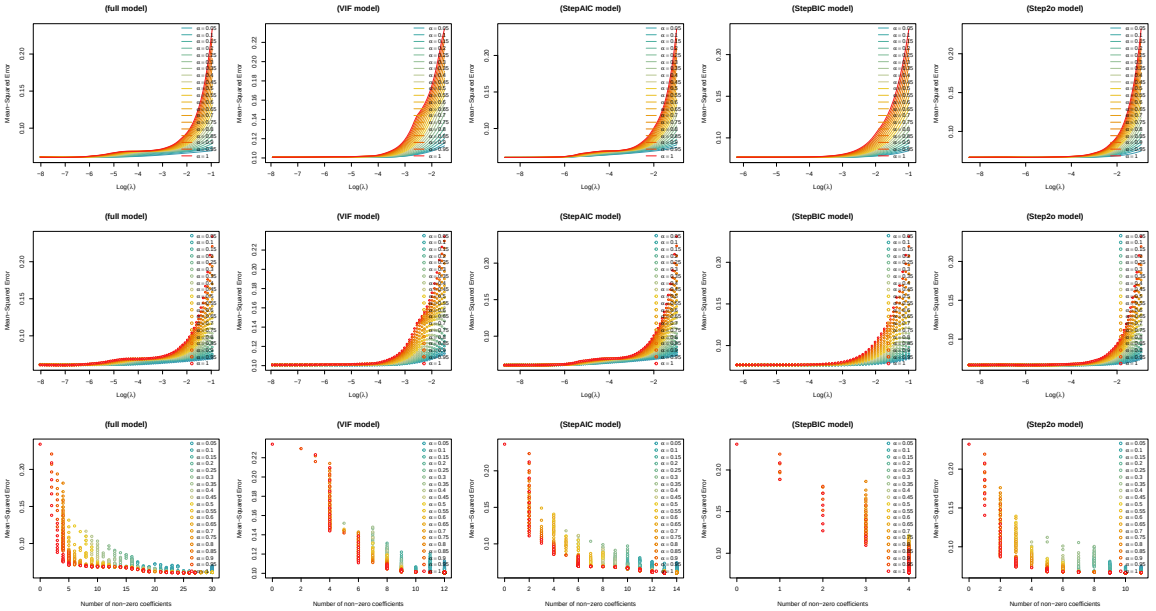


Figure 5.10: Gaussian `nested CV outer fold 1`

We can achieve the following results by employing the `model$final_param` and `model$final_summary` methods. It's important to note that for stepwise regression, the stepwise procedure should be incorporated within the cross-validation process. However, in the table below, this isn't the case, so we won't place much emphasis on it. In subsequent sections, we will explore how to properly implement this.

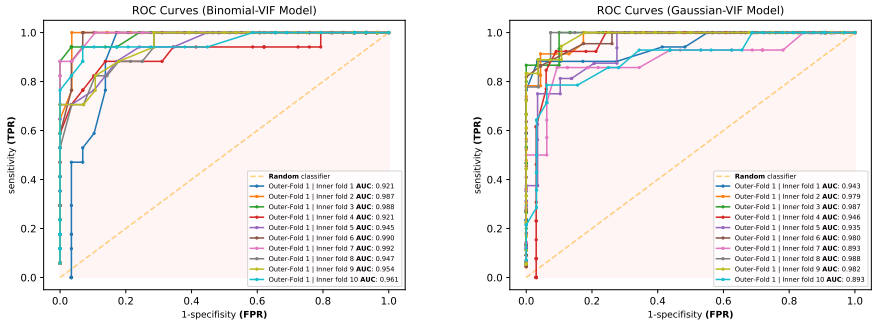
Metric/HP	ncv_bin.full	ncv_bin.vif	ncv_bin.stepAIC	ncv_bin.stepBIC	ncv_bin.step2o
ROC AUC	0.9950	0.9559	0.9963	0.9927	0.9936
Accuracy	0.9754	0.9086	0.9772	0.9701	0.9701
Balanced accuracy	0.9708	0.8956	0.9741	0.9647	0.9657
F1	0.981	0.927	0.981	0.976	0.975
tune-lambda	0.0023	0.0037	0.0003	0.0000	0.0001
tune-alpha	0.1	1	0	0.3	0.9

Table 5.1: `Binomial nested CV` performance summary

Metric/HP	ncv_gauss.full	ncv_gauss.vif	ncv_gauss.stepAIC	ncv_gauss.stepBIC	ncv_gauss.step2o
ROC AUC	0.9940	0.9562	0.9946	0.9919	0.9917
Accuracy	0.958	0.902	0.967	0.935	0.953
Balanced accuracy	0.948	0.894	0.957	0.918	0.939
F1	0.966	0.922	0.974	0.949	0.963
tune-lambda	0.0011	0.0065	0.0002	0.0063	0.0102
tune-alpha	0.95	1	0	0	0.25

Table 5.2: `Gaussian nested CV` performance summary

Furthermore, the figures below displays the ROC curves on the outer fold 1 for each inner fold of the VIF model, both for the Gaussian and Binomial families.



(a) `Outer fold 1 - binomial`

(b) `Outer fold 1 - gaussian`

5.3 Repeated Cross-Validation

Repeated cross-validation is an extension of K-fold cross-validation, a widely used technique for assessing the performance of machine learning models. Both methods involve dividing the dataset into subsets, but they differ in the number of times the process is performed.

Repeated cross-validation extends the K-fold cross-validation by performing the entire K-fold process multiple times with different random splits of the data into folds. Each repetition of the K-fold cross-validation produces a set of performance metrics (e.g., `ROC AUC`, `F1-score`) and the results from all repetitions are averaged to provide an even more robust estimation of the model's performance.

By repeating the K-fold cross-validation process, repeated cross-validation can further reduce the variance in the performance estimates, leading to a more reliable assessment of the model's true performance. However, it comes at the cost of increased computational resources and time, as the model needs to be trained and tested multiple times for each repetition.

In summary, repeated cross-validation improves upon K-fold cross-validation by repeating the entire K-fold process multiple times, providing a more robust performance estimate at the expense of increased computational complexity.

5.4 Bootstrap Method

The bootstrap method is a statistical technique used for estimating the variability and accuracy of a given estimator or statistic by resampling from the original data set. The term "bootstrap" is derived from the phrase "to pull oneself up by one's bootstraps", which represents the idea of using the data itself to generate more information about the estimator.

The bootstrap method involves repeatedly drawing random samples with replacement from the original data set and computing the statistic of interest for each resampled data set. By calculating the statistic on many resampled data sets, a distribution can be formed, which provides insight into the variability and accuracy of the estimator.

Bootstrap is useful because it can be applied to complex models and data sets when traditional parametric methods are not feasible or when the assumptions for those methods are not met. It allows for nonparametric estimation of confidence intervals, hypothesis testing and model selection. This versatile technique has been widely used in various fields, such as biology, finance, social sciences and engineering.

In his 1992 paper [11], Bradley Efron first presented the bootstrap method. This groundbreaking work revolutionized statistical inference by providing a computationally feasible and conceptually straightforward approach to estimate the sampling distribution of a statistic. Efron's contributions to the field of statistics have been widely recognized and he has received numerous awards and honors for his work, including the National Medal of Science in 2005.

5.4.1 Improving Logistic Regression through Bootstrapping

The paper [3] delves into the application of bootstrap techniques for building predictive models, particularly concentrating on logistic regression models. Bootstrap methods generate several new datasets by resampling the original data with replacement, which allows for the evaluation of a predictive model's stability and performance.

The authors emphasize the importance of harnessing bootstrap methods for dual purposes: model development and model validation. They propose a three-phase approach for creating and validating predictive models using bootstrap techniques:

1. `Model Development:` Fit the logistic regression model to the initial dataset and identify statistically significant predictors.
2. `Model Validation:` Employ bootstrap samples to estimate optimism, a metric of overfitting. Optimism represents the difference between the model's performance on the original data and its performance on the bootstrap samples.
3. `Model Assessment:` Correct the model's performance measures for optimism.

By implementing the bootstrap method in this way, researchers can develop more stable and accurate predictive models while also addressing overfitting concerns.

5.4.2 Confidence Interval using Bootstrapping

The `bootstrap method` is a powerful, non-parametric statistical technique for estimating the sampling distribution of a statistic, such as a confidence interval, by resampling from the original data. It is particularly useful when the underlying distribution is unknown, complex, or difficult to work with. The basic idea of the bootstrap method for calculating a confidence interval involves the following steps:

1. Obtain a sample: Start with an original sample of size n, drawn from a population.

2. Resample with replacement: Generate a large number of bootstrap samples, typically thousands, by randomly drawing n observations from the original sample with replacement. This means that an observation can be selected multiple times and some observations might not be selected at all in a given bootstrap sample.

3. Calculate the statistic of interest: For each bootstrap sample, compute the statistic of interest, such as the mean, median, or standard deviation. This will result in a collection of statistics, one for each bootstrap sample.

4. Estimate the sampling distribution: The collection of statistics calculated in the previous step serves as an approximation of the sampling distribution of the statistic of interest. This approximation allows us to make inferences about the population without making strong assumptions about the underlying distribution.

5. Compute the confidence interval: To calculate the confidence interval, determine the percentiles of the bootstrap sampling distribution that correspond to the desired level of confidence. For example, for a 95% confidence interval, you would find the 2.5th and 97.5th percentiles of the bootstrap sampling distribution. The values at these percentiles serve as the lower and upper bounds of the confidence interval, respectively.

It is important to note that the bootstrap method relies on the assumption that the original sample is representative of the population. If the original sample is biased or too small, the bootstrap confidence interval may not accurately reflect the true population parameter. Additionally, the bootstrap method can be computationally intensive, especially for large datasets or complex statistics, but advances in computing power and parallel processing have made it increasingly accessible for many applications.

5.4.3 Bootstrap Confidence Interval for Correlation Coefficient using BCA Method

The Two-sided 95% bootstrap confidence interval for the true Pearson correlation coefficient can be calculated using the bias-corrected and accelerated (BCa) bootstrap interval, with the aid of the `confintr` package from `R`. Specifically, we obtain this interval by applying the BCa method to 9000 bootstrap replications.

```
library(confintr)

lower_bound_spearman <- list()
upper_bound_spearman <- list()
spearman_vals <- list()

# Loop through the columns of the data frame
for (i in 2:dim(data)[2]) {
```

```
    print(paste("For", names(data[c(1,i)])[2]))

    ci_result <- ci_cor(data[c(1,i)],
                        method = "spearman",
                        type="bootstrap",
                        seed=1,
                        R=9000)
    #print(ci_result)

    ## 2.5% , 97.5%  CI ##
    lower_bound_spearman[[i - 1]] <- ci_result$interval[1]
    upper_bound_spearman[[i - 1]] <- ci_result$interval[2]
    spearman_vals[[i - 1]] <- ci_result$estimate
    }

# Convert the list to a vector
lower_bound_spearman <- unlist(lower_bound_spearman)
upper_bound_spearman <- unlist(upper_bound_spearman)
spearman_vals <- unlist(spearman_vals)

lower_bound_pearson <- list()
upper_bound_pearson <- list()
pearson_vals <- list()

# Loop through the columns of the data frame
for (i in 2:dim(data)[2]) {
    print(paste("For", names(data[c(1,i)])[2]))

    ci_result <- ci_cor(data[c(1,i)],
                        method = "pearson",
                        type="bootstrap",
                        seed=1,
                        R=9000)

    print(ci_result)

    ## 2.5% , 97.5%  CI ##
    lower_bound_pearson[[i - 1]] <- ci_result$interval[1]
    upper_bound_pearson[[i - 1]] <- ci_result$interval[2]
    pearson_vals[[i - 1]] <- ci_result$estimate
    }

# Convert the list to a vector
lower_bound_pearson <- unlist(lower_bound_pearson)
upper_bound_pearson <- unlist(upper_bound_pearson)
pearson_vals <- unlist(pearson_vals)

corr_df <- data.frame(Predictor=names(data)[2:31],
                      Pearson=round(pearson_vals, 5),
```

```
                    lower_bound=round(lower_bound_pearson, 5),
                    upper_bound=round(upper_bound_pearson, 5),
                    Spearman=round(spearman_vals, 5),
                    lower_bound_=round(lower_bound_spearman, 5),
                    upper_bound_=round(upper_bound_spearman, 5))
corr_df
```

Predictor	**Pearson**	lower bound	upper bound	**Spearman**	lower bound	upper bound
radius_mean	0.73003	0.69541	0.76039	0.73278	0.69079	0.76661
texture_mean	0.41519	0.34063	0.47980	0.46197	0.39256	0.52403
perimeter_mean	0.74264	0.71021	0.77157	0.74850	0.70994	0.77931
area_mean	0.70898	0.66847	0.74364	0.73412	0.69215	0.76765
smoothness_mean	0.35856	0.28148	0.42640	0.37189	0.29815	0.43964
compactness_mean	0.59653	0.54192	0.64305	0.60929	0.55170	0.65946
concavity_mean	0.69636	0.63084	0.73925	0.73331	0.69206	0.76747
concave points_mean	0.77661	0.74436	0.80335	0.77788	0.74640	0.80373
symmetry_mean	0.33050	0.25333	0.39968	0.33257	0.25534	0.40325
fractal_dimension_mean	−0.01284	−0.09515	0.07372	−0.02590	−0.11221	0.06159
radius_se	0.56713	0.49792	0.62345	0.61691	0.56050	0.66715
texture_se	−0.00830	−0.08328	0.07383	0.01942	−0.06081	0.10027
perimeter_se	0.55614	0.48539	0.61296	0.63041	0.57621	0.67771
area_se	0.54824	0.45440	0.65238	0.71418	0.66963	0.75130
smoothness_se	−0.06702	−0.14181	0.02210	−0.05219	−0.13508	0.02580
compactness_se	0.29300	0.21004	0.36880	0.38067	0.30916	0.44820
concavity_se	0.25373	0.11886	0.37230	0.47034	0.40368	0.53022
concave points_se	0.40804	0.31412	0.47855	0.48872	0.42251	0.54802
symmetry_se	−0.00652	−0.09495	0.08839	−0.09230	−0.17355	−0.00920
fractal_dimension_se	0.07797	−0.00587	0.16538	0.20149	0.12126	0.27892
radius_worst	0.77645	0.74800	0.80105	0.78793	0.75748	0.81154
texture_worst	0.45690	0.38611	0.51528	0.47672	0.40867	0.53634
perimeter_worst	0.78291	0.75462	0.80630	0.79632	0.76762	0.81831
area_worst	0.73383	0.69310	0.76574	0.78690	0.75629	0.81112
smoothness_worst	0.42146	0.35187	0.48504	0.42551	0.35467	0.49304
compactness_worst	0.59100	0.53965	0.63672	0.60681	0.55087	0.65802
concavity_worst	0.65961	0.58459	0.70715	0.70573	0.66097	0.74363
concave points_worst	0.79357	0.76175	0.81821	0.78167	0.74765	0.80790
symmetry_worst	0.41629	0.34860	0.47406	0.39684	0.32211	0.46658
fractal_dimension_worst	0.32387	0.24605	0.39382	0.31148	0.23030	0.38813

