Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

ESCUELA POLITECNICA SUPERIOR

Trabajo Fin de Grado

Development of an R package to learn supervised classification techniques

Author: Víctor Amador Padilla

Advisor: Juan José Cuadrado

2023

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Development of an R package to learn supervised classification techniques

Author: Víctor Amador Padilla

Advisor: Juan José Cuadrado

Tribunal:

President:

1st Vocal:

2nd Vocal:

Deposit date: September 25th, 2023

Abstract

This TFG aims to develop a custom R package for teaching supervised classification algorithms, starting with the identification of requirements, including algorithms, data structures, and libraries. A strong theoretical foundation is essential for effective package design. Documentation will explain each function's purpose, accompanied by necessary paperwork.

The package will include R scripts and data files in organized directories, complemented by a user manual for easy installation and usage, even for beginners. Built entirely from scratch without external dependencies, it's optimized for accuracy and performance.

In conclusion, this TFG provides a roadmap for creating an R package to teach supervised classification algorithms, benefiting researchers and practitioners dealing with real-world challenges.

Keywords: Supervised classification, RStudio, R package, package development, CRAN, decision rules, linear regresion, perceptron.

Contents

Abstract v

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1

2 Theoretical and social framework 3

2.1 Introduction . . . 3
2.2 State of Art . . . 5
2.3 R Language . . . 7
2.4 Supervised Classification . . . 8
2.4.1 Common concepts for all supervised classification algorithms. . . . 9
2.4.1.1 Data collection and understanding . . . 9
2.4.1.2 Data preparation . . . 10
2.4.1.3 Cross-Validation . . . 13
2.4.1.4 Hyperparameter tuning . . . 16
2.4.1.5 Checks and metrics . . . 16
2.4.2 Decision Trees . . . 20
2.4.2.1 hyperparameter tuning in trees . . . 20
2.4.2.2 Pruning . . . 20
2.4.2.3 Random Forest . . . 21
2.4.2.4 Gradient Boosting Trees . . . 23
2.4.3 Neural Networks . . . 26
2.4.3.1 Multi-Neuron Perceptrons (MLPs) . . . 26

3 Development **41**

3.1 Introduction . . . 41
3.2 Regression . . . 42
3.2.1 Linear Regression . . . 42
3.2.1.1 Theoretical approach . . . 42
3.2.1.2 Coded implementation . . . 45
3.2.1.3 Results . . . 48
3.2.2 Polynomial Regression . . . 50
3.2.2.1 Theoretical approach . . . 50
3.2.2.2 Coded implementation . . . 52
3.2.2.3 Results . . . 55

3.3 k-NN . . . 56
3.3.1 Theoretical approach . . . 56
3.3.1.1 Training Phase . . . 56
3.3.1.2 Prediction Phase . . . 56
3.3.1.3 Mathematical Formulation . . . 62
3.3.1.4 Choosing the Right k . . . 62
3.3.1.5 Advantages and Limitations . . . 64
3.3.1.6 Conclusion . . . 65
3.3.2 Coded implementation . . . 66
3.3.3 Results . . . 70
3.4 Perceptron . . . 73
3.4.1 Theoretical approach . . . 73
3.4.2 Coded implementation . . . 74
3.4.3 Results . . . 78
3.5 Decision Tree . . . 81
3.5.1 Theoretical approach . . . 81
3.5.1.1 Key Components of a Decision Tree . . . 82
3.5.1.2 Number of max child node selection . . . 87
3.5.2 Coded implementation . . . 88
3.5.3 The `decision_tree` Function . . . 89
3.5.4 The `aux_decision_tree` Function . . . 90
3.5.5 The `gain_method` Function . . . 92
3.5.6 The `entropy`, `gini`, and `error` Functions . . . 93
3.5.6.1 The `entropy` Function . . . 94
3.5.6.2 The `gini` Function . . . 94
3.5.6.3 The `error` Function . . . 95
3.5.7 The `all_combs` Function . . . 95
3.5.8 The `comb` Function . . . 96
3.5.8.1 Printing the tree . . . 98
3.5.9 Results . . . 99
3.6 Complementary functions . . . 103
3.7 Showcase dataframes . . . 104
3.8 How to create and upload an R package . . . 105
3.8.1 RStudio . . . 105
3.8.2 Roxygen2 . . . 106
3.8.3 Devtools . . . 107
3.8.4 Upload to CRAN . . . 108
3.8.4.1 Package versioning . . . 108
3.8.4.2 GitHub . . . 108
3.8.4.3 CRAN . . . 109

4 Conclusions and future improvements 111
4.1 Conclusions . . . 111
4.2 Future improvements . . . 112

Bibliography 115

List of Figures

2.1 K-Fold Cross-Validation 14
2.2 Stratified Cross-Validation 15
2.3 Leave-One-Out Cross-Validation 15
2.4 ROC and AUC 18
2.5 Bias-Variance Trade-Off 18
2.6 Underfitting 19
2.7 Fitting 19
2.8 Overfitting 19
2.9 Random Forest Method 22
2.10 Gradient Boosting Tree 24
2.11 MLPs Structure 27
2.12 MLPs Forward propagation 28
2.13 Sigmoid Activation Function 29
2.14 Tanh Activation Function 30
2.15 ReLu Activation Function 30
2.16 Softplus Activation Function 31
2.17 Linear Activation Function 31
2.18 Unit Step Activation Function 32
2.19 Selu Activation Function 33
2.20 GELU Activation Function 34
2.21 Swish Activation Function 34
2.22 CNNs Structure 36
2.23 RNNs Structure 37
2.24 GANs Structure 38
2.25 GANs 1’st aproximation 38
2.26 GANs 2’nd aproximation 39
2.27 GANs n’th aproximation 39

3.1 simple scatterplot 43
3.2 scatterplot matrix 43
3.3 Linear Regression 44
3.4 Regression Line Flowchart 48
3.5 Linear Regression Result 50
3.6 Multivariate Linear Regression Result 51
3.7 Multivariate Polynomial Regression Result (Degree = 4) 55
3.8 Multivariate Polynomial Regression Result (Degree = 8) 56
3.9 euclidean distance 57
3.10 Manhattan Distance 57

3.11 Chebyshev Distance 58
3.12 Minkowski Distance 58
3.13 Hamming Distance 59
3.14 Octile Distance 60
3.15 Jaccard Distance 60
3.16 Cosine Distance 60
3.17 K = 1 61
3.18 K = 3 61
3.19 K = 4 61
3.20 KNN Flowchart 69
3.21 KNN Result Details 72
3.22 KNN Result Scatterplot Matrix 72
3.23 Perceptron Structure 74
3.24 Perceptron Learning Rule 74
3.25 Linear Separability 75
3.26 Non-Linear Separability 75
3.27 Perceptron Flowchart 78
3.28 Perceptron function predictions 79
3.29 Perceptron function, Xor dataset predictions 80
3.30 Perceptron output explanations 81
3.31 Decision Tree Structure 83
3.32 Decision Tree Results_1 99
3.33 Decision Tree Results_2 100
3.34 Decision Tree Results_3 100
3.35 Decision Tree Results_4 100
3.36 Decision Tree Results_5 101
3.37 Decision Tree Results_6 101
3.38 Decision Tree Results_7 102
3.39 Decision Tree Results_8 102

List of Tables

3.1 input data 49
3.2 Flower Features 70
3.3 And_data. 79
3.4 Or_data. 79
3.5 Xor_data. 80
3.6 db2 99

Chapter 1

Introduction

This document is a comprehensive guide to the development of a package in the R language for supervised classification analysis. The package is designed to group various information processing algorithms that utilize supervised classification techniques. It has been created with the purpose of providing an effective tool for individuals seeking to learn these algorithms.