Table 5.3: Two-sided 95% bootstrap confidence interval for the true Pearson/Spearman correlation coefficient based on 9000 bootstrap replications and the bca method

5.4.4 StepAIC with Bootstrapping

The bootStepAIC package in R is used for bootstrap model selection using the stepwise Akaike Information Criterion (AIC) method. The boot.stepAIC function performs the bootstrap stepwise AIC procedure on a given dataset. The output of the function is a list containing the following elements:

- Covariates: This element contains a matrix of the covariates (independent variables) used in the bootstrap stepwise AIC procedure. Each row of the matrix represents a bootstrap sample and each column corresponds to a specific covariate.
- Sign: This element contains a matrix of the same dimensions as the Covariates matrix. It represents the sign of the estimated coefficients for each covariate in the bootstrap samples. A positive sign indicates a positive relationship between the covariate and the dependent variable, while a negative sign suggests a negative relationship.
- Significance: This element contains a matrix of p-values for the estimated coefficients in each bootstrap sample. p-values are used to test the null hypothesis that the corresponding covariate has no effect on the dependent variable. Lower p-values indicate that the null hypothesis can be rejected, suggesting a significant relationship between the covariate and the dependent variable.

We simulate 30 bootstrap samples for each boot.stepAIC model:

```
library(bootStepAIC)
registerDoMC(cores = 2)
n <- nrow(train_data); logn <- log(n)
bootstrap.stepAIC <- function (init.model, data=train_data,
                               direction, B, k=2) {
    suppressWarnings({
    system.time(boot_stepAIC <- boot.stepAIC(
                                    init.model,
                                    data=data,
                                    scope=list(
                                           lower=null.model,
                                           upper=mle.fit),
                                    direction=direction, B=B, k=k))
        })
    return (boot_stepAIC) }
boot_stepAIC.forward <- bootstrap.stepAIC(null.model,
                                          direction='forward', B=30)
boot_stepAIC.backward <- bootstrap.stepAIC(full.model,
                                          direction='backward', B=30)
boot_stepBIC.forward <- bootstrap.stepAIC(null.model,
                                          direction='forward', B=30,
                                          k=logn)
```

```
boot_stepBIC.backward <- bootstrap.stepAIC(full.model,
                                           direction='backward', B=30,
                                           k=logn)
boot_stepBIC.forward$Covariates
boot_stepAIC.backward$Covariates
```

boot_stepAIC.backward	(%)
area_worst	96.67
fractal_dimension_worst	90.00
compactness_mean	86.67
smoothness_se	86.67
radius_worst	83.33
concave.points_mean	80.00
concavity_se	80.00
texture_worst	76.67
radius_mean	66.67
radius_se	63.33
symmetry_se	56.67
symmetry_worst	53.33
concavity_worst	50.00
texture_mean	50.00
texture_se	50.00
perimeter_mean	46.67
area_mean	43.33
concave.points_worst	43.33
fractal_dimension_se	43.33
compactness_se	40.00
compactness_worst	40.00
concavity_mean	40.00
smoothness_mean	36.67
concave.points_se	33.33
fractal_dimension_mean	33.33
perimeter_worst	33.33
smoothness_worst	30.00
area_se	26.67
perimeter_se	20.00
symmetry_mean	20.00

boot_stepBIC.forward	(%)
texture_worst	83.33
perimeter_worst	80.00
smoothness_worst	66.67
concave.points_worst	60.00
area_worst	56.67
symmetry_worst	46.67
compactness_se	40.00
concave.points_mean	40.00
symmetry_mean	33.33
concavity_mean	26.67
area_se	23.33
texture_mean	23.33
texture_se	23.33
compactness_mean	20.00
fractal_dimension_worst	20.00
perimeter_mean	20.00
concavity_worst	16.67
radius_se	16.67
smoothness_mean	16.67
concave.points_se	13.33
perimeter_se	10.00
symmetry_se	10.00
area_mean	6.67
concavity_se	6.67
fractal_dimension_mean	6.67
fractal_dimension_se	6.67
radius_mean	6.67
radius_worst	6.67
smoothness_se	6.67
compactness_worst	3.33

Table 5.4: Stepwise selection (via `AIC/BIC`) with bootstrapping

The table above presents the percentage of variable selection in multiple bootstraps of the dataset. It is evident that based on `AIC`, variables such as `'area_worst'`, `'fractal dimension_worst'`, `'compactness_mean'` and others, are consistently chosen in almost every bootstrap iteration. Similarly, relying on `BIC`, `'texture_worst'` and `'perimeter_worst'` appear to be frequently selected. The green color on the right indicates that those features were selected by the `stepAIC` in the previous chapter, while the red color on the left signifies that those features were removed during the backward procedure. For the forward selection, we can observe that the `'perimeter_mean'` doesn't seem to be that important for the model, whereas `'symmetry_worst'` appears to be a better

choice. For the backward selection, it's noticeable that we have discarded important features like 'fractal_dimension_worst' and 'smoothness_se'. According to the bootstrap analysis, these could provide valuable information.

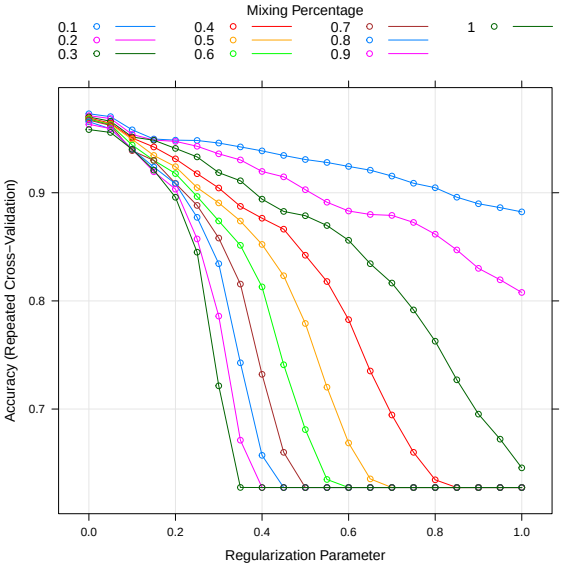
5.5 Assessing Model Generalization: Repeated 10-Fold Cross-Validation

Repeated cross-validation offers a more robust performance estimate by reducing variability through multiple iterations. It maximizes data usage, allowing the model to train and test on various data combinations, potentially yielding a more generalized model. Repeated CV helps detect instability in model performance by identifying significant variation across iterations. It also mitigates bias and variance that can occur due to random data splitting. In R:

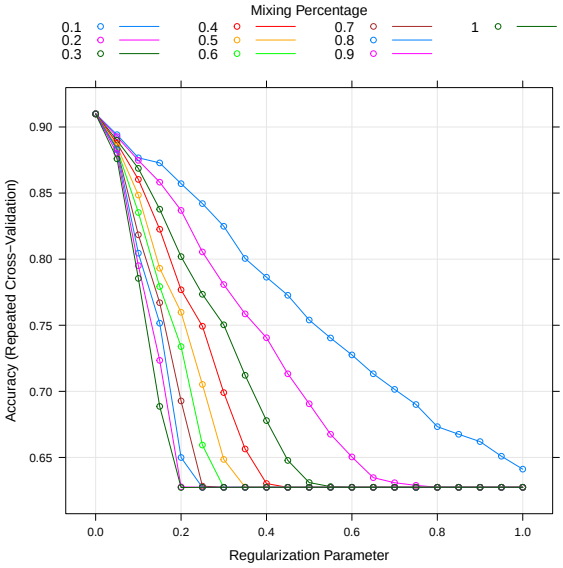
```
# Repeated Cross Validation
library(caret)
library(MASS)
eGrid <- expand.grid(.alpha = (1:10) * 0.1, .lambda = seq(0, 1, 0.05))
Control <- trainControl(method = "repeatedcv", repeats = 100,
                        verboseIter=FALSE)
repeated_CV_binom_train <- function (X_) {
    system.time(
        rcv <- train(
                x=X_, y=as.factor(data$y), method="glmnet",
                family = "binomial", tuneGrid=eGrid,
                type.measure='auc', trControl=Control))
    return (rcv)
}
rcv_bin.full <- repeated_CV_binom_train(X)
rcv_bin.vif <- repeated_CV_binom_train(X[names(VIF_data)[-1]])
rcv_bin.stepAIC <- repeated_CV_binom_train(
                        X[names(best_stepAIC.data)[-1]])
rcv_bin.stepBIC <- repeated_CV_binom_train(
                        X[names(best_stepBIC.data)[-1]])
rcv_bin.step2o <- repeated_CV_binom_train(data_2o)
```

The following code provides us with a visualization of the repeated cross-validation during the training. The results are shown in the Figures 5.11. On the x-axis, we have the regularization parameter (λ) and on the y-axis we have the accuracy. Each color corresponds to a specific value of the hyperparameter α.

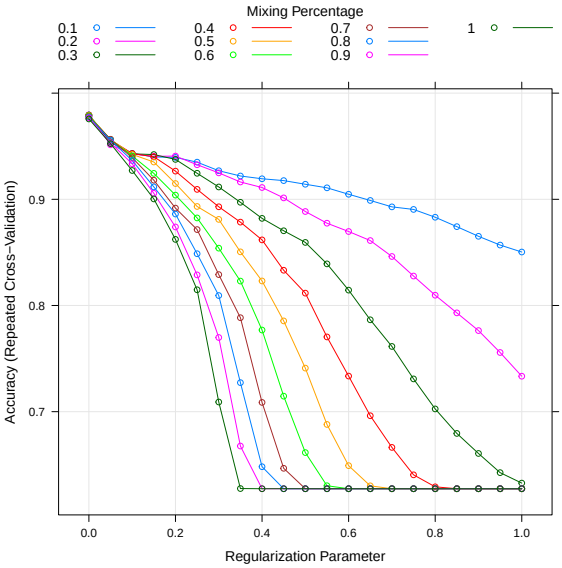
```
plot(rcv_bin.full)
plot(rcv_bin.vif)
plot(rcv_bin.stepAIC)
plot(rcv_bin.stepBIC)
plot(rcv_bin.step2o)
```



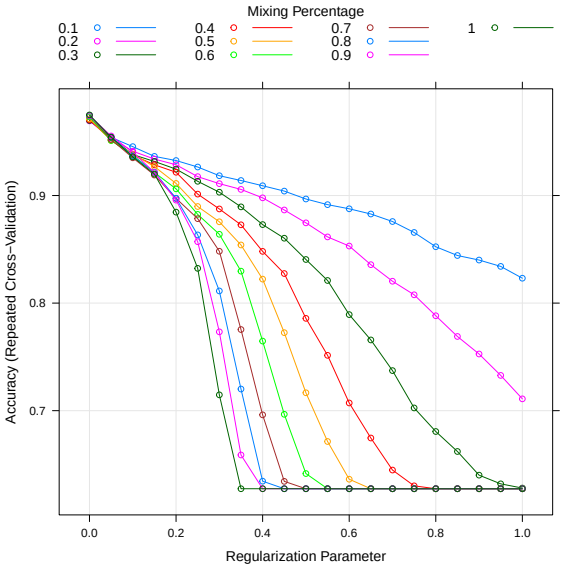
(c) `Full model`



(d) `VIF model`



(e) `Stepwise AIC/BIC model`



(f) `Stepwise 2nd order model`

Figure 5.11: `RepeatedCV` regularization parameters

One can compute each confidence interval by calculating the 0.025 and 0.975

quantiles of the accuracies obtained from the corresponding cross-validation loop (Figure 5.12). Let's explore how to do this in R.

```
rcv.hist <- function(model, main='histogram', bins=8) {
    rcv.acc <- model$resample$Accuracy
    hist(rcv.acc, breaks = bins,
         xlab = "Accuracy", main = main,
         col = "skyblue", border = "black")

    # Calculate the 95% confidence interval
    ci <- quantile(rcv.acc, probs = c(0.025, 0.975))

    # Add vertical lines to the histogram
    abline(v = ci[1], col = "red",
           lwd = 2, lty = 2)  # lower quantile
    abline(v = ci[2], col = "red",
           lwd = 2, lty = 2) # upper quantile
    text(x = mean(ci), y=max(hist(rcv.acc, plot = FALSE)$counts),
         labels="95% Conf. Int.", adj=c(0.5, -0.5), col = "red")
    # Add a legend with the confidence interval values
    legend("topleft",
           legend=paste("95% CI: (", round(ci[1], 2), ",",
                        round(ci[2], 2), ")", sep = ""),
           lty = 2, lwd = 2, col = "red")
    }

### plot the histograms ###
par(mfrow=c(2,3))

rcv.hist(rcv_bin.full, main="rcv_bin.full model")
rcv.hist(rcv_bin.vif, bins=16,
         main="rcv_bin.vif model")
rcv.hist(rcv_bin.stepAIC, bins=10,
         main="rcv_bin.stepAIC model")
rcv.hist(rcv_bin.stepBIC, bins=10,
         main="rcv_bin.stepBIC model")
rcv.hist(rcv_bin.step2o, bins=10,
         main="rcv_bin.step2o model")
```

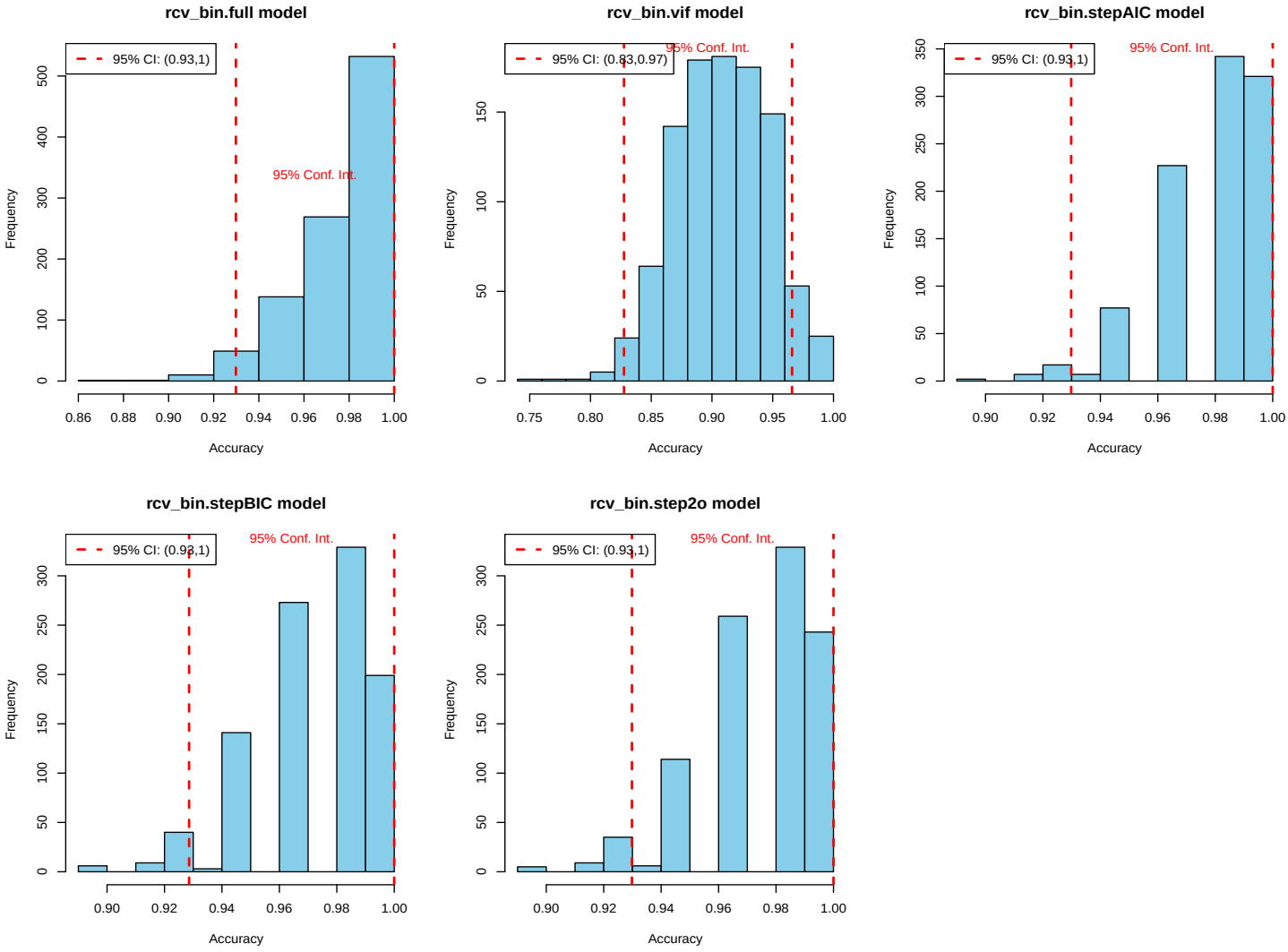


Figure 5.12: `RepeatedCV` accuracies & 95% CI

```
rbind(rcv_bin.full$bestTune, rcv_bin.vif$bestTune,
      rcv_bin.stepAIC$bestTune, rcv_bin.stepBIC$bestTune,
      rcv_bin.step2o$bestTune)[1]
```

	alpha
rcv.full	0.1
rcv.vif	0.1
rcv.stepAIC	0.3
rcv.stepBIC	1
rcv.step2o	1

Table 5.5: `RepeatedCV` tune regularization

```
meanRCV1 <- round(mean(rcv_bin.full$resample$Accuracy), 4)
meanRCV2 <- round(mean(rcv_bin.vif$resample$Accuracy), 4)
meanRCV3 <- round(mean(rcv_bin.stepAIC$resample$Accuracy), 4)
meanRCV4 <- round(mean(rcv_bin.stepBIC$resample$Accuracy), 4)
meanRCV5 <- round(mean(rcv_bin.step2o$resample$Accuracy), 4)

meanRCV <- data.frame(
  rcv_bin.full = meanRCV1,
  rcv_bin.vif = meanRCV2,
  rcv_bin.stepAIC = meanRCV3,
  rcv_bin.stepBIC = meanRCV4,
  rcv_bin.step2o = meanRCV5
)
rownames(df) <- "Average RCV acc."

meanRCV
```

	rcv_bin.full	rcv_bin.vif	rcv_bin.stepAIC	rcv_bin.stepBIC	rcv_bin.step2o
Mean RCV acc.	0.973	0.91	0.9794	0.9727	0.9749

Table 5.6: Mean `RepeatedCV` accuracies

5.6 Improving Performance through Model Averaging

Model averaging is a simple yet powerful method in statistics. It combines the insights from many different models to give more dependable predictions than a single model. Instead of choosing one 'best' model using metrics like `AIC` or `BIC`, this method looks at all possible models, taking an average of their predictions. The weight given to each model's prediction often depends on how well it performs.

The beauty of this approach lies in its numerous advantages. Foremost, it deftly manages the uncertainty that often surrounds model selection. It's not unusual for different models to yield similarly high performances, making it challenging to select the top performer. Model averaging, however, takes a more holistic approach, incorporating data from all models to deliver a well-rounded prediction. Besides, it enhances predictive performance by leveraging the strengths of different models. The outcome is often more accurate and trustworthy predictions than those derived from a single model.

But model averaging does come with some challenges. One big challenge is that it requires a lot of computing power. It needs to fit and evaluate many models, which can be demanding on a computer. This is where special software tools like the `MuMIn` package can really help.

`MuMIn` (which stands for Multi-Model Inference) is a handy tool in `R` that provides functions for choosing models and averaging them based on information criteria. It works with many types of models, including Generalized Linear Models (`GLMs`).

Model averaging comes in handy particularly when we face a state of model uncertainty. This situation arises when several models fit the data almost equally well, making it tough to identify the superior one. Model uncertainty typically comes into play when we're dealing with a host of predictor variables, opening up the possibility to create countless credible models.

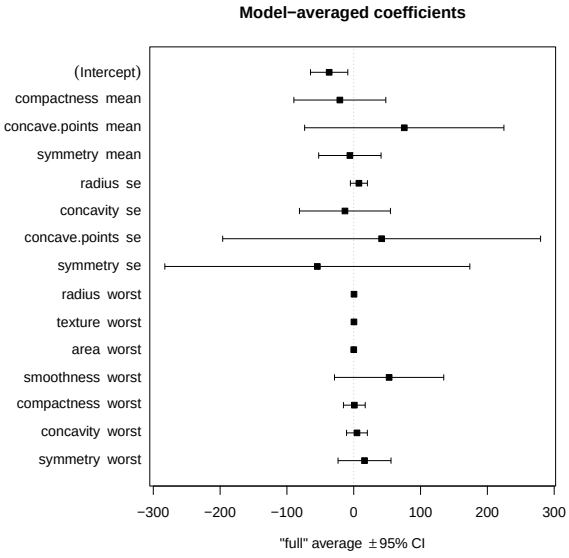
To put it simply, whenever we have various models, each of which is convincing or valuable in its own manner and our goal is to gather insights from all to achieve the most precise predictions, model averaging proves to be an extremely beneficial tool.

```
library(MuMIn)
#arm: Adaptive Regression by Mixing
suppressWarnings({
    stepAIC.glm <- glm(y ~.,
                   data=data[names(best_stepAIC.data)],
```

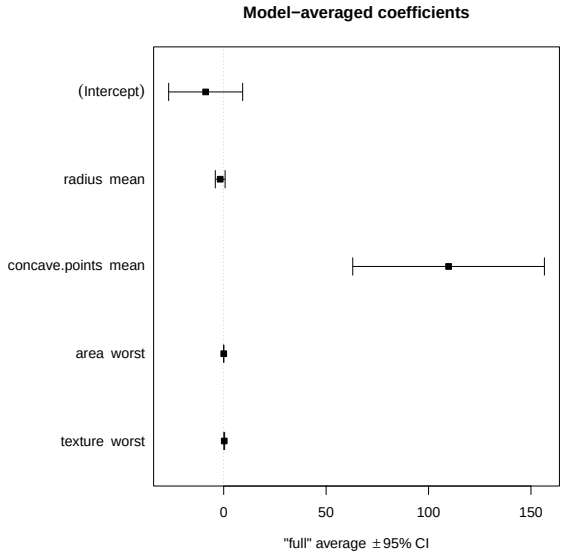
```
                    family=binomial(link='logit'))
system.time(
    stepAIC.arm_glm <- arm.glm(stepAIC.glm,
                               R=8, #permutations
                               weight.by='aic'))})
stepBIC.glm <- glm(y ~.,
                    data=data[names(best_stepBIC.data)],
                    family=binomial(link='logit'))
system.time(
    stepBIC.arm_glm <- arm.glm(stepBIC.glm,
                               R=300, #permutations
                               weight.by='aic'))

suppressWarnings({step2o.glm <- glm(y ~.,
                    data=cbind(y=data$y,data_2o),
                    family=binomial(link='logit'))

system.time(
    step2o.arm_glm <- arm.glm(step2o.glm,
                               R=8, #permutations
                               weight.by='aic'))})
suppressWarnings({vif.glm <- glm(y ~.,
                    data=data[names(VIF_data)],
                    family=binomial(link='logit'))
system.time(
    vif.arm_glm <- arm.glm(vif.glm,
                               R=10, #permutations
                               weight.by='aic'))})
### View the avg coeff ###
plot(stepAIC.arm_glm)
plot(step2o.arm_glm)
plot(vif.arm_glm)
plot(stepBIC.arm_glm)
```

In Figure 5.13, we see a representation of model-averaged coefficients obtained from the MuMIn package in R. This plot provides a visual summary of the average impact of each feature in the model, along with their 95% confidence intervals. On the y-axis, we have the features or variables used in the model. The x-axis, on the other hand, displays the "full" average effect size of these features. This average is calculated from all models considered in the model averaging process, which is a technique used to account for model uncertainty.

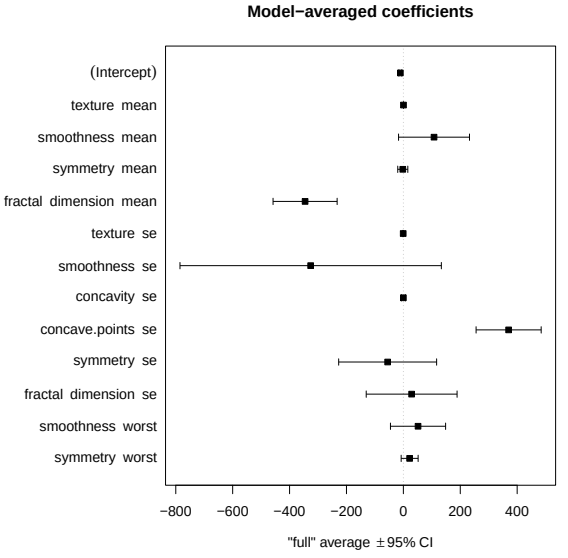
By examining Figure 5.13, we can gain insights into the importance and impact of each feature in our model. Features with larger average effect sizes and smaller confidence intervals are generally more influential.



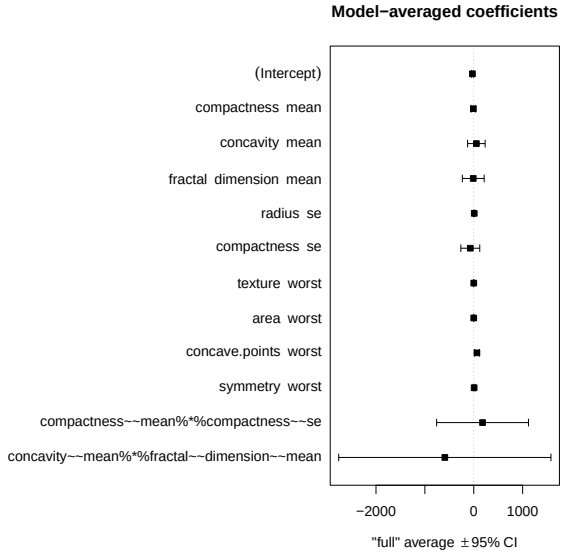
(a) `Stepwise-AIC model`



(b) `Stepwise-BIC model`



(c) `VIF model`



(d) `Stepwise 2nd order model`

Figure 5.13: `Model-averaged` coefficients

5.7 Non-Parametric Ensemble Methods with Bootstrapping

Non-parametric ensemble methods that use bootstrapping include a variety of machine learning algorithms, which merge multiple models to enhance the accuracy of their predictions without relying on assumptions about the data distribution. Ensemble learning techniques, such as bagging, have attracted considerable interest in the field of machine learning. Bagging is an ensemble learning approach that employs bootstrap data to generate an assortment of predictors. The final prediction is achieved by merging the outputs of individual predictors, typically via averaging or majority voting.