The R language is a powerful tool that is specifically designed for the handling of large data sets and for statistical and mathematical analysis. It is widely used by individuals around the world, and the community continues to grow and develop new mechanisms and processes. Therefore, it is essential to continuously improve the existing materials and provide new and clearer contents to support the growth of the R language community.

The Comprehensive R Archive Network (CRAN) is a collection of file storage servers designed to gather R packages created by language users. These packages undergo expert review before being included in the network. By contributing to CRAN, package authors actively contribute to the advancement of the R language and its resources. They provide access to their research work, offering documentation and support for other users who may need it. Our development objectives include submitting this package to this organization to share new knowledge and enhance the capabilities of R as a powerful tool. Meeting CRAN's publication requirements will be a key focus in the development of this project.

The main objective of this package is to collect and document the primary algorithms for supervised classification analysis. The package is designed to explain how each algorithm functions, and to provide an adaptable data model for each algorithm to enhance its performance. This package is intended for students and frequent users of this technology, providing them with the necessary tools to learn and develop their projects.

The package includes auxiliary functions that perform partial algorithm functions, as well as detailed explanations of the steps followed by the algorithm. This allows more inexperienced users to understand how the algorithm works by observing the step-by-step process of the data and partial classifications. This feature enables users to execute the algorithm with their own data, thus gaining a deeper understanding of its functioning.

Furthermore, the package includes several examples of the algorithms in action. These examples are designed to demonstrate how the algorithms can be used to solve real-world problems. They provide users with a clear understanding of the application of the algorithms and demonstrate their effectiveness in solving a wide range of problems.

Overall, this package provides users with a comprehensive guide to the algorithms for supervised classification analysis. It is an excellent resource for individuals seeking to learn these algorithms and is an essential tool for students and frequent users of this technology. The package's adaptable data model, data preparation features, and visualization tools, along with the included examples, provide users with the necessary tools to develop their projects and enhance their knowledge of the R language.

Chapter 2

Theoretical and social framework

2.1 Introduction

In today's rapidly evolving landscape, the value of information is unparalleled for businesses, governments, and individuals seeking meaningful insights. Data Science, a dedicated discipline, delves into the study and analysis of data, relying on extensive data storage and subsequent processing to drive informed decision-making based on observed patterns and relationships.

To unlock the full potential of data, a diverse set of tools is crucial for interpreting the vast volumes of collected information. It is imperative to develop robust systems capable of handling this complex task. However, a challenge arises in the form of defining concise objectives due to the absence of standardized interpretation. Each analysis is contingent upon specific goals. Diverse conclusions are drawn from the same dataset, but their objectives diverge.

Data Science seamlessly blends statistics, mathematics, and computer science to process seemingly unremarkable information and derive valuable insights. As a relatively nascent field, it continuously evolves, constantly pushing the boundaries of knowledge to leverage the data's potential for economic advantages across diverse industries. Pioneering researchers continually enhance the field, expanding its breadth and impact.

Moreover, alongside Data Science, new concepts like Big Data, macrodata, and other data processing sciences have emerged. These disciplines are increasingly vital in modern enterprises and environments where data-driven insights are indispensable. Consequently, dedicated roles have surfaced to address the evolving demands and requirements.

In conjunction with Data Science, essential tools have evolved to meet these emerging needs. One such tool is the R language, purpose-built for efficient data manipulation. Despite its inception in 1993, R has gained substantial influence as data processing assumes greater significance. Its versatile application domains have expanded, owing to its user-friendly nature and ability to handle massive datasets. Additionally, languages like Python, while not exclusive to data processing, offer powerful features for information analysis and manipulation.

Frameworks such as Hadoop (Apache Software Foundation, 2006) also called HDFS (Hadoop Distributed File System) and Spark (University of Berkeley, 2009) and technologies like Apache Hive (Apache Software Foundation, 2010) or Cloudera Impala (Apache Software Foundation, 2013) have emerged as indispensable tools. These open-source platforms specialize in managing immense amounts of data simultaneously, striving for operational efficiency. Born out of the rising demand for data analysis, they have become the go-to choices for specialized companies. Although sharing a similar architecture, Spark emerged as an evolution, optimizing processing operations, maintenance, and execution time compared to Hadoop. All this technologies are also versatile and compatible with each other, allowing different frameworks to interact and provide a new unprecedented data processing capability. The availability of open-source licenses has fueled their widespread adoption, making them the prevailing frameworks in use today.

2.2 State of Art

The development of R packages for supervised classification has experienced significant growth in recent years. Numerous undergraduate thesis projects have addressed this area, focusing on creating efficient and user-friendly tools and algorithms for the data science community. These packages have become essential components in the data analysis workflow, enabling users to easily implement and evaluate supervised classification methods.

A key aspect in the development of supervised classification packages is the incorporation of machine learning algorithms. Undergraduate thesis projects have explored various techniques, such as decision trees, support vector machines (SVM), neural networks, and ensemble methods like Random Forest and Gradient Boosting. These algorithms have been implemented and optimized in R packages, providing a wide range of options to tackle classification problems.

Another relevant focus in these projects has been the incorporation of data preprocessing techniques in the supervised classification packages. Normalization, feature selection, outlier detection and treatment, as well as missing data imputation, are some of the key aspects addressed in these works. The developed packages have empowered users to efficiently perform these preprocessing tasks, aligned with best practices.

The evaluation and validation of supervised classification models have also been highlighted in the undergraduate thesis projects related to the development of R packages. These works have addressed the implementation of evaluation metrics such as accuracy, sensitivity, specificity, and area under the ROC curve. Additionally, functions for cross-validation, model selection, and result visualization have been developed, allowing users to gain a more comprehensive and robust understanding of the implemented classification models.

The development of R packages for supervised classification has evolved significantly through research and related undergraduate thesis projects. These works have addressed fundamental aspects such as the implementation of machine learning algorithms, data preprocessing techniques, and model evaluation, enabling users to benefit from more robust and efficient tools in their supervised classification tasks.

In this environment of developing R packages for supervised classification, a package that aims to explain supervised classification algorithms can play a valuable role. While there are already numerous packages focused on implementing and optimizing classification algorithms, an explanatory package can provide additional benefits to users by enhancing their understanding and interpretation of these algorithms.