A pertinent research paper in this area is the influential work by Leo Breiman [8]. In this publication, Breiman presents the concept of bagging and demonstrates its effectiveness in reducing prediction errors.

5.7.1 Decision Trees & Random Forest

`Decision trees` are a popular machine learning technique used for both classification and regression tasks. They work by breaking down complex problems into smaller, more manageable pieces through a tree-like structure. The tree consists of nodes representing decision points and branches connecting these nodes based on specific conditions or rules.

At the top of the tree is the root node, which represents the initial decision point. From there, the tree branches out into additional nodes, representing further decisions based on certain conditions. These nodes eventually lead to the leaf nodes, which are the final outcomes or predictions.

To create a decision tree, the algorithm selects the best feature or attribute to split the data at each node, based on a specific criterion. For classification tasks, criteria such as `Gini impurity`[1] or `information gain` are commonly used to determine the best split.

Entropy:

$$H(X) = -\sum_{i=1}^{k} P(X = x_i) \log_2 P(X = x_i).$$

Conditional Entropy:

$$H(Y \mid X) = \mathbb{E}_x\big\{H(Y \mid X = x)\big\}.$$

[1]The `Gini impurity` is a measure of how often a randomly chosen element from a dataset would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the dataset, `Gini(t)` $= 1 - \sum p_i^2$

Information Gain:

$$\texttt{IG}(X) = H(Y) - H(Y \mid X).$$

When a new instance is presented to the decision tree for prediction, it starts at the root node and follows the branches based on the conditions at each node until it reaches a leaf node. The prediction is then determined by the majority class in the case of classification or the average target value in the case of regression.

Decision trees are appreciated for their simplicity and ease of interpretation, as they closely resemble human decision-making processes. However, they can be prone to overfitting, which is why techniques like pruning or ensemble methods such as bagging and boosting are often used to improve their performance.

`Random Forest` is an effective ensemble learning method that brings together numerous decision trees to enhance prediction accuracy and minimize overfitting. By constructing a forest of decision trees, with each tree trained on a distinct subset of data, the algorithm combines their individual predictions to produce a final result. The following steps outline how the Random Forest algorithm operates:

- **Bootstrapping:** For each tree in the forest, a random subset of the data is selected with replacement (bootstrapping). This results in different decision trees being trained on different subsets of data.
- **Random feature selection:** At each split in the decision tree, only a random subset of features is considered for splitting the data. This introduces more diversity among the trees and helps to reduce correlation between them.
- **Training individual trees:** Each decision tree is trained independently on its respective subset of data and features. The trees are typically grown to their maximum depth without pruning, which helps to minimize bias.
- **Aggregating predictions:** When a new instance is presented to the Random Forest for prediction, each tree in the forest generates its own prediction. In the case of classification, the final prediction is determined by majority voting among all trees, while for regression tasks, the average prediction of all trees is used.

The `Random Forest` algorithm effectively addresses the overfitting issue common with individual decision trees by averaging the predictions of multiple trees, thus resulting in a more robust and accurate model. Additionally, the random feature selection at each split helps to reduce the correlation between trees, further enhancing the model's performance.

5.7.2 Implementation of Nested Cross-Validation in Python

Algorithm 2 Nested Cross-Validation using `GridSearchCV`

Input: `Dataset` **D**, `outer folds` $\texttt{N}^{\texttt{outer}}$, `inner folds` $\texttt{N}^{\texttt{inner}}$, `parameter grid`
Output: `Average Performances`

for $i = 1$ to $\texttt{N}^{\texttt{outer}}$ **do**
  Split dataset into `outer` training set $\texttt{Train}^{\texttt{outer}}$ and `test` set $\texttt{Test}^{\texttt{outer}}$
  for $j = 1$ to $\texttt{N}^{\texttt{inner}}$ **do**
    Split $\texttt{Train}^{\texttt{outer}}$ into `inner` training set $\texttt{Train}^{\texttt{inner}}$ and validation set $\texttt{Val}^{\texttt{inner}}$
    for all combinations of `hyperparameters` in parameter `grid` **do**
      Train the model on $\texttt{Train}^{\texttt{inner}}$ using the current `hyperparameters`
      Evaluate the model on $\texttt{Val}^{\texttt{inner}}$ and store the `performance metric`
    end for
    Determine the best `hyperparameters` based on the `performance metric` for the current `inner fold`
  end for
    Train the model on $\texttt{Train}^{\texttt{outer}}$ using the best `hyperparameters` found in the inner loop
    Evaluate the model on $\texttt{Test}^{\texttt{outer}}$ and store the `performance metric`
end for
Calculate the `average performance metric` across all `outer folds`

We have developed an implementation of nested cross-validation in Python using the `Scikit-learn` library:

```
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from collections import defaultdict

from sklearn.model_selection import (GridSearchCV,
                                     RandomizedSearchCV,
                                     cross_val_score,
                                     StratifiedKFold)

from sklearn.metrics import (roc_auc_score, f1_score,
                             precision_score, recall_score,
                             average_precision_score,
                             balanced_accuracy_score,
                             matthews_corrcoef,
                             make_scorer, get_scorer)
```

```
class NestedCV:

    """
    This implementation of Nested Cross-Validation was
        developed by Ioannis Maris in 2023.
    It is compatible with Python 3.11. This class is capable of
         performing Nested CV
    for any Scikit-learn classifier, including Random Forest,
        Support Vector Classifier (SVC),
    Logistic Regression and more. Note that to use it for
        regression purposes,
    you only need to change the evaluation metrics to options
        such as
    Mean Squared Error (MSE), Mean Absolute Error (MAE), or R-
        squared (R2).
    """
    import sys
    import os
    import warnings
    if not sys.warnoptions:
        warnings.simplefilter("ignore")
        os.environ["PYTHONWARNINGS"] = "ignore"

    def __init__(self, innercv: int = 10, outercv: int = 10):
        self.innercv = innercv
        self.outercv = outercv

    def __repr__(self):
        return f"NestedCV(inner loops: {self.innercv}, outer
            loops: {self.outercv})"

    def fit(self,
            X: pd.DataFrame,
            y: pd.DataFrame,
            pipeline: Pipeline,
            grid_param: dict,
            trace: bool = True,
            njobs: bool = False):

        response = y.columns[0]; col = list(X.columns)
        arr2df = lambda X: pd.DataFrame(X, columns=col)
        arr2vec = lambda y: pd.DataFrame(y, columns=[response])
        X = np.array(X); y = np.ravel(y) #'revectorize' them
            again

        custom_average_precision_score = lambda y_true, y_pred,
             pos_label=None, needs_proba=True:\
                 average_precision_score(y_true, y_pred,
                     pos_label=pos_label)
```

```
        average_precision_scorer = make_scorer(
                            custom_average_precision_score,
                            pos_label=1, needs_proba=True)

        scoring_metrics = {
            'roc_auc': 'roc_auc',
            'F1': make_scorer(f1_score),
            'F1_macro': 'f1_macro',
            'precision': make_scorer(precision_score),
            'recall': make_scorer(recall_score),
            'average_precision': average_precision_scorer,
            'balanced_accuracy': make_scorer(
               balanced_accuracy_score),
            'accuracy': 'accuracy',
            'matthews_corrcoef': make_scorer(matthews_corrcoef)
                ,
        }
        self.nested_cv_scores = []
        best_params_counts = defaultdict(int)

        inner_cv = StratifiedKFold(
                        n_splits=self.innercv,
                        shuffle=True,
                        random_state=5666)
        outer_cv = StratifiedKFold(
                        n_splits=self.outercv,
                        shuffle=True,
                        random_state=5666)

        n_splits = inner_cv.n_splits
        best_indices = []; self.results_dict = {}
        self.best_hp_list = []

        outer_cv_scores = {metric: [] for metric in
            scoring_metrics.keys()}

        #Nested CV loop
        for i, (train_idx, test_idx) in enumerate(outer_cv.
            split(X, y)):
            # Start counting time
            outer_start_time = time.time()

            X_train, X_test = X[train_idx], X[test_idx]
            y_train, y_test = y[train_idx], y[test_idx]

            if not njobs:
                inner_cv_search = GridSearchCV(
                                    estimator=pipeline,
                                    param_grid=grid_param,
                                    cv=inner_cv,
```

```
                            scoring=scoring_metrics,
                            refit='roc_auc',
                            return_train_score=True)
        else:
            inner_cv_search = GridSearchCV(
                            estimator=pipeline,
                            param_grid=grid_param,
                            cv=inner_cv,
                            scoring=scoring_metrics,
                            refit='roc_auc',
                            return_train_score=True,
                            n_jobs=-1) #use all cores
                                    #parallel backend
        percentage_done = (i+1) / outer_cv.n_splits * 100
        if trace:
            print(f"\rInner CV training & hyperparameter
                tuning on outer fold {i+1} ...")

        inner_cv_search.fit(arr2df(X_train), arr2vec(
            y_train))

        y_test_prob = inner_cv_search.predict_proba(arr2df(
            X_test))[:, 1]
        outer_cv_score = roc_auc_score(y_test, y_test_prob)

        cv_results = inner_cv_search.cv_results_
        best_indices.append(cv_results["rank_test_roc_auc"
            ].argmin())

        if trace:
            print('\n', 69*"_", '\n')
            print(f"-> Outer fold {i+1} results:\n")

        outer_fold_results = []
        for metric in scoring_metrics.keys():
            # the performance of the best model for each
                inner cross-validation fold
            test_scores = [cv_results[f"split{split}_test_{
                metric}"][
                    best_indices[-1]] for split in range(
                        inner_cv.get_n_splits())]
            outer_fold_results.append(test_scores)
            if trace:
                print(f"Inner {metric} scores: {np.array(
                    test_scores).round(4)}")
                print(f"mean-inner-{metric} : {np.array(
                    test_scores).mean():.3f}\n\n")
        # Create a dataframe for the current outer fold
        outer_fold_df = pd.DataFrame(outer_fold_results,
```

```
                        columns=[f"Inner Fold {j+1}" for
                            j in range(self.innercv)],
                        index=scoring_metrics.keys())
            metrics_mean_values = outer_fold_df.mean(axis=1)
            outer_fold_df['Mean value'] = metrics_mean_values
            self.results_dict[f"outer_fold{i + 1}"] =
                outer_fold_df

            if trace:
                print(f"\n-> Outer fold {i+1} mean roc_auc: {
                    outer_cv_score:.3f}\n")

            self.nested_cv_scores.append(outer_cv_score)

            self.best_params = inner_cv_search.best_params_
            self.best_hp_list.append(self.best_params)
            best_params_hashable = tuple(sorted(self.
                best_params.items()))
            best_params_counts[best_params_hashable] += 1

            if trace:
                print(f"Inner Fold {i+1} best hyperparameters
                    :\n\n{self.best_params}")
                print("\n\n", 16*'-',
                      f"{percentage_done:.2f}% of the procedure
                          is complete",
                      16*'-','\n')

            current_outer_fold_scores = {}
            for metric in scoring_metrics.keys():
                # Get the corresponding scorer object or string
                scorer = scoring_metrics[metric]
                ## Check if the scorer has 'needs_proba' set to
                     True
                if hasattr(scorer, '_kwargs') and scorer.
                    _kwargs.get('needs_proba', False):
                    y_test_score = y_test_prob

                else:
                    y_test_pred = inner_cv_search.predict(
                        arr2df(X_test))
                    y_test_score = y_test_pred

                 # Convert metric string to its corresponding
                     scorer object
                if isinstance(scorer, str):
                    scorer = get_scorer(scorer)

                # Calculate the outer fold score using the
                    scorer and the test data
```

```
                outer_fold_score = scorer(inner_cv_search,
                    arr2df(X_test), y_test)
                current_outer_fold_scores[metric] =
                    outer_fold_score

            for metric in scoring_metrics.keys():
                outer_cv_scores[metric].append(
                    current_outer_fold_scores[metric])

            outer_end_time = time.time()
            outer_time_elapsed = outer_end_time -
                outer_start_time
            if trace:
                mins, secs = divmod(outer_time_elapsed, 60)
                outer_time = f"{int(mins)} min. and {secs:.2f}"
                     if int(mins)!=0 else f"{secs:.2f}"
                print(f"Time taken for outer-fold-{i+1}: ",
                      outer_time, "sec.\n")

        mean_outer_cv_scores = {metric: np.mean(scores) for
            metric,
                                scores in outer_cv_scores.items
                                    ()}
        self.mean_outer_cv_scores = mean_outer_cv_scores

        #Save the name of the cls for later on...
        last_step_name = list(pipeline.named_steps.keys())[-1]
        self.classifier_name = pipeline.named_steps[
            last_step_name].__class__.__name__
        #save the pipeline, params for later on (fit final
            model)...
        self.pipe = pipeline
        self.params = grid_param
    def mean_roc_auc(self, Format: int = 6) -> float: #Nested
        CV mean roc auc
        # Calculate the average score of the nested cross-
            validation
        nested_cv_average_score = np.mean(self.nested_cv_scores
            )
        return round(nested_cv_average_score, Format)
    def inner_scores(self, outer_fold=1) -> pd.DataFrame:
        return self.results_dict["outer_fold"+str(outer_fold)]
    def best_hp(self) -> pd.DataFrame:
        # Create a dataframe for the best hyperparameters for
            each outer fold
        best_hp_df = pd.DataFrame(self.best_hp_list,
                                  columns=self.best_params.keys
                                      ())
        best_hp_df.columns = pd.MultiIndex.from_tuples(
                               [('Best Hyperparameters',
```

```
                                col) for col in best_hp_df.
                                    columns])
        best_hp_df.index = [f"Outer Fold {i+1}" for i in range(
            self.outercv)]
        best_hp_df.columns = pd.MultiIndex.from_tuples(
                                [('Best Hyperparameters',
                                col[1].split('__')[-1]) for col
                                    in best_hp_df.columns])
        return best_hp_df
    def performance(self):
        #Mean NestedCV performance for each metric score
        mean_scores_dict = self.mean_outer_cv_scores
        mean_scores_df = pd.DataFrame(mean_scores_dict,
                                index=[f'\'{self.
                                    classifier_name}\' NestedCV
                                    Performance']).T
        return mean_scores_df
    def most_frequent_hp(self) -> dict:
        best_params_counts = defaultdict(int)

        # Count the occurrences of each set of best
            hyperparameters
        for params in self.best_hp_list:
            best_params_hashable = tuple(sorted(params.items())
                )
            best_params_counts[best_params_hashable] += 1
        # Find the most frequent set of best hyperparameters
        most_frequent_best_params = max(best_params_counts, key
            =best_params_counts.get)
        most_frequent_best_params = dict(
            most_frequent_best_params)

        return most_frequent_best_params
## Fit the final model using ALL data. 'Best' is based on most
     frequent HP (on outer folds) ##
    def fit_best_model(self, X: pd.DataFrame, y: pd.DataFrame)
        -> Pipeline:
        most_frequent_best_params = self.most_frequent_hp()
        # Set the most frequent hyperparameters to the pipeline
        for param_name, param_value in
            most_frequent_best_params.items():
            self.pipe.set_params(**{param_name: param_value})
        self.pipe.fit(X, y) #fit (Usually we fit using ALL the
            data)
        return self.pipe
    def total_models_fitted(self) -> int:
        # Calculate the number of combinations in the
            param_grid
        combinations = 1
        for param_values in self.params.values():
```

```
            combinations *= len(param_values)
        # Multiply the number of combinations by the inner CV
            and outer CV loops
        total_models = combinations * self.innercv * self.
            outercv
        # Add 1 for the final model fitted with the entire
            dataset
        total_models += 1

        return total_models
```

The following figure represents each step of nested cross-validation with hyper-parameter tuning when $K = 10$.

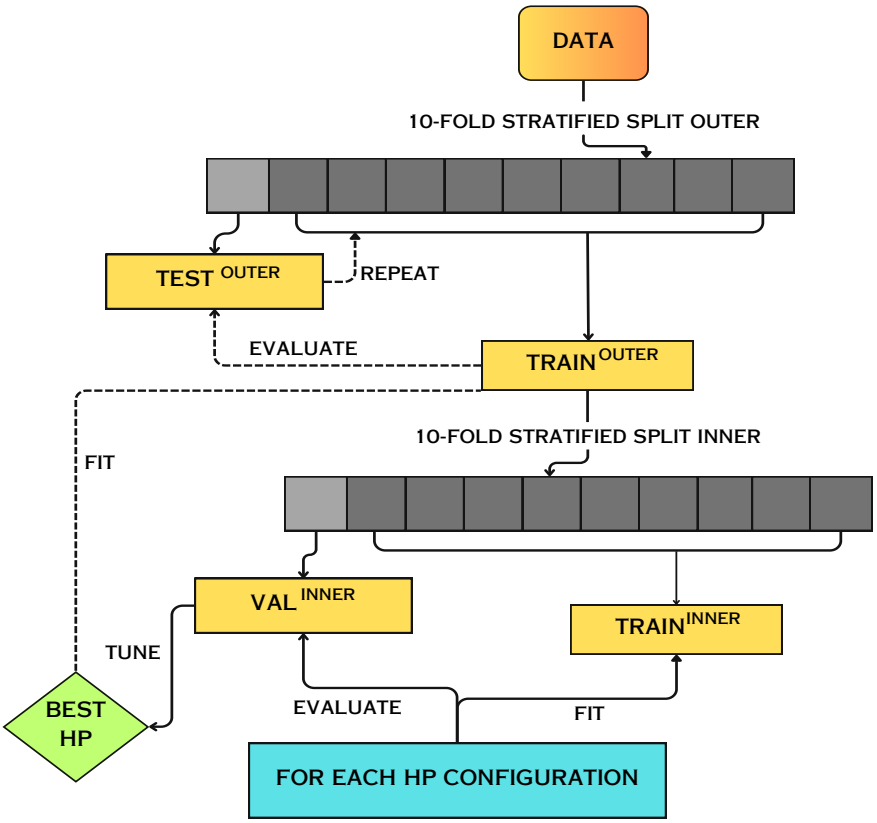


Figure 5.14: `10×10 Nested Cross-Validation`

Let's see an example using `'LogisticRegression'` from `scikit-learn`. Note

that we are using `'saga'`[2] optimizer to solve the `ElasticNet` problem. We will adjust the hyperparameter `'C'` (known as `lambda` in `R`) and the regularization parameter `l1_ratio` (known as `alpha` in `R`).

```
%%time

from sklearn.linear_model import LogisticRegression


LR_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('classifier', LogisticRegression(penalty='elasticnet',
                                      solver='saga',
                                      random_state=69,
                                      max_iter=200,
                                      n_jobs=-1))
    ])                              # n_jobs: parallel backend

LR_param_grid = {
    'classifier__C': [1/0.05, 1/0.01, 1, 1/1.5]+
                     (np.linspace(2, 50, 30)**-1).tolist(),
    'classifier__l1_ratio': np.linspace(0.1,1,10).round(1)
}

NestedCV_LR = NestedCV(innercv=10, outercv=10)

NestedCV_LR.fit(X, y, LR_pipe, LR_param_grid,
                trace=False, njobs=True)

## CPU times: user 34.1 s, sys: 2.8 s, total: 36.9 s
## Wall time: 5min 4s
```

We are using `parallel backend` to speed things up. It took around 5 minutes to fit. Let's view some results:

```
NestedCV_LR.inner_scores(outer_fold=10).round(3)
```

	Inner Fold 1	Inner Fold 2	Inner Fold 3	Inner Fold 4	Inner Fold 5	Inner Fold 6	Inner Fold 7	Inner Fold 8	Inner Fold 9	Inner Fold 10	Mean value
roc_auc	1.000	1.000	1.000	0.992	0.998	1.000	0.998	0.992	1.000	0.961	0.994
F1	1.000	0.974	0.973	0.973	0.973	0.974	0.914	0.947	0.974	0.944	0.965
F1_macro	1.000	0.979	0.979	0.979	0.979	0.979	0.935	0.958	0.979	0.957	0.972
precision	1.000	0.950	1.000	1.000	1.000	0.950	1.000	0.947	0.950	1.000	0.980
recall	1.000	1.000	0.947	0.947	0.947	1.000	0.842	0.947	1.000	0.895	0.953
average_precision	1.000	1.000	1.000	0.989	0.997	1.000	0.997	0.988	1.000	0.960	0.993
balanced_accuracy	1.000	0.985	0.974	0.974	0.974	0.984	0.921	0.958	0.984	0.947	0.970
accuracy	1.000	0.981	0.981	0.980	0.980	0.980	0.941	0.961	0.980	0.961	0.975
matthews_corrcoef	1.000	0.960	0.959	0.958	0.958	0.959	0.877	0.916	0.959	0.918	0.947

```
#Mean Outer Performances for each metric
NestedCV_LR.performance()
```

[2] Stochastic Average Gradient Descent with Adaptive learning rate

'LogisticRegression' NestedCV Performance	
roc_auc	0.994444
F1	0.968138
F1_macro	0.975180
precision	0.985627
recall	0.952381
average_precision	0.993430
balanced_accuracy	0.971984
accuracy	0.977193
matthews_corrcoef	0.951410

```
#Best tuned HP for each outer-fold
NestedCV_LR.best_hp()
```

	Best Hyperparameters	
	C	l1_ratio
Outer Fold 1	1.	0.1
Outer Fold 2	1.	0.1
Outer Fold 3	1.	0.2
Outer Fold 4	0.143564	0.1
Outer Fold 5	0.5	0.9
Outer Fold 6	0.5	0.7
Outer Fold 7	0.143564	0.1
Outer Fold 8	0.273585	0.1
Outer Fold 9	0.097315	0.2
Outer Fold 10	1.	0.1

We can see that the results are similar to those obtained using `nestedcv` in `R`. What is the total number of models that underwent training?

```
NestedCV_LR.total_models_fitted()
```

```
34001
```

The following method returns the final model fitted using ALL the data:

```
NestedCV_LR.fit_best_model(X,y)
```

5.7.3 Nested Cross-Validation for Random Forest Classifier

We will now use the `'RandomForestClassifier'` object from the `scikit-learn` library. The following hyperparameters will be tuned to optimize the model's performance:

- `n_estimators`: The number of trees in the forest. It is an integer value representing how many individual decision trees will be built within the random forest. A larger number of trees typically leads to better performance but also increases computational complexity and can lead to overfitting if not controlled properly.
- `criterion`: The function used to measure the quality of a split. It can be either `'gini'` for `Gini impurity` or `'entropy'` for `information gain`. Both are used to determine how well a feature separates the data into different classes, with lower values indicating better splits.
- `min_samples_split`: The minimum number of samples required to split an internal node. It can help control overfitting by preventing the tree from growing too deep. Larger values will result in shallower trees and more generalization.
- `min_samples_leaf`: The minimum number of samples required to be at a leaf node. This parameter ensures that a leaf node has a minimum number of samples, which can help control overfitting and improve generalization.
- `max_features`: The number of features to consider when looking for the best split. It can be either `'sqrt'` (square root of the total number of features) or `'log2'` (logarithm base 2 of the total number of features). This parameter adds randomness to the process of selecting features, which can reduce overfitting and improve generalization.

```
%%time

### nestedcv-RF using ALL predictors ###

RF_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(bootstrap=True,
                                          random_state=69))
    ])

RF_grid_param = {
    'classifier__n_estimators': (100, 150, 250, 450),
    'classifier__criterion': ('gini', 'entropy'),
    'classifier__min_samples_split': (2, 3, 5),
    'classifier__min_samples_leaf': (1, 2, 4),
    'classifier__max_features': ('sqrt', 'log2')
```

```
    }

"""
Including additional hyperparameters results in a substantial
    increase
in the time required to fit each combination. Thats why one
    could use
RandomizedSearchCV' instead of GridSearchCV."""

NestedCV_RF_full = NestedCV(innercv=10, outercv=10)

NestedCV_RF_full.fit(X, y, RF_pipe, RF_grid_param,
                     trace=False, njobs=True)

## CPU times: user 28.5 s, sys: 2.84 s, total: 31.3 s
## Wall time: 17min 5s

## Results ##

NestedCV_RF_full.performance()
```

'RandomForestClassifier' NestedCV Performance	
roc_auc	0.990975
F1	0.944181
F1_macro	0.956195
precision	0.961739
recall	0.928788
average_precision	0.988124
balanced_accuracy	0.953203
accuracy	0.959524
matthews_corrcoef	0.913697

```
NestedCV_RF_full.inner_scores(outer_fold=1).round(3)
```

	Inner Fold 1	Inner Fold 2	Inner Fold 3	Inner Fold 4	Inner Fold 5	Inner Fold 6	Inner Fold 7	Inner Fold 8	Inner Fold 9	Inner Fold 10	Mean value
roc_auc	0.979	0.997	0.977	1.000	0.998	0.990	1.000	0.998	0.997	0.979	0.992
F1	0.857	0.974	0.973	0.974	0.974	0.889	0.944	0.973	0.923	0.919	0.940
F1_macro	0.892	0.979	0.979	0.979	0.979	0.914	0.957	0.979	0.938	0.936	0.953
precision	0.938	0.950	1.000	0.950	0.950	0.941	1.000	1.000	0.900	0.944	0.957
recall	0.789	1.000	0.947	1.000	1.000	0.842	0.895	0.947	0.947	0.895	0.926
average_precision	0.968	0.995	0.978	1.000	0.997	0.984	1.000	0.997	0.995	0.974	0.989
balanced_accuracy	0.880	0.985	0.974	0.984	0.984	0.905	0.947	0.974	0.942	0.932	0.951
accuracy	0.904	0.981	0.980	0.980	0.980	0.922	0.961	0.980	0.941	0.941	0.957
matthews_corrcoef	0.792	0.960	0.958	0.959	0.959	0.832	0.918	0.958	0.876	0.874	0.909

```
### Best HP (hyperparameters) ###
NestedCV_RF_full.best_hp()
```

	Best Hyperparameters				
	criterion	max_features	min_samples_leaf	min_samples_split	n_estimators
Outer Fold 1	entropy	log2	2	5	250
Outer Fold 2	entropy	log2	1	5	100
Outer Fold 3	entropy	log2	1	2	150
Outer Fold 4	entropy	log2	2	2	100
Outer Fold 5	entropy	sqrt	2	5	250
Outer Fold 6	entropy	log2	2	5	150
Outer Fold 7	entropy	log2	1	5	150
Outer Fold 8	entropy	sqrt	2	5	100
Outer Fold 9	entropy	log2	1	3	150
Outer Fold 10	entropy	log2	1	5	450

```
NestedCV_RF_full.total_models_fitted()
```

```
14401
```

Based on the above results, it seems that Logistic Regression outperforms Random Forest when using all predictors.