Such a package can include functionalities that help users explore the inner workings of different supervised classification algorithms. It can offer visualizations and interactive tools to illustrate the decision boundaries, feature importance, or model parameters, allowing users to gain insights into how the algorithms make predictions. This can be particularly useful for researchers, data scientists, and practitioners who seek a deeper understanding of the algorithms they are working with.

Furthermore, the explanatory package can provide detailed documentation, tutorials, and examples that explain the underlying principles and concepts of various supervised classification algorithms. It can cover topics like algorithmic assumptions, strengths, weaknesses, and applicability in different scenarios. This educational aspect can empower users to make informed choices when selecting and applying classification algorithms to their specific datasets.

In summary, a package focused on explaining supervised classification algorithms can complement the existing landscape of R packages by providing users with enhanced understanding, interpretability, and educational resources. By offering visualizations, documentation, and explanations of algorithmic behavior and prediction outcomes, this package can empower users to make better-informed decisions and gain deeper insights into the supervised classification models they employ.

2.3 R Language

R stands out as a specialized language for statistical and mathematical analysis, specifically tailored to handle extensive datasets. It evolved from the S programming language, known for its pioneering contributions to statistical programming. While initially relying on Fortran-based subroutines, they fell short in terms of performance and scalability when dealing with large data volumes. As the field expanded, there arose a pressing need for enhanced tools, leading to the birth of R.

This powerful tool offers a plethora of options for data visualization and interpretation, providing users with diverse and customizable results that align with their unique objectives, being one of its key advantages. With its extensive range of features , plenty of them provided by diverse packages supplied by developers, users have the flexibility to harness its capabilities according to their specific needs. R is an integral part of the GNU project and distributed under the GNU GPL (General Public License), a free and open-source operating system that advocates for software freedom, focusing on users' rights to use, study, modify, and share software, ensuring its availability as free software for anyone to utilize. This freedom to share and modify has played a significant role in its current prominence. R has emerged as the language of choice for data management, processing, and graphical representation.

Python also has a strong ecosystem for data analysis, particularly with libraries like Pandas, NumPy, and scikit-learn. Python's versatility, general-purpose nature, and integration with other domains such as web development and machine learning make it a popular choice for data analysis as well.

However we have chosen R over Python for this project mainly for these reasons:

- Statistical and Data Analysis Focus: R was specifically designed for statistical analysis and has a rich ecosystem of packages and libraries dedicated to data analysis. It provides a wide range of statistical models, tests, and visualizations, making it particularly well-suited for statistical exploration and research.

- Data Visualization Capabilities: It offers extensive data visualization capabilities. These allows for the creation of highly customizable and publication-quality plots, making it easier to visually communicate insights from your data.

- Community and Packages: It has a vibrant and active community of statisticians, data scientists, and researchers who contribute to the development of numerous packages. These packages cover a wide range of statistical techniques, machine learning algorithms, and data manipulation tools, providing a comprehensive toolkit for data analysis.

- Reproducibility and Documentation: R promotes reproducible research through literate programming tools like R Markdown, which allows you to combine code, documentation, and visualizations in a single document. This makes it easier to create reproducible analyses and share them with others.

- Legacy and Research Adoption: R has a long history in the field of statistics and research, and many academic institutions and researchers have adopted it as their preferred tool for data analysis. This legacy has resulted in a wealth of resources, tutorials, and case studies that can aid in learning and applying statistical techniques.

Given its remarkable attributes, R will be the primary language employed for the development of this package, enabling users to leverage its vast potential for effective data analysis and exploration.

2.4 Supervised Classification

Supervised classification is a fundamental concept in machine learning that involves training a model to assign predefined labels to input data based on known examples. It is a type of supervised learning where the dataset used for training consists of labeled instances, each associated with a target class. The goal of supervised classification is to learn a mapping from the input features to the corresponding class labels, allowing the model to accurately classify unseen instances.

Supervised classification finds applications in various domains. For instance, in medical diagnosis, a model can be trained to classify medical images as cancerous or non-cancerous based on a set of labeled images. In sentiment analysis, a model can classify text data as positive or negative sentiments to analyze customer reviews. In fraud detection, a model can be trained to identify fraudulent transactions based on historical data labeled as fraudulent or legitimate.

The benefits of supervised classification are numerous. It enables automation and scalability, allowing for efficient processing of large volumes of data. It aids in decision-making by providing predictions and insights based on learned patterns. Supervised classification also supports the development of intelligent systems, such as recommendation engines, personal assistants, and autonomous vehicles.

Supervised classification is a machine learning technique that involves training a model to assign predefined labels to input data. It finds applications in various fields, providing the ability to automate tasks, make informed decisions, and develop intelligent systems. By learning from labeled examples, supervised classification empowers machines to accurately classify unseen instances based on learned patterns and relationships.

To illustrate, let's consider an example of email spam classification. Suppose we have a dataset of emails, each labeled as either "spam" or "not spam." The features of the emails could include the subject line, sender's address, and content. By using a supervised classification algorithm, such as logistic regression, decision trees, or neural networks, we can train a model to learn patterns and relationships within the dataset. The model can then be used to classify incoming emails as either spam or not spam based on the learned patterns.

- Linear regression: 3.2.1 Is a supervised learning algorithm used for modeling the relationship between a dependent variable and one or more independent variables. It assumes a linear relationship between the input features and the target variable. By estimating the slope and intercept, linear regression finds the best-fitting line that minimizes the differences between predicted and actual values. It is computationally efficient, interpretable, and widely applied in fields like economics and social sciences to understand and predict relationships between variables.
- Polynomial regression: 3.2.2 Is a type of regression analysis used in statistics and machine learning to model the relationship between a dependent variable and one or more independent variables when the relationship is not linear but exhibits a curved or nonlinear pattern. Unlike simple linear regression, which fits a straight line, polynomial regression fits a polynomial equation to the data, allowing it to capture more complex relationships. This is important in real-world applications because many natural phenomena and data sets do not adhere to linear behavior, and by employing polynomial regression, we can better understand, predict, and analyze such non-linearity. It finds utility in various fields, including economics, physics, biology, and engineering, enabling researchers and analysts to make more accurate predictions, uncover hidden patterns, and gain insights into complex systems by modeling them with higher-degree polynomial equations.
- K-Nearest Neighbors (KNN): 3.3 It is a non-parametric algorithm that classifies instances based on their proximity to labeled instances in the feature space. It assigns a class label to a new instance

by considering the majority class of its k nearest neighbors. KNN is simple and intuitive, works well with non-linear decision boundaries, and can handle multi-class problems. However, it requires storing the entire training dataset and can be sensitive to the choice of the number of neighbors.

- Decision Trees: Decision Trees 3.5 are versatile and intuitive algorithms for classification. They build a tree-like structure by recursively splitting the data based on feature conditions that maximize information gain using diverse methods such as Gini impurity, Entropy impurity or Error impurity. Each internal node represents a test on a feature, and each leaf node represents a class label. Decision Trees are easy to interpret, can handle categorical and numerical features, and capture non-linear relationships. However, they can be prone to overfitting and may not generalize well to unseen data. More advanced techniques such as gradient boosting trees 2.4.2.4 and random forest 2.4.2.3 solve some of this issues and perform notably better.