Let's investigate the performance of Random Forest in conjunction with the `stepAICc()` method. Keep in mind that `stepAICc` should be incorporated within cross-validation and tuned as a hyperparameter, taking values such as `'forward'`, `'backward'`, or `'both'`. First, we need to implement the `stepAICc()` method in Python. To achieve this, we can leverage the `rpy2` package. Here's how: First, create an R file named `get_stepAICc.R` and store the stepwise selection algorithm within it. Modify the algorithm to return the vector `'names(stepAICc.direction$model$coeff)[-1]'`, which represents the selected predictors.

```
import pandas as pd
import numpy as np
import rpy2
import rpy2.robjects as robjects
import warnings
import time
warnings.filterwarnings('ignore')

## re-write the data in R form
data = pd.read_csv("data.csv").drop(['id', 'Unnamed: 32'],
                                    axis='columns')
data['diagnosis'] = data['diagnosis'].map({'M':1, 'B':0})
data = data.rename(columns={'diagnosis': 'y'})

#Python ignores 'dots' (we rename them in order to use stepAICc
    from R)
data = data.rename(
       columns={
```

```
            'concave points_worst': 'concave.points_worst',
            'concave points_se': 'concave.points_se',
            'concave points_mean': 'concave.points_mean'})
X = data.loc[:, data.columns != 'y']
y = data[['y']]

# Pandas ---> R data.frame

from rpy2.robjects import pandas2ri
import rpy2.robjects as ro

# Activate the automatic conversion
# between pandas and R data frames
pandas2ri.activate()

# Convert the pandas DataFrame to an R data.frame
r_data_frame = pandas2ri.py2rpy(data)

# Check the R data.frame
#print(r_data_frame)

# If you want to use the R data.frame in R functions
ro.r.assign('R_DF', r_data_frame)

#Get access to StepAICc from R
r = robjects.r
source = r['source']("get_stepAICc.R")

### To get forward selection for example do this:
### stepAICc_forward = list(r['stepAICc_coef'](ro.r('R_DF'),
###                                            direction='forward'))
```

Now, to utilize `stepAICc` as a feature selection algorithm in Python, we need to convert it to a format compatible with scikit-learn. To accomplish this, we create the following class:

```
from sklearn.base import BaseEstimator, TransformerMixin

pandas2ri.activate()

class StepAICc(BaseEstimator, TransformerMixin):

    def __init__(self, direction: str = "backward"):
        self.direction = direction

    def fit(self, X: pd.DataFrame, y: pd.DataFrame):
        data = pd.concat([X, y], axis=1)

        r_data_frame = pandas2ri.py2rpy(data)
        # If you want to use the R data.frame in R functions
        ro.r.assign('R_DF', r_data_frame)
```

```
        self.feature_indices_ = list(
                          r['stepAICc_coef'](ro.r('R_DF'),
                          direction=self.direction)
                          )
        return self

    def transform(self, X):
        self.featuresDF = X[self.feature_indices_]
        return self.featuresDF

    def get_feature_names_out(self):
        return self.featuresDF.columns.tolist()
```

This class effectively executes the stepwise selection from `R` while adhering to the `scikit-learn` format.

```
%%time

### Use StepAICc 'filter' inside the cross-validation ###

RF_stepAICc_pipe = Pipeline([
    ('StepAICc', StepAICc()),
    ('Scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(bootstrap=True,
                                          criterion='entropy',
                                          random_state=69))
    ])

RF_stepAICc_grid_param = {
    'StepAICc__direction': ('forward', 'backward', 'both'),
    'classifier__n_estimators': (100, 250),
    'classifier__min_samples_split': (2, 5),
    'classifier__min_samples_leaf': (1, 3),
    'classifier__max_features': ('sqrt', 'log2')
    }

NestedCV_RF_stepAICc = NestedCV(innercv=10, outercv=10)

NestedCV_RF_stepAICc.fit(X, y,
           RF_stepAICc_pipe,
           RF_stepAICc_grid_param)

### Wall time 10h 36min
```

Results:

```
NestedCV_RF_stepAICc.performance()
```

'RandomForestClassifier' NestedCV Performance	
roc_auc	0.985
F1	0.944
F1_macro	0.956
precision	0.965
recall	0.924
average_precision	0.983
balanced_accuracy	0.952
accuracy	0.960
matthews_corrcoef	0.913

```
NestedCV_RF_stepAICc.best_hp()
```

Best Hyperparameters

	direction	max_features	min_samples_leaf	min_samples_split	n_estimators
Outer Fold 1	forward	sqrt	1	5	250
Outer Fold 2	backward	sqrt	1	5	100
Outer Fold 3	forward	sqrt	1	5	100
Outer Fold 4	forward	sqrt	1	2	100
Outer Fold 5	forward	sqrt	1	2	250
Outer Fold 6	backward	sqrt	1	2	250
Outer Fold 7	forward	sqrt	1	5	100
Outer Fold 8	forward	sqrt	1	2	250
Outer Fold 9	backward	sqrt	1	2	100
Outer Fold 10	both	sqrt	1	5	250

In reality, Random Forest inherently performs feature selection through its algorithm, which builds multiple decision trees on various subsets of features. Therefore, it doesn't require an additional feature selection process. However, in the upcoming section, we will utilize `stepAICc` for linear and logistic regression models.

5.7.4 Gradient Boosting Machines (GBM)

Gradient Boosting Machines (GBMs) are a powerful ensemble learning technique used for both regression and classification tasks. They combine the predictive power of multiple weak learners, usually decision trees, to create a more robust and accurate model. The general idea behind gradient boosting is to iteratively add new models to the ensemble while minimizing a specified loss function. Here's an overview of how Gradient Boosting Machines work:

1. Initialize the model with a constant value or a weak learner, typically a shallow decision tree.

2. Calculate the residuals (differences between the actual and predicted values) for each data point.

3. Train a new weak learner (decision tree) to predict the residuals, focusing more on the instances with larger residuals by assigning higher weights.

4. Combine the new weak learner with the existing ensemble by optimizing the coefficients to minimize the overall loss function.

5. Repeat steps 2-4 for a specified number of iterations or until a stopping criterion is met, such as minimal improvement in the loss function or reaching a maximum number of trees.

GBMs can be more preferable than single decision trees or random forests for several reasons:

- ◇ `Improved accuracy`: By combining multiple weak learners, GBMs can capture complex patterns in the data and generally provide better predictive performance than single decision trees.
- ◇ `Flexibility`: GBMs can be applied to a wide range of tasks, such as binary/multiclass classification, regression and ranking problems. They can also be adapted to various loss functions and customized to specific problem domains.
- ◇ `Robustness` to overfitting: Gradient boosting employs regularization techniques such as shrinkage (learning rate) and early stopping to prevent overfitting, leading to better generalization performance compared to single decision trees.
- ◇ `Feature importance`: GBMs can provide a ranking of the importance of input features, which can be valuable for feature selection, interpretation and understanding the model's behavior.

Gradient Boosting Machines have some disadvantages when compared to random forests. They usually take longer to train because they train sequentially, while random forests can train trees simultaneously. GBMs also involve more hyperparameters, such as `learning rate`, `tree depth` and a `loss function` to be optimized during gradient boosting[3], which can make finding the best combination for optimal performance more challenging. Additionally, GBMs can be sensitive to noise and outliers, potentially impacting the model's performance negatively. However, preprocessing the data and using techniques like robust loss functions or outlier detection can help address these issues.

5.7.5 Adaptive Boosting (`AdaBoost`)

Adaptive Boosting, or `AdaBoost`, is a popular ensemble learning technique used for both classification and regression tasks. It was first introduced by Yoav Freund and Robert Schapire in 1996. The main idea behind `AdaBoost` is to combine the predictions of multiple weak learners, usually decision trees or

[3]For classification tasks, the options are `'deviance'` (default) for logistic regression loss and `'exponential'` for exponential loss

other simple models, to create a more accurate and robust model. The algorithm adjusts the weights of the training instances, focusing more on the difficult-to-classify instances in each iteration. here's how `AdaBoost` works for classification tasks:

1. Initialize the weights of the training instances, typically assigning equal weights to all instances.
2. Train a weak learner (e.g., a shallow decision tree) on the weighted training data.
3. Compute the training error for the weak learner by comparing its predictions with the actual target values, taking into account the instance weights.
4. Calculate the importance (alpha) of the weak learner, which depends on its training error. A smaller error leads to a larger alpha, meaning the weak learner has a higher contribution to the final ensemble.
5. Update the instance weights by increasing the weights of misclassified instances and decreasing the weights of correctly classified instances. This encourages the next weak learner to focus more on the misclassified instances.
6. Normalize the updated weights to ensure they sum up to one.
7. Repeat steps 2-6 for a specified number of iterations or until a stopping criterion is met, such as no improvement in the training error or reaching a maximum number of weak learners.
8. Combine the weak learners by taking a weighted majority vote (for binary classification) or a weighted sum (for regression) based on their importance.

The seminal papers by Friedman [17] and Freund & Schapire [14] have significantly shaped the landscape of ensemble learning. Friedman's paper introduced the Gradient Boosting Machines algorithm, providing a powerful and flexible method for combining weak learners to improve prediction accuracy, while Freund Schapire's paper laid the foundation for `AdaBoost`, a robust and efficient boosting algorithm that has become a cornerstone of machine learning and has inspired many subsequent advancements in ensemble learning.

5.7.6 Nested Cross-Validation for `GBM`, `AdaBoost`

Let's compare the performance of `GBMs`, `AdaBoost` and `Random Forest`. To obtain an unbiased estimation of these models, we will use the `NestedCV` class.

```
%%time
# Gradient Boosting Machines (GBM)
from sklearn.ensemble import GradientBoostingClassifier
GB_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('classifier', GradientBoostingClassifier(learning_rate
       =0.01,
                                               random_state
                                                  =5666))
    ])
param_grid = {
    'classifier__loss': ['deviance', 'exponential'],
    'classifier__learning_rate': [0.001, 0.01],
    'classifier__n_estimators': [5000, 10000, 20000],
    'classifier__loss': ['deviance', 'exponential'],
    'classifier__min_samples_split': [2, 5],
    'classifier__min_samples_leaf': [1, 2],
    'classifier__max_features': ['sqrt', 'log2'],
    'classifier__subsample': [0.5, 0.75, 1.0],
}
NestedCV_GB = NestedCV(innercv=10, outercv=10)

NestedCV_GB.fit(X, y, GB_pipe,
                param_grid, njobs=True, trace=False)
### CPU times: user 1min 44s, sys: 21.6 s, total: 2min 6s
### Wall time: 14h 36min 45s
```

The code provided below pertains to the `AdaBoost` classifier.

```
%%time
# AdaBoost
from sklearn.ensemble import AdaBoostClassifier

AB_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('classifier', AdaBoostClassifier(random_state=5666))
    ])
param_grid = {
    'classifier__n_estimators': [5000, 10000, 15000],
    'classifier__algorithm': ['SAMME', 'SAMME.R'],
    'classifier__learning_rate ': [0.005 , 0.01]
}
NestedCV_AB = NestedCV(innercv=10, outercv=10)

NestedCV_AB.fit(X, y, AB_pipe, param_grid,
                njobs=True, trace=False)
### CPU times: user 3min 8s, sys: 1.75 s, total: 3min 10s
### Wall time: 28min 46s
```

Here's an overview of the `NestedCV` results:

```
pd.concat([NestedCV_AB.performance(),
```

```
            NestedCV_GB.performance()],
            axis=1).round(3)
```

	AdaBoostClassifier NestedCV Performance	GradientBoostingClassifier NestedCV Performance
roc_auc	0.993	0.992
F1	0.966	0.944
F1_macro	0.973	0.956
precision	0.986	0.966
recall	0.948	0.924
average_precision	0.991	0.989
balanced_accuracy	0.970	0.952
accuracy	0.975	0.960
matthews_corrcoef	0.948	0.914

```
NestedCV_GB.best_hp()
```

	Best Hyperparameters						
	learning_rate	loss	max_features	min_samples_leaf	min_samples_split	n_estimators	subsample
Outer Fold 1	0.01	deviance	sqrt	2	5	10000	0.75
Outer Fold 2	0.01	deviance	log2	1	5	5000	0.50
Outer Fold 3	0.01	deviance	log2	2	2	10000	0.50
Outer Fold 4	0.01	deviance	sqrt	1	5	10000	0.50
Outer Fold 5	0.01	deviance	log2	2	5	10000	0.75
Outer Fold 6	0.01	deviance	sqrt	2	5	10000	0.75
Outer Fold 7	0.01	deviance	sqrt	1	2	5000	0.50
Outer Fold 8	0.01	exponential	sqrt	2	5	10000	1.00
Outer Fold 9	0.01	deviance	log2	2	2	10000	0.75
Outer Fold 10	0.01	deviance	log2	2	2	5000	1.00

```
NestedCV_AB.best_hp()
```

	Best Hyperparameters		
	algorithm	learning_rate	n_estimators
Outer Fold 1	SAMME.R	0.01	5000
Outer Fold 2	SAMME.R	0.01	5000
Outer Fold 3	SAMME.R	0.01	5000
Outer Fold 4	SAMME.R	0.01	5000
Outer Fold 5	SAMME.R	0.01	5000
Outer Fold 6	SAMME.R	0.01	15000
Outer Fold 7	SAMME.R	0.01	15000
Outer Fold 8	SAMME.R	0.01	5000
Outer Fold 9	SAMME.R	0.01	5000
Outer Fold 10	SAMME.R	0.01	5000

`SAMME.R` (Stagewise Additive Modeling using a Multiclass Exponential loss function with Real-valued predictions) is an extension of the AdaBoost algorithm for multi-class classification problems. The algorithm was introduced by J. Zhu, H. Zou, S. Rosset and T. Hastie in their 2009 paper [19]

The original AdaBoost algorithm was designed for binary classification tasks, where there are only two classes. In multi-class problems, there are more than two classes to predict. SAMME.R extends AdaBoost to handle such scenarios by using class probabilities instead of binary predictions when calculating the weight updates.

The main idea behind SAMME.R is to compute the probability estimates of each class for a given instance by considering the weighted votes of all the weak classifiers in the ensemble. Then, the algorithm uses these probability estimates to update the instance weights and calculate the contribution of the current weak classifier.

5.8 Nested Cross-Validation with StepAICc Selection for glmnet & hqreg Models using Python

In this section, we will use the 'NestedCV' class for nested cross-validation along with stepwise selection (based on AICc) in glmnet and hqreg models. In order to use the cv.glmnet() or cv.hqreg() models from R, we will need the rpy2 package and create an R function in Python that takes training data, prediction data and returns the predictions in that dataset. Since our NestedCV method only works with scikit-learn classifiers, we have to modify the cv.glmnet() and the cv.hqreg() models accordingly. Let's see how to do that.

5.8.1 NestedCV for cv.glmnet()

```
from rpy2.robjects import r, pandas2ri
from rpy2.robjects.packages import importr
from sklearn.base import BaseEstimator, ClassifierMixin

pandas2ri.activate()
glmnet = importr("glmnet")

class CVGlmnetClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, family="gaussian", alpha=0.5,
        random_state=False):
        self.family = family
        self.alpha = alpha
        self.random_state = random_state

    def fit(self, X, y):
        if isinstance(X, pd.DataFrame):
            X = np.array(X)
        if isinstance(y, pd.DataFrame):
```

```
            y = np.ravel(y)

        if isinstance(X, np.ndarray):
            X = pd.DataFrame(X)
        if isinstance(y, np.ndarray):
            y = pd.Series(y)

        self.X_ = X
        self.y_ = y
        self.classes_ = np.unique(y)

        return self

    def predict_proba(self, X):
        np.random.seed(self.random_state)

        if isinstance(X, np.ndarray):
            X = pd.DataFrame(X)

        ### R code for cv.glmnet() ###
        r_cv_glmnet = r(f'''
            cv_glmnet <- function (X, y, family,
                                   alpha, newx, random_state) {{

                if (family == "binomial") {{
                    set.seed(random_state)
                    model <- cv.glmnet(as.matrix(X),
                                       as.numeric(y),
                                       alpha=alpha,
                                       family="binomial",
                                       type.measure="auc",
                                       nfolds=10,
                                       parallel=TRUE)
                }} else if (family == "gaussian") {{
                    set.seed(random_state)
                    model <- cv.glmnet(as.matrix(X),
                                       as.numeric(y),
                                       alpha=alpha,
                                       family="gaussian",
                                       nfolds=10,
                                       parallel=TRUE)
                }}
                return (as.numeric(predict(model,
                                           newx=as.matrix(newx)
                                               ,
                                           type="response")))
            }}
        ''')

        y_pred = np.array(
```

```
            r_cv_glmnet(
                X=self.X_,
                y=self.y_,
                family=self.family,
                alpha=self.alpha,
                newx=X,
                random_state=self.random_state))

        y_pred = np.array(list(
                    map(lambda x: [abs(1-x), x], y_pred)))

        return y_pred

    def predict(self, X):
        binary_pred = np.array(list(
                        map(lambda x: 1 if x>=0.5 else 0,
                        self.predict_proba(X)[:, 1])))
        return binary_pred

    def score(self, X, y, metric='roc_auc'):
        # Create a dictionary of available metrics
        metrics = {
            'f1': f1_score,
            'roc_auc': roc_auc_score,
            'recall': recall_score,
            'precision': precision_score,
            'balanced_accuracy': balanced_accuracy_score
        }

        # Check if the specified metric is valid
        if metric not in metrics:
            raise ValueError(f"Invalid metric '{metric}',
                choose from {list(metrics.keys())}")

        # Calculate and return the score using the specified
            metric
        y_pred = self.predict(X)
        if metric=='roc_auc':
            y_pred = self.predict_proba(X)[:, 1]
        score_func = metrics[metric]

        return score_func(y, y_pred)
```

The above class is an implementation of `cv.glmnet`, but with the format of `scikit-learn`. Let's now fit this class within the `NestedCV` one. Here's an example of full model `gaussian glmnet`.

```
%%time
cvglmnet_gaussian_pipe_full = Pipeline([
    ('Scaler', StandardScaler()),
    ('classifier', CVGlmnetClassifier(family='gaussian',
```

```
                                        random_state=5666))
    ])
param_grid_glmnet_full = {
     #We optimize 'lambda' on cv.glmnet, we only need to opt.
         alhpa
    'classifier__alpha': (0, 1) #Ridge Lasso
    }
NestedCV_cvglmnet_gaussian_full = NestedCV(innercv=10, outercv
    =10)

NestedCV_cvglmnet_gaussian_full.fit(X, y,
           cvglmnet_gaussian_pipe_full,
           param_grid_glmnet_full,
           njobs=True, trace=False)
## CPU times: user 17.6 s, sys: 858 ms, total: 18.4 s
## Wall time: 25.2 s
```

Ridge regularization seems to be the best choice here.

```
NestedCV_cvglmnet_gaussian_full.performance()
```

'CVGlmnetClassifier' NestedCV Performance	
roc_auc	0.991931
F1	0.930079
F1_macro	0.947062
precision	0.999999
recall	0.871861
average_precision	0.990571
balanced_accuracy	0.935931
accuracy	0.952569
matthews_corrcoef	0.900797

Let's now see how it performs with `stepAICc()` filter:

```
%%time
cvglmnet_binomial_pipe = Pipeline([
    ('StepAICc', StepAICc()),
    ('Scaler', StandardScaler()),
    ('classifier', CVGlmnetClassifier(family='binomial',
                                        random_state=5666))
    ])
param_grid_glmnet = {
     #We optimize 'lambda' on cv.glmnet, we only need to opt.
         alhpa
    'classifier__alpha': (0, 0.3, 0.5, 0.7, 1),
    'StepAICc__direction': ('both', 'forward', 'backward')
    }
NestedCV_cvglmnet_binomial = NestedCV(innercv=10, outercv=10)
NestedCV_cvglmnet_binomial.fit(X, y,
           cvglmnet_binomial_pipe,
```

```
            param_grid_glmnet, trace=False)
## CPU times: user 15h 44min 8s, sys: 2h 26min 49s, total: 18h
    10min 58s
## Wall time: 4h 38min 59s
%%time
cvglmnet_gaussian_pipe = Pipeline([
    ('StepAICc', StepAICc()),
    ('Scaler', StandardScaler()),
    ('classifier', CVGlmnetClassifier(family='gaussian',
                                      random_state=5666))
    ])
NestedCV_cvglmnet_gaussian = NestedCV(innercv=10, outercv=10)
NestedCV_cvglmnet_gaussian.fit(X, y, cvglmnet_gaussian_pipe,
                               param_grid_glmnet, trace=False)
## CPU times: user 15h 4min 28s, sys: 2h 27min 9s, total: 17h
    31min 38s
## Wall time: 3h 27min 46s
```

Results:

```
pd.concat([NestedCV_cvglmnet_gaussian.best_hp(),
           NestedCV_cvglmnet_binomial.best_hp()], axis=1)
```

	Best HP (Gaussian)		Best HP (Binomial)	
	direction	alpha	direction	alpha
Outer Fold 1	backward	0	forward	1
Outer Fold 2	both	0.3	backward	0.3
Outer Fold 3	backward	0	backward	0.5
Outer Fold 4	both	0.3	both	1
Outer Fold 5	backward	0	backward	0.5
Outer Fold 6	backward	0.3	both	0.3
Outer Fold 7	forward	0	forward	0.3
Outer Fold 8	backward	0	forward	0.5
Outer Fold 9	both	0.7	backward	0.3
Outer Fold 10	backward	0	both	0.3

```
pd.concat([NestedCV_cvglmnet_gaussian.performance().round(3),
           NestedCV_cvglmnet_binomial.performance().round(3)],
           axis=1)
```

	CVGlmnetClassifier (Gaussian) NestedCV Perf.	CVGlmnetClassifier (Binomial) NestedCV Perf.
roc_auc	0.987	0.99
F1	0.925	0.953
F1_macro	0.943	0.964
precision	0.994	0.977
recall	0.867	0.933
average_precision	0.987	0.99
balanced_accuracy	0.932	0.96
accuracy	0.949	0.967
matthews_corrcoef	0.893	0.929

5.8.2 NestedCV for cv.hqreg()

In this subsection, we will explore the implementation of least absolute deviations (LAD) using nested cross-validation. First, we'll create a Python class for the `hqreg` package, which performs quantile regression (LAD) in `R`. Afterward, we will convert this class into the `scikit-learn` format. Next, we will fit a complete model to the `NestedCV` class and evaluate its performance. Then, we will repeat this process using a `stepAICc()` filter within the cross-validation to see how it improves the performance of the model.

```
from rpy2.robjects import r, pandas2ri
from rpy2.robjects.packages import importr
from sklearn.base import BaseEstimator, ClassifierMixin

pandas2ri.activate()
hqreg = importr("hqreg")

class CVhqregClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, alpha=0.5, random_state=False):
        self.alpha = alpha
        self.random_state = random_state

    def fit(self, X, y):
        if isinstance(X, pd.DataFrame):
            X = np.array(X)
        if isinstance(y, pd.DataFrame):
            y = np.ravel(y)
        if isinstance(X, np.ndarray):
            X = pd.DataFrame(X)
        if isinstance(y, np.ndarray):
            y = pd.Series(y)
        self.X_ = X
        self.y_ = y
        self.classes_ = np.unique(y)
        return self

    def predict_proba(self, X):
```

```
        np.random.seed(self.random_state)
        if isinstance(X, np.ndarray):
            X = pd.DataFrame(X)
        ### R code for cv.hqreg() ###
        r_cv_hqreg = r('''
            cv_hqreg <- function (X, y, alpha=0.5,  newx,
                random_state) {
                quiet <- function(x) {
                             sink(tempfile())
                             on.exit(sink())
                             invisible(force(x))
                           }  #Hides the CV messeges.
                set.seed(random_state)
                model <-  quiet(cv.hqreg(
                            X=as.matrix(X),
                            y=as.numeric(y),
                            nfolds=10, #10fold CV
                            alpha=alpha,
                            method='quantile', #LAD loss
                            seed=5666))
                return (as.numeric(predict(model,
                                           as.matrix(newx))))
                }
        ''')
        y_pred = np.array(
            r_cv_hqreg(
                X=self.X_,
                y=self.y_,
                alpha=self.alpha,
                newx=X,
                random_state=self.random_state))
        y_pred = np.array(list(
                    map(lambda x: [abs(1-x), x], y_pred)))
        return y_pred

    def predict(self, X):
        binary_pred = np.array(list(
                        map(lambda x: 1 if x>=0.5 else 0,
                        self.predict_proba(X)[:, 1])))
        return binary_pred

    def score(self, X, y, metric='roc_auc'):
        # Create a dictionary of available metrics
        metrics = {
            'f1': f1_score,
            'roc_auc': roc_auc_score,
            'recall': recall_score,
            'precision': precision_score,
            'balanced_accuracy': balanced_accuracy_score
        }
```

```
        # Check if the specified metric is valid
        if metric not in metrics:
            raise ValueError(f"Invalid metric '{metric}',
                choose from {list(metrics.keys())}")
        # Calculate and return the score using the specified
            metric
        y_pred = self.predict(X)
        if metric=='roc_auc':
            y_pred = self.predict_proba(X)[:, 1]
        score_func = metrics[metric]

        return score_func(y, y_pred)
```

The above class can now perform quantile regression (from a classification standpoint) and it follows the structure of `scikit-learn` classifier. Let's see how `NestedCV` performs using all predictors (no filtering). We are going to tune the `alpha` hyperparameter for the `ElasticNet` regularization. Note that `cv.hqreg()` automatically performs `10-fold CV` and tunes the `lambda` hyperparameter.

```
## Full model LAD ##

LAD_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('classifier', CVhqregClassifier(random_state=5666))
    ])

param_grid_LAD = {
    'classifier__alpha': (0, 0.1, 0.5, 0.7, 1),
    }

NestedCV_LAD = NestedCV(innercv=10, outercv=10)


NestedCV_LAD.fit(X, y,
                 LAD_pipe,
                 param_grid_LAD,
                 trace=False)
```

`NestedCV` with `StepAICc()` selection:

```
## LAD with StepAICc() inside the CV ##

LAD_stepAICc_pipe = Pipeline([
    ('StepAICc', StepAICc()),
    ('Scaler', StandardScaler()),
    ('classifier', CVhqregClassifier(random_state=5666))
    ])

param_grid_LAD_stepAICc = {
    'classifier__alpha': (0, 0.1, 0.5, 0.7, 1),
```

```
    'StepAICc__direction': ('both', 'forward', 'backward')
    }

NestedCV_LAD_stepAIcc = NestedCV(innercv=10, outercv=10)


NestedCV_LAD_stepAIcc.fit(X, y,
                          LAD_stepAICc_pipe,
                          param_grid_LAD_stepAICc,
                          trace=False)
```

The results of the nested cross-validation for the penalized LAD, both with and without the `stepAICc` filtering, are presented below.

```
pd.concat([NestedCV_LAD.performance(),
           NestedCV_LAD_stepAIcc.performance()], axis=1)
```

	`CVhqregClassifier` (`no filter`) NestedCV Perf.	`CVhqregClassifier` (`StepAICc`) NestedCV Perf.
`roc_auc`	0.99074	0.98611
`F1`	0.89362	0.89178
`F1_macro`	0.92054	0.91982
`precision`	0.99999	0.99999
`recall`	0.81082	0.81017
`average_precision`	0.98911	0.98592
`balanced_accuracy`	0.90541	0.90509
`accuracy`	0.92976	0.92976
`matthews_corrcoef`	0.85430	0.85426

```
pd.concat([NestedCV_LAD.best_hp(),
           NestedCV_LAD_stepAIcc.best_hp()], axis=1)
```