- Neural networks: Specifically the Perceptron, are fundamental components of modern machine learning. Neural networks 2.4.3 are a class of models inspired by the structure and functioning of biological neurons. They consist of interconnected nodes, known as neurons, organized into layers. Each neuron takes input from the previous layer, applies a weighted sum operation, and passes the result through an activation function to produce an output.

  The Perceptron 3.4 is a basic neural network model that serves as a building block for more complex architectures. It is a type of feedforward neural network with a single layer of neurons. The Perceptron takes input features, assigns weights to them, and computes a weighted sum. Then, it applies an activation function, typically a step function, to produce a binary output. The training of the Perceptron involves adjusting the weights based on prediction errors using a learning algorithm called the perceptron learning rule.

2.4.1 Common concepts for all supervised classification algorithms.

Irrespective of the specific supervised classification model that is chosen for implementation, there exist several shared aspects that apply universally. These aspects encompass elements such as cross-validation and the selection of hyperparameters, among others. These commonalities emphasize the importance of rigorous evaluation and optimization in ensuring the effectiveness and generalizability of supervised classification models.

2.4.1.1 Data collection and understanding

Data collection and understanding are pivotal stages in constructing successful supervised classification models. These stages lay the groundwork for accurate analysis and informed decision-making by ensuring that the data used for modeling is comprehensive, accurate, and deeply understood.

Data collection involves gathering pertinent information from diverse sources like databases, surveys, APIs, and websites. This process requires meticulous planning to guarantee that the data collected accurately represents the problem at hand. Defining the data's scope and identifying relevant features are crucial steps that minimize bias and set the stage for creating a robust classification model.

After data collection, gaining a profound understanding of its intricacies is vital. This encompasses studying the data's structure, content, and quality. Key components of data understanding include exploring feature types (categorical, numerical, text-based) and their distribution.

Missing data and data quality checks are integral. Identifying missing values and determining their patterns, as well as detecting inaccuracies or inconsistencies, ensures the credibility of the analysis.

Integrating domain expertise enhances the process, helping recognize meaningful patterns, understand feature relevance, and identify anomalies or outliers not immediately apparent through data analysis.

Correlated variables are another aspect to consider. Analyzing the relationships between features can provide insights into redundant or highly correlated variables. High correlation might indicate collinearity, which can affect model stability and interpretability. Addressing these correlations through feature selection or engineering can enhance the model's effectiveness.

The exploration of data through visualizations such as histograms, scatter plots, box plots, and heatmaps uncovers feature distribution, variable relationships, and potential outliers. This aids in making informed decisions about preprocessing and model selection.

Thorough documentation maintains transparency and reproducibility. Recording metadata, including data source, collection methods, and preprocessing steps, ensures that others grasp the data's context and limitations.

Data collection and understanding establish the bedrock for effective supervised classification models. An in-depth grasp of data features, distribution, and quality informs subsequent modeling steps, resulting in more precise and dependable outcomes.

2.4.1.2 Data preparation

Data preparation is a critical and often time-consuming phase in building effective supervised classification models. The quality of your dataset and how well it's preprocessed significantly impact the model's performance and generalization ability. This process involves various steps, each aimed at ensuring the data is clean, relevant, and ready for analysis.

- **Data Cleaning:** Also known as data cleansing or data scrubbing [1] [2], is a fundamental process in the preparation of data for supervised classification models. It involves identifying and rectifying errors, inconsistencies, missing values, and outliers in a dataset. By ensuring that the data is accurate, complete, and reliable, data cleaning significantly improves the performance and validity of the subsequent classification model.

  Missing values are a common issue in datasets that can adversely affect model performance. In data cleaning, missing values must be handled appropriately. Depending on the context, you can choose to delete rows with missing values, fill them with statistical measures such as the mean or median of the feature, or employ more sophisticated imputation techniques. The choice of method depends on the amount of missing data, the feature's nature, and the impact on the model's integrity.

  Outliers are data points that deviate significantly from the rest of the data. Outliers can skew model predictions and adversely affect the model's performance. Data cleaning involves detecting outliers using methods like the Interquartile Range (IQR) or Z-score and deciding whether to remove or transform them. Removing extreme outliers can prevent them from influencing the model, while transforming techniques like log transformation can mitigate their impact.

  Errors in the dataset, such as typographical mistakes or incorrect entries, can introduce noise into the model. Data cleaning also means identifying and correcting these inaccuracies. Techniques like data profiling and validation rules can help uncover inconsistencies. Additionally, cross-referencing with domain-specific knowledge or external sources can aid in correcting errors and ensuring data accuracy.

  Ensuring consistent data formats is crucial for accurate modeling. Data cleaning includes tasks like standardizing date formats, converting units, and addressing inconsistent categorization. This step minimizes confusion during analysis and ensures that the data is uniform and interpretable.

After the data cleaning process, it's essential to verify that all identified issues have been successfully addressed. This involves rechecking for missing values, outliers, and inaccuracies. Documenting the data cleaning steps taken is crucial for transparency and reproducibility. A well-documented data cleaning process helps other stakeholders understand the transformations applied and validates the integrity of the dataset.

- **Feature Ingeneering:** Feature engineering is a pivotal process within the realm of supervised classification, where the raw dataset is transformed, augmented, and refined to create informative and relevant features for modeling. This crucial step empowers machine learning algorithms to extract meaningful patterns and relationships from the data, ultimately leading to more accurate and robust classification models.

  Feature engineering involves a variety of techniques:

  - **Feature Selection:** Choosing the most relevant features based on their contribution to the target variable and eliminating redundant or irrelevant ones [3]. This reduces noise in the model and speeds up training.
  - **Feature Creation:** Crafting new features that encapsulate essential information. For instance, converting a timestamp into day of the week or generating interaction terms to capture relationships between existing features.
  - **Feature Transformation:** Adjusting the distribution or scale of features to make them more suitable for the chosen algorithm [4]. Techniques include normalization (scaling features to a standard range) and log or power transformations.
  - **Domain Knowledge and Creativity:** A critical aspect of feature engineering is domain expertise. Understanding the problem and the data's context allows for the creation of features that encapsulate the most critical aspects of the problem. For instance, in image classification, manually crafting features like edges or textures can aid the model in capturing essential visual patterns.
  - **Handling Categorical Data:** Categorical variables pose unique challenges [5]. One-hot encoding converts categorical variables into binary features, while label encoding assigns numerical labels. Both approaches help the algorithm process categorical data. However, careful consideration is needed to avoid introducing bias or misinterpretation.
  - **Dimensionality Reduction:** In cases where the dataset has a large number of features, dimensionality reduction techniques like Principal Component Analysis (PCA) can be employed. These techniques consolidate information while retaining the most significant variance, thereby reducing computational complexity [6].
  - **Iterative Process and Model Feedback:** Feature engineering is an iterative process. Models can provide insights into which features contribute the most to their performance. Analyzing feature importance scores from tree-based models or coefficients from linear models can guide further feature selection and refinement [7].
  - **Caution against Overfitting:** While feature engineering enhances a model's performance, excessive feature creation or transformation can lead to overfitting. Overfitting occurs when the model learns the training data's noise and performs poorly on unseen data [8]. Regularization techniques and cross-validation can help manage this risk [9].

- **Data Splitting:**Data splitting is a fundamental step in the process of building supervised classification models [10]. It involves partitioning the dataset into distinct subsets to facilitate training, validation, and evaluation of the model's performance. Proper data splitting ensures that the model's ability to generalize to new, unseen data is accurately assessed, preventing issues like overfitting and providing a more reliable estimate of the model's performance.

The primary purpose of data splitting is to create separate sets for training and testing. The training set is used to train the model, allowing it to learn the underlying patterns and relationships in the data. The testing set, on the other hand, is used to evaluate the model's performance. It simulates real-world scenarios where the model encounters new, unseen data.

Data splitting plays a crucial role in preventing overfitting, a situation where the model memorizes the training data instead of learning its underlying patterns. By evaluating the model on a separate testing set, you can gauge its ability to generalize. If the model performs well on the training set but poorly on the testing set, it indicates overfitting.

In addition to a single train-test split, cross-validation is a powerful technique that enhances the assessment of model performance. It involves creating multiple train-test splits and evaluating the model on each of them. This process provides a more robust estimate of the model's performance by reducing the impact of the specific split on the assessment.

In classification tasks with imbalanced class distribution, stratified sampling is used to ensure that the proportion of each class remains consistent in both the training and testing sets. This approach prevents the model from being biased toward the majority class.

To ensure that the data subsets are representative of the overall dataset, randomization is employed during data splitting. Random shuffling helps reduce the risk of systematic bias and ensures that the model is exposed to diverse samples during training and evaluation.

In some cases, a validation set is introduced in addition to the training and testing sets. The validation set is used for hyperparameter tuning and model selection. By trying different hyperparameter configurations and selecting the one with the best performance on the validation set, you can optimize the model's performance before the final testing phase.

Data splitting is a critical step that safeguards against overfitting, provides a realistic assessment of model performance, and aids in the selection of optimal hyperparameters. It establishes a clear distinction between training and testing data, ensuring that the model's performance metrics accurately reflect its ability to generalize to new, unseen data.

- **Handling Imbalance:** Class imbalance is a common challenge in supervised classification [11], where the distribution of classes within the dataset is skewed, leading to one or more classes having significantly fewer instances than others. Addressing class imbalance is crucial for building classification models that provide fair and accurate predictions across all classes.

  Understanding imbalanced classes is essential because they can lead to biased model performance. Most machine learning algorithms aim to minimize overall error, which often results in prioritizing the majority class. Consequently, the minority class may be misclassified or overlooked, leading to suboptimal predictions for that class.

  There are two main techniques to handle class imbalance:

  - **Oversampling:** This increases the number of instances in the minority class by duplicating or generating synthetic samples [12]. Techniques like SMOTE create synthetic samples by interpolating between existing ones, balancing class distribution and helping the model learn the minority class better.

  - **Undersampling:** This reduces the number of instances in the majority class to match the minority class [13]. While it addresses imbalance, it might lead to loss of information and underutilization of available data.

  Certain algorithms are designed to handle imbalanced data inherently. For instance, ensemble methods like Random Forest and Gradient Boosting, which we will develop later on, can effectively deal with class imbalance due to their ability to combine multiple weak learners. These algorithms tend to assign higher weights to misclassified instances from the minority class, thereby giving more attention to underrepresented classes.

  In some cases, a combination of oversampling and undersampling techniques, known as hybrid approaches, can provide optimal results. These methods seek to strike a balance between addressing class imbalance and minimizing potential downsides.

- **Encoding and Scaling:** In the context of supervised classification, encoding and scaling are fundamental preprocessing techniques that play a pivotal role in preparing raw data for machine learning models. These techniques transform features into formats that machine learning algorithms can readily interpret, thereby enhancing the model's performance and improving its ability to make accurate predictions.

  Machine learning algorithms often require numerical data, and encoding serves as the bridge for converting categorical variables into numerical representations. Label encoding is one approach, here, every encoded value associates with a class label index. The other most used approach is One-Hot Encoding, where for a categorical feature with n unique classes, n binary columns (flags) are created, with a "1" indicating the presence of that category and "0" indicating absence.

  Numerical features frequently have varying scales, which can lead to some features disproportionately influencing the learning process. **Scaling numerical features** mitigates this issue, and two prevalent techniques are:

  - **Normalization (Min-Max Scaling):** $X_{\text{normalized}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$
  - **Standardization (Z-Score Scaling):** $X_{\text{standardized}} = \frac{X - \mu}{\sigma}$ where μ is the mean and σ is the standard deviation.

  Poor encoding might introduce bias due to incorrect ordinal relationships. Unscaled features can hinder convergence or optimization, while appropriately scaled and encoded features offer algorithms a level playing field for learning and generalizing.

  While encoding and scaling are powerful, consider Curse of Dimensionality, one-hot encoding can lead to high-dimensional data, challenging model performance. Interpretability should be taken into consideration too, scaling might impact the interpretability of features, especially in linear models or decision trees.

  Encoding and scaling should be part of a preprocessing pipeline to ensure consistent transformations on training and testing data, preventing data leakage and ensuring model integrity. Encoding and scaling are indispensable for preparing data for classification models. These techniques empower algorithms to effectively interpret both categorical and numerical features, enhancing model convergence, robustness, and predictive accuracy.

2.4.1.3 Cross-Validation

Cross-validation is a robust technique used to evaluate the performance of machine learning models, particularly in situations where limited data is available or to ensure unbiased assessment. It provides a more reliable estimate of a model's performance by simulating how the model would perform on unseen data. Cross-validation is crucial for selecting the best model, fine-tuning hyperparameters, and assessing how well the chosen model generalizes to new data.

This technique involves partitioning the available dataset into multiple subsets or "folds." It alternates between using one fold for testing and the rest for training. This process is repeated multiple times, with each fold taking a turn as the test set. The results from each iteration are then aggregated to provide an overall assessment of the model's performance. There are several ways to do this:

- **K-Fold Cross-Validation:** The most common form of cross-validation is K-Fold Cross-Validation 2.1, where the dataset is divided into K equally-sized folds. The process is as follows:

  1. The dataset is randomly shuffled.
  2. The dataset is divided into K folds.
  3. For each fold, the model is trained on K-1 folds and validated on the validation set.
  4. The performance metric (e.g., accuracy, F1-score) on the validation set is recorded for each iteration.
  5. After all iterations, the model with the best performance on the validation set is selected.
  6. The final assessment is conducted on the test set, providing an estimate of the model's generalization performance.