	Best `HP` (`no filter`)	Best `HP` (`StepAICc filter`)	
	`alpha`	`direction`	`alpha`
`Outer Fold 1`	0	`backward`	0
`Outer Fold 2`	0	`both`	0.7
`Outer Fold 3`	0	`backward`	0
`Outer Fold 4`	0	`both`	0.1
`Outer Fold 5`	0	`backward`	0.1
`Outer Fold 6`	0.1	`forward`	1
`Outer Fold 7`	0	`forward`	0.1
`Outer Fold 8`	0	`backward`	0
`Outer Fold 9`	0	`both`	0.1
`Outer Fold 10`	0	`backward`	0

We will now repeat the same procedure of nested cross-validation, but this time for VIF-filtered predictors and the second-order-terms (filtered with `stepAIC`). The following Python code specifically pertains to the VIF-filtered variables.

```
VIF_X = [
  'texture_mean', 'smoothness_mean', 'symmetry_mean',
  'fractal_dimension_mean', 'texture_se', 'smoothness_se',
  'concavity_se', 'concave.points_se', 'symmetry_se',
  'fractal_dimension_se', 'smoothness_worst', 'symmetry_worst']
X_VIF = X[VIF_X]

%%time
## VIF model nestedcv LAD ##

LAD_pipe = Pipeline([
    ('Scaler', StandardScaler()),
    ('classifier', CVhqregClassifier(random_state=5666))
    ])

param_grid_LAD = {
    'classifier__alpha': (0, 0.1, 0.5, 0.7, 1),
    }

NestedCV_LAD_VIF = NestedCV(innercv=10, outercv=10)

NestedCV_LAD_VIF.fit(X_VIF, y,
                 LAD_pipe,
                 param_grid_LAD,
                 njobs=True)
### CPU times: user 5min 33s, sys: 2.16 s, total: 5min 35s
### Wall time: 5min 39s
```

As before, we perform nested cross-validation with hyperparameter tuning, this time focusing on the second-order terms. The following code corresponds to this process.

```
#Select the 2o terms from stepwise selection in R
second_order = [
    'compactness_mean', 'concavity_mean',
    'fractal_dimension_mean', 'radius_se',
    'compactness_se', 'texture_worst',
    'area_worst', 'concave.points_worst',
    'symmetry_worst', 'compactness_mean:compactness_se',
    'concavity_mean:fractal_dimension_mean']

sord1 = np.ravel(X['compactness_mean']*X['compactness_se'])
sord2 = np.ravel(X['concavity_mean']*X['fractal_dimension_mean'
                                       ])
X_2o = X[second_order[:-2]]
X_2o[second_order[-2]] = sord1
X_2o[second_order[-1]] = sord2

%%time
## stepAIC sec. ord. model nestedCV LAD ##
LAD_pipe = Pipeline([
```

```
    ('Scaler', StandardScaler()),
    ('classifier', CVhqregClassifier(random_state=5666))
    ])
param_grid_LAD = {
    'classifier__alpha': (0, 0.1, 0.5, 0.7, 1),
    }
NestedCV_LAD_2o = NestedCV(innercv=10, outercv=10)
NestedCV_LAD_2o.fit(X_2o, y,
                    LAD_pipe,
                    param_grid_LAD,
                    njobs=True)
### CPU times: user 2min 28s, sys: 1.23 s, total: 2min 29s
### Wall time: 2min 32s
```

Ridge regularization (α=0) was chosen as the optimal hyperparameter for each outer fold in both nested CV objects (VIF and second-order). Let's examine the average performance of both models based on nested CV.

```
pd.concat([NestedCV_LAD_VIF.performance(),
           NestedCV_LAD_2o.performance()],
           axis=1)
```

	`CVhqregClassifier (VIF)` NestedCV Perf.	`CVhqregClassifier (step2o)` NestedCV Perf.
`roc_auc`	0.951930	0.988101
`F1`	0.859662	0.859557
`F1_macro`	0.889801	0.895817
`precision`	0.890895	0.983918
`recall`	0.835065	0.768398
`average_precision`	0.924692	0.982254
`balanced_accuracy`	0.885350	0.879953
`accuracy`	0.898214	0.908647
`matthews_corrcoef`	0.783548	0.809782

To summarize everything that has been stated regarding LAD regression, it is evident that the full model, including all predictors without any filtering, outperforms all the other models. The second-order model with `stepAIC` demonstrates promising performance, closely approaching the performance of the full model. However, the `stepAIC` model shows slightly inferior performance compared to the second-order model. Ultimately, the VIF model exhibits the lowest performance, with an average ROC AUC of 0.95, while the other models achieve ROC AUC values above 0.98.

6

Conclusions

The aim of this thesis was to investigate alternative methods for addressing supervised classification problems by comparing a range of models, including penalized LAD and OLS. These models are of particular interest due to their simplicity, numerical stability, and ease of interpretation, making them accessible and practical for a wide range of users. The results of this thesis could be especially useful for medical professionals, helping them to identify which predictors are truly impactful and which ones do not provide valuable information.

The table below summarizes the nested cross-validation performances from the previous chapter. This will be the basis for comparing the models. The comparison will hinge on three metrics: ROC AUC, `F1`-score, and Balanced Accuracy.

Model	**Nested CV Performance**		
	`ROC AUC`	`F1-Score`	`Balanced Accuracy`
`Full Binomial` (Penalized Logistic Regression; `R-Pathwise CD`)	0.995	0.981	0.971
`Full Binomial` (Penalized Logistic Regression; `Python-saga`)	0.994	0.968	0.972
`VIF Binomial` (Penalized Logistic Regression)	0.956	0.927	0.896
`StepAICc Filter: Binomial` (Penalized Logistic Regression)	0.990	0.953	0.960
`Second-Order-Terms: Binomial` (Penalized Logistic Regression)	0.994	0.975	0.966
`Full Gaussian` (Penalized Linear Regression)	0.993	0.948	0.942
`VIF Gaussian` (Penalized Linear Regression)	0.956	0.922	0.894
`StepAICc Filter: Gaussian` (Penalized Linear Regression)	0.987	0.925	0.932
`Second-Order-Terms: Gaussian` (Penalized Linear Regression)	0.992	0.963	0.939
`Full LAD` (Penalized Least Absolute Deviations)	0.991	0.894	0.905
`VIF LAD` (Penalized Least Absolute Deviations)	0.952	0.860	0.885
`StepAICc Filter: LAD` (Penalized Least Absolute Deviations)	0.986	0.892	0.905
`Second-Order-Terms: LAD` (Penalized Least Absolute Deviations)	0.988	0.860	0.880
`Random Forest`	0.991	0.944	0.953
`Gradient Boosting Machines`	0.992	0.944	0.952
`Adaptive Boosting`	0.993	0.966	0.970

Table 6.1: Nested cross-validation summary

The presented table provides a comprehensive summary of the performance metrics from a series of models trained using various techniques. The metrics evaluated include ROC AUC, `F1`-score, and Balanced Accuracy. Penalized logistic regression, using all first-order predictors, demonstrates the highest overall performance across all metrics, with a ROC AUC of 0.995, `F1`-score of 0.981, and Balanced Accuracy of 0.971. The high `F1`-score is especially noteworthy as it suggests a balance between precision (the ability to correctly identify cancerous cases) and recall (the ability to correctly identify non-cancerous cases), crucial for medical diagnostics where false positives and false negatives can have significant implications.

Ensemble learning methods such as Random Forest, Gradient Boosting Machines, and Adaptive Boosting have shown strong performance as well, with AdaBoost closely following the performance of penalized logistic regression. These models leverage the power of multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone. Their near top-tier performance aligns with the general expectation that ensemble methods often outperform individual models.

The inclusion of second-order terms in the model did not considerably improved the performance, implying that the additional complexity did not contribute significant predictive power. This suggests that the relationships in the data are largely linear or that the second-order terms added are not capturing the non-linear relationships effectively. VIF-filtered models, on the other hand, did not fare as well. This could be due to the removal of significant predictors during the process, leading to a loss of important information. In future work, adjusting the threshold might be beneficial to explore the impact of including more predictors.

Despite their typical application to regression tasks, both Penalized Linear Regression and Penalized LAD Regression have demonstrated competitive performance in this binary classification problem of breast cancer diagnosis. The Penalized Linear Regression model showed strong performance with a ROC AUC of 0.993, `F1`-Score of 0.948, and Balanced Accuracy of 0.942. The Penalized LAD model, a variant of quantile regression known for its robustness against outliers, also held its own in this context. These findings suggest the flexibility of these models, providing evidence that they can be effective in scenarios beyond their standard use cases, including when dealing with categorical response variables.

To summarize, the presented analysis underscores the importance of careful model selection and predictor handling in predictive healthcare analytics, particularly in life-critical contexts such as early diagnosis of breast cancer. Despite the diversity of methods applied, penalized logistic regression emerged as the top-performing approach, exhibiting a balance between precision and recall. This balance is critical in a healthcare setting to minimize false diagnoses and

optimize patient care. Meanwhile, the comparable performance of ensemble learning techniques also signifies their potential utility, particularly AdaBoost, which is only slightly behind the logistic model. Conversely, the use of VIF filtering and second-order terms did not provide substantial benefits in this context, underscoring that adding complexity does not always result in better predictive power. The unexpected performance of penalized linear and LAD regressions underlines their versatility and broad applicability. Moving forward, these findings can guide methodological choices in similar predictive tasks, stressing the need for a balance between model performance, interpretability, and simplicity. In all, these insights represent a valuable contribution to the ongoing pursuit of improving early cancer diagnosis, a task of immense importance in enhancing patient prognosis and quality of life.

References

[1] Hirotogu Akaike. Information theory and an extension of the maximum likelihood principle. *Selected papers of hirotugu akaike*, pages 199–213, 1998.

[2] Hirotugu Akaike. A new look at the statistical model identification. *IEEE transactions on automatic control*, 19(6):716–723, 1974.

[3] Peter C Austin and Jack V Tu. Bootstrap methods for developing predictive models. *The American Statistician*, 58(2):131–137, 2004.

[4] Eric Bair, Trevor Hastie, Debashis Paul, and Robert Tibshirani. Prediction by supervised principal components. *Journal of the American Statistical Association*, 101(473):119–137, 2006.

[5] Eric Bair and Robert Tibshirani. Semi-supervised methods to predict patient survival from gene expression data. *PLoS biology*, 2(4):e108, 2004.

[6] Gilbert Bassett Jr and Roger Koenker. Asymptotic theory of least absolute error regression. *Journal of the American Statistical Association*, 73(363):618–622, 1978.

[7] Daniel J Benjamin and James O Berger. Three recommendations for improving the use of p-values. *The American Statistician*, 73(sup1):186–191, 2019.

[8] Leo Breiman. Bagging predictors. *Machine learning*, 24:123–140, 1996.

[9] Kenneth P. Burnham and David R. Anderson. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*. Springer, New York, NY, 2002.

[10] Jan De Leeuw. *Least squares optimal scaling of partially observed linear systems*. Springer, 2004.

[11] Bradley Efron. *Bootstrap methods: another look at the jackknife*. Springer, 1992.

[12] Bradley Efron, Trevor Hastie, Iain Johnstone, and Robert Tibshirani. Least angle regression. 2004.

[13] Wolfgang Forstmeier, Eric-Jan Wagenmakers, and Timothy H Parker. Detecting and avoiding likely false-positive findings–a practical guide. *Biological Reviews*, 92(4):1941–1968, 2017.

[14] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[15] J Friedman, T Hastie, and H Hofling. Coordinate ascend optimization. *Ann. Appl. Stats*, 1:302–332, 2007.

[16] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

[17] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[18] Carl Friedrich Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium auctore Carolo Friderico Gauss*. sumtibus Frid. Perthes et IH Besser, 1809.

[19] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360, 2009.

[20] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer, New York, NY, 2009.

[21] Gergely Hegyi and László Zsolt Garamszegi. Using information theory as a substitute for stepwise regression in ecology and behavior. *Behavioral Ecology and Sociobiology*, 65:69–76, 2011.

[22] Peter D. Hoff. *A First Course in Bayesian Statistical Methods*. Springer Texts in Statistics. Springer, New York, NY, 2009.

[23] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[24] Adrien Marie Legendre. *Nouvelles méthodes pour la détermination des orbites des comètes; par AM Legendre...* chez Firmin Didot, libraire pour lew mathematiques, la marine, l ..., 1806.

[25] Norman Matloff. *The art of R programming: A tour of statistical software design*. No Starch Press, 2011.

[26] John Neter, Michael H Kutner, Christopher J Nachtsheim, William Wasserman, et al. Applied linear statistical models. 1996.

[27] Joseph Raphson. *Analysis aequationum universalis*. Typis TB prostant venales apud A. and I. Churchill, 1702.

[28] Yosiyuki Sakamoto, Makio Ishiguro, and Genshiro Kitagawa. Akaike information criterion statistics. *Dordrecht, The Netherlands: D. Reidel*, 81(10.5555):26853, 1986.

[29] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[30] Ioannis Tsamardinos, Elissavet Greasidou, and Giorgos Borboudakis. Bootstrapping the out-of-sample predictions for efficient and accurate cross-validation. *Machine learning*, 107:1895–1922, 2018.

[31] Ioannis Tsamardinos, Amin Rakhshani, and Vincenzo Lagani. Performance-estimation properties of cross-validation-based protocols with simultaneous hyper-parameter optimization. *International Journal on Artificial Intelligence Tools*, 24(05):1540023, 2015.

[32] Muhammad Umer, Mahum Naveed, Fadwa Alrowais, Abid Ishaq, Abdullah Al Hejaili, Shtwai Alsubai, Ala'Abdulmajid Eshmawi, Abdullah Mohamed, and Imran Ashraf. Breast cancer detection using convoluted features and ensemble machine learning algorithm. *Cancers*, 14(23):6015, 2022.

[33] Jun Xu. *Modern Applied Regressions: Bayesian and Frequentist Analysis of Categorical and Limited Response Variables with R and Stan*. CRC Press, 2022.

[34] Hui Zou. The adaptive lasso and its oracle properties. *Journal of the American statistical association*, 101(476):1418–1429, 2006.