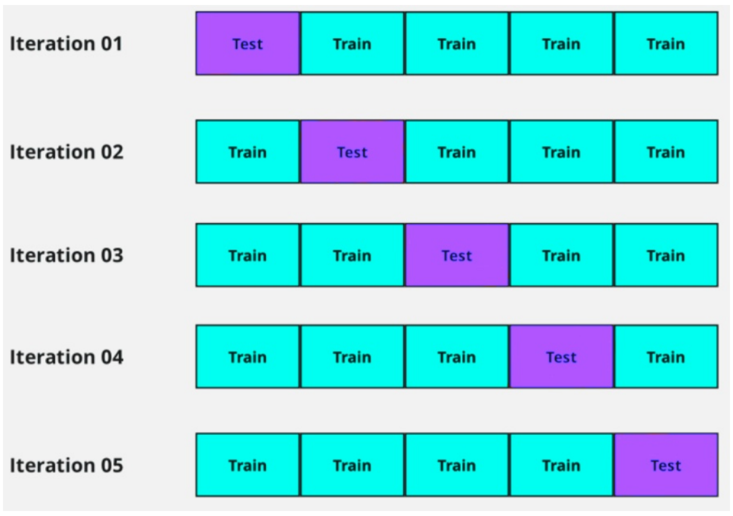


Figure 2.1: K-Fold Cross-Validation

- **Stratified Cross-Validation:** Stratified Cross-Validation 2.2 is employed when dealing with imbalanced datasets. It maintains the class distribution proportions in each fold. This ensures that no single fold is dominated by a particular class, providing a more representative assessment of the model's performance. For example, lets see this 70% - 30% distribution:

- **Leave-One-Out Cross-Validation (LOOCV):** LOOCV 2.3 is a special variant where each data point is used as the test set once and the remaining points are used for training, so n iterations are made, being n the total amount of data. While providing an unbiased performance estimate, LOOCV can be computationally expensive for larger datasets.

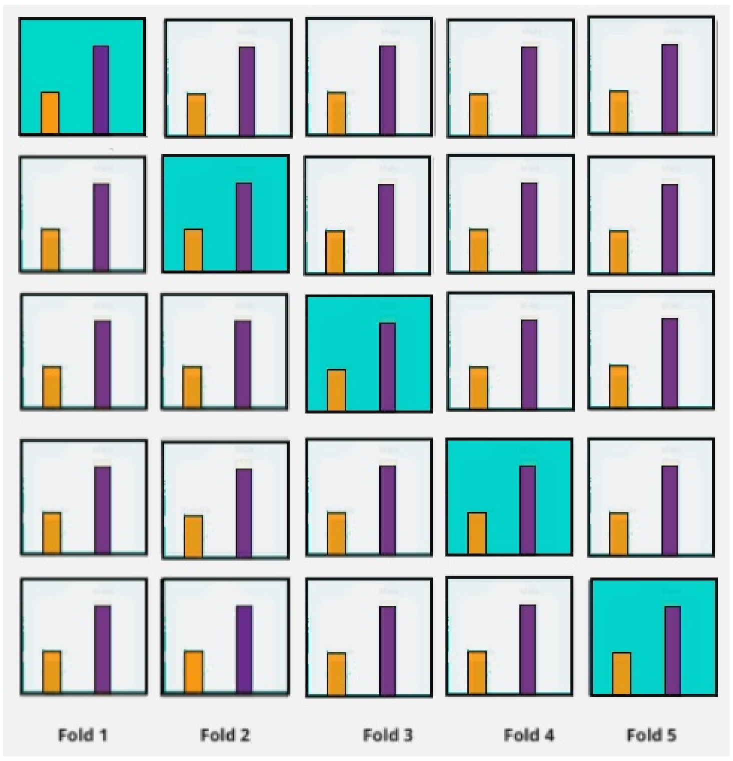


Figure 2.2: Stratified Cross-Validation

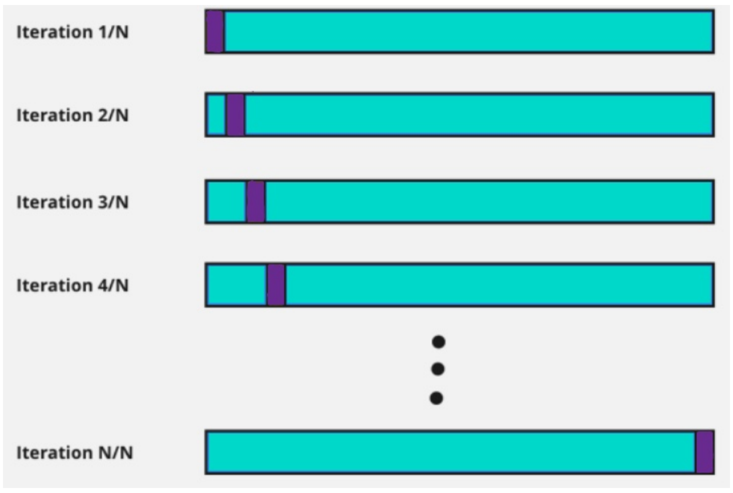


Figure 2.3: Leave-One-Out Cross-Validation

Let E_i be the performance metric (e.g., accuracy) on the validation set for the ith fold. The mean performance μ across all folds is given by:

$$\mu = \frac{1}{K}\sum_{i=1}^{K} E_i \tag{2.1}$$

The variance s^2 of the performance metric can be calculated as:

$$s^2 = \frac{1}{K-1}\sum_{i=1}^{K}(E_i - \mu)^2 \tag{2.2}$$

Cross-validation with a validation step ensures that model selection and hyperparameter tuning are conducted independently from the testing phase. It aids in selecting the best model configuration and hyperparameters based on performance on the validation set. Helps to prevent overfitting by providing a more accurate assessment of how the model will perform on unseen data.

Chosing a correct value of K is also really relevant depending on the cross-validation method used. Typical values for K range from 5 to 10. Smaller datasets benefit from LOOCV, while larger datasets may use a smaller K value. The choice of K depends on available data and computational resources.

In summary, cross-validation with a validation step is a crucial technique for robustly evaluating machine learning models. It aids in model selection, hyperparameter tuning, and providing a more accurate estimate of a model's performance on unseen data. Through the process of partitioning data, training, validating, and testing iteratively, cross-validation enhances the reliability of model assessment.

2.4.1.4 Hyperparameter tuning

Hyperparameter tuning is a crucial step in machine learning to find the best set of hyperparameters for a model, including decision trees [14]. Hyperparameters are settings that are not learned from the data but rather set before training. Here are some common hyperparameter tuning techniques for decision trees:

- **Grid Search:**Grid search involves specifying a set of possible values for each hyperparameter and exhaustively trying all combinations. It evaluates the model's performance using a chosen metric (e.g., accuracy for classification, mean squared error for regression) on a validation dataset for each combination. The combination with the best performance is selected.
- **Random Search:** Random search selects random combinations of hyperparameters from predefined ranges. This technique is more efficient than grid search when there are many hyperparameters or when some hyperparameters have a minimal impact on performance.
- **Bayesian Optimization:** Bayesian optimization uses probabilistic models to predict the performance of different hyperparameter settings. It balances the exploration of new hyperparameters with exploiting previously explored promising regions. Bayesian optimization can be more efficient in finding optimal hyperparameters with fewer evaluations compared to grid or random search.
- **Feature Selection:** Some decision tree algorithms allow you to set hyperparameters that control the number of features considered at each split. This can help mitigate the risk of overfitting by limiting the complexity introduced by a large number of features.

The choice of hyperparameters can significantly affect a model's performance, and the best hyperparameters might differ depending on the dataset and problem. It's essential to perform cross-validation to ensure that the selected hyperparameters generalize well to unseen data.

2.4.1.5 Checks and metrics

Model evaluation is a pivotal phase in the machine learning workflow, involving the assessment of a trained model's performance on unseen data. To ensure robustness and suitability, a variety of metrics and checks are employed, each offering unique insights into how well a model generalizes and whether it meets the desired criteria. The selection of appropriate metrics and checks is contingent on the specific problem, data characteristics, and the objectives of the analysis.

- **Accuracy:** Accuracy 2.3 is a commonly used metric, especially when dealing with balanced datasets. It calculates the ratio of correctly predicted instances to the total number of instances.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Instances}} \tag{2.3}$$

  While it offers a straightforward view of overall model performance, accuracy might not be suitable for imbalanced datasets, where a class dominates the other. In such cases, accuracy can be misleading, as a model may achieve high accuracy by favoring the majority class. However, for balanced datasets, accuracy provides a reliable measure of correctness.

- **Precision and Recall:** Precision and recall are particularly valuable for scenarios involving imbalanced classes, where the class distribution is skewed. Precision 2.4 measures the proportion of positive predictions that are truly positive, providing an indicator of the model's accuracy in identifying positive instances.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2.4}$$

  Recall 2.5, on the other hand, gauges the model's capability to detect all relevant positive instances.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{2.5}$$

  These metrics are especially useful when striking a balance between avoiding false positives (precision) and minimizing false negatives (recall) is essential.

- **F1-Score:** The F1-score 2.6 is a harmonic mean of precision and recall, offering a balanced measure when there's a trade-off between the two. It's particularly valuable when both high precision and high recall are desired.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.6}$$

  The F1-score helps find a middle ground between precision and recall, ensuring that the model's overall performance isn't skewed toward either extreme.

- **ROC Curve and AUC:** The Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC) provide insights into a model's discriminatory power in binary classification tasks. The ROC curve illustrates the relationship between true positive rate (recall) and false positive rate at different classification thresholds. AUC summarizes the ROC curve's performance, with higher values indicating superior model discrimination. The ROC curve and AUC are excellent tools for visualizing and quantifying a model's ability to distinguish between classes.

- **Confusion Matrix:** A confusion matrix 2.7 offers a comprehensive view of a model's performance by categorizing predictions into true positives, false positives, true negatives, and false negatives.

$$\begin{pmatrix} \text{True Positive} & \text{False Negative} \\ \text{False Positive} & \text{True Negative} \end{pmatrix} \tag{2.7}$$

  This matrix provides essential information for understanding the types of errors a model makes and its accuracy across different classes. Analyzing the confusion matrix aids in diagnosing areas where the model might require improvement or fine-tuning.

- **Bias-Variance Trade-Off:** Understanding the bias-variance trade-off is crucial for assessing a model's generalization ability [15]. High bias can lead to underfitting 2.6, where the model over-

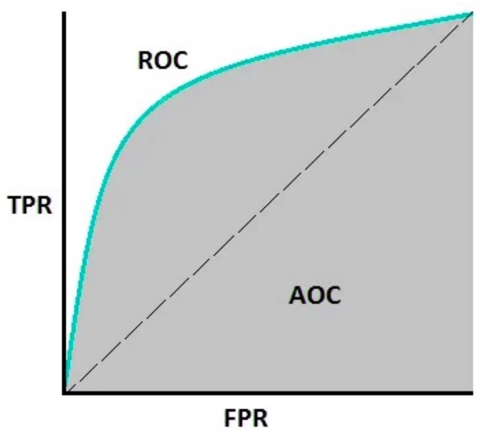


Figure 2.4: ROC and AUC

simplifies the data, while high variance can lead to overfitting 2.8, where the model captures noise. Striking a balance between bias and variance is essential for achieving a model that performs well 2.7 on both training and testing data.

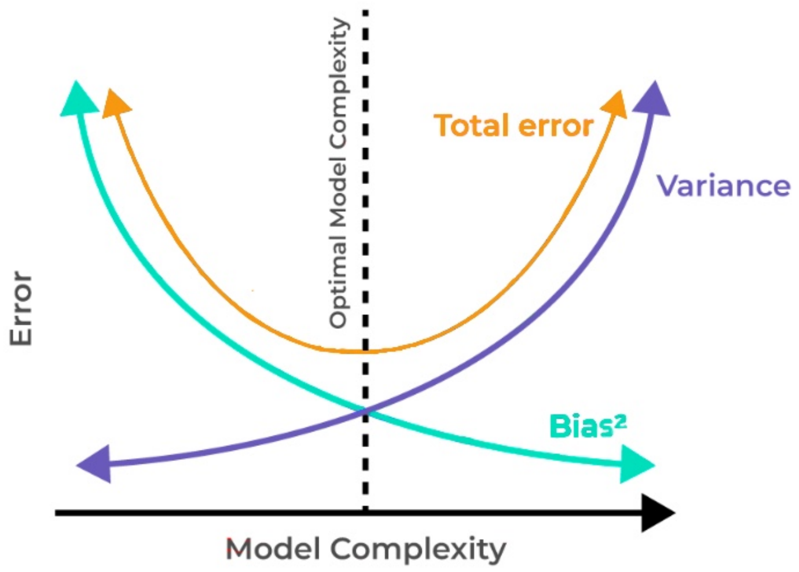


Figure 2.5: Bias-Variance Trade-Off

- **Overfitting and Underfitting Checks:** Comparing a model's performance on training and validation/test data helps in detecting signs of overfitting or underfitting. If a model performs exceptionally well on training data but poorly on validation/test data, it might be overfitting. Conversely, if a model's performance is consistently subpar on both training and validation/test data, it might be underfitting.

- **Domain-Specific Checks:** Depending on the problem domain, certain metrics and checks might be more relevant than others. For instance, in medical diagnoses, false negatives (missed diagnoses) could be more critical than false positives (false alarms). Domain expertise guides the selection of metrics that align with the problem's requirements and priorities.

- **Regularization Checks:** When regularization techniques like L1 (Lasso) 2.8 and L2 (Ridge) 2.9 are applied, the impact on model performance and feature importance should be examined.

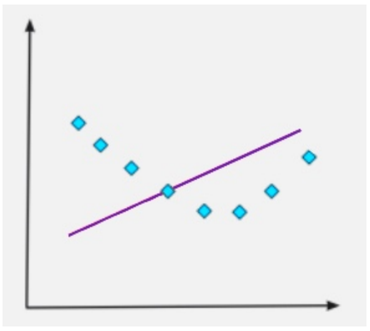


Figure 2.6: Underfitting

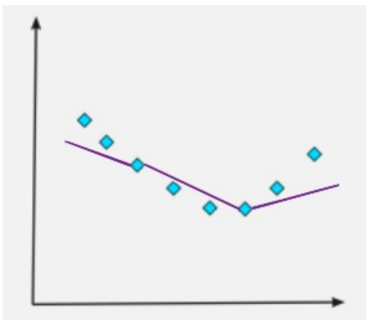


Figure 2.7: Fitting

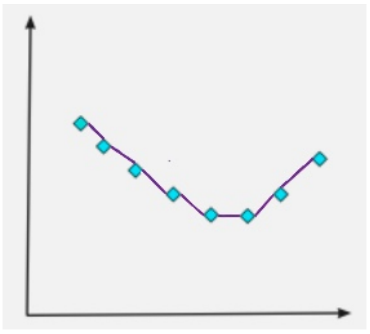


Figure 2.8: Overfitting

L1 regularization encourages sparse feature importance, potentially highlighting the most relevant features. L2 regularization mitigates extreme feature weights, leading to a more stable model.

$$\text{Lasso Regularization:} \quad \min_{\beta}\left\{\frac{1}{2n}\sum_{i=1}^{n}(y_i-\beta_0-\sum_{j=1}^{p}x_{ij}\beta_j)^2+\lambda\sum_{j=1}^{p}|\beta_j|\right\} \tag{2.8}$$

$$\text{Ridge Regularization:} \quad \min_{\beta}\left\{\frac{1}{2n}\sum_{i=1}^{n}(y_i-\beta_0-\sum_{j=1}^{p}x_{ij}\beta_j)^2+\lambda\sum_{j=1}^{p}\beta_j^2\right\} \tag{2.9}$$

Being:

- n: The number of data points in your dataset.
- y_i: The observed target value for the i-th data point.
- β_0: The intercept term in the linear regression equation.
- x_{ij}: The value of the j-th feature for the i-th data point.
- β_j: The coefficient corresponding to the j-th feature.
- λ: The regularization parameter, also known as the tuning parameter, that controls the strength of the penalty term. A larger λ will result in more aggressive shrinking of coefficients.

- **Metrics for Regression:** In regression problems, different metrics can come into play. For example the Mean Absolute Error (MAE) 2.10 quantifies the average absolute difference between predicted and actual values, offering insights into prediction accuracy. The Mean Squared Error (MSE) 2.11 computes the average of squared

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i-\hat{y}_i| \tag{2.10}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2 \tag{2.11}$$

In conclusion, model evaluation encompasses an array of metrics and checks that collectively contribute to a comprehensive understanding of a model's performance. The choice of metrics depends on the nature of the data and the specific objectives of the analysis. Rigorous evaluation ensures the selection of an appropriate model, validates its generalization ability, and guides the refinement of hyperparameters and features.

2.4.2 Decision Trees

A decision tree is a powerful and widely used machine learning algorithm for both classification and regression tasks. It's a graphical representation of a decision-making process that mimics how humans make decisions by breaking down complex problems into smaller, more manageable sub-problems. Decision trees are particularly useful for tasks where the relationship between input features and the target variable is non-linear and may involve interactions among features. The concept of decision trees in machine learning can be traced back to the late 1960s and early 1970s [16].

In the domain of machine learning and data analysis, decision trees serve as a fundamental tool for data-driven decision-making. These structures partition data based on key features and criteria, facilitating effective problem-solving. Basic decision trees and the developed implementation are fully detailed here 3.5 and Its previous reading is highly recommended to obtain a better understanding of the up-following topics.

However, the significance of decision trees reaches far beyond their basic concepts. In the forthcoming sections, we will delve into advanced techniques and algorithms that amplify the potential of decision trees. Specifically, we will explore topics such as pruning, hyperparameter tuning, gradient boosting trees, and random forests. These advanced methodologies refine decision tree models, optimizing their performance and expanding their utility across complex data analysis and prediction tasks. Our discussion will provide comprehensive insights into these techniques, enabling you to harness the full capabilities of decision trees in your machine learning endeavors.

2.4.2.1 hyperparameter tuning in trees

In addition to the previously discussed hyperparameter tuning methods, decision tree-based algorithms offer several other methodologies that can be employed to ascertain the optimal hyperparameters, enabling the model to achieve its optimal performance [17].

- **Tree Depth and Minimum Samples per Leaf:**Decision trees have hyperparameters like maximum tree depth and minimum samples required to form a leaf node. A deeper tree may capture more complex relationships but could lead to overfitting. The minimum samples per leaf influences when to stop splitting nodes further. Tuning these hyperparameters helps find a balance between complexity and generalization.

- **Impurity Criteria:** The choice of impurity criteria (Gini impurity, entropy, etc.) affects the tree's splitting decisions. Different criteria can lead to different tree structures. Tuning the impurity criteria can impact model performance, especially when the data has specific characteristics.

- **Pruning Parameters:** Pruning hyperparameters control the amount of pruning applied to the tree after construction. They determine how aggressively the model eliminates unnecessary branches to avoid overfitting. Adjusting pruning parameters can significantly impact the tree's complexity and predictive performance.

2.4.2.2 Pruning

Pruning is a fundamental technique within the realm of decision tree construction [18], plays a pivotal role in achieving an intricate equilibrium between the complexity of a model and its ability to generalize well to unseen data. This strategic method involves the deliberate removal of specific branches or nodes from a decision tree's structure, effectively refining its architecture and mitigating the risk of capturing noise or anomalies inherent in the training data [19].

The impetus for engaging in pruning becomes particularly pronounced when decision trees exhibit symptoms of overfitting. Overfitting arises when a model becomes too finely tuned to the nuances of the training data, capturing not only the underlying patterns but also noise and outliers. This results in a suboptimal ability to generalize and predict accurately when faced with new, previously unseen instances. To identify overfitting, practitioners conduct meticulous comparisons of the model's performance on the training data against an independent validation or test dataset. Deviations between these contexts signal the presence of potential overfitting.

The process of pruning encompasses several stages, often following the "prune-then-test" methodology:

1. **Constructing the Full Tree:** The journey commences with the creation of a decision tree using the complete training dataset, allowing the tree to flourish unhindered.

2. **Iterative Pruning:** Beginning at the lowermost branches of the tree, the iterative process involves removing branches or nodes. Each removal prompts an evaluation of the validation dataset's performance. If the model's accuracy remains stable or improves with a node's elimination, it is pruned.

3. **Refining to Perfection:** This iterative cycle persists until further pruning no longer enhances validation performance. The final product is a pruned tree, embodying an equilibrium between complexity and predictive precision.

Pruning decision trees is accompanied by a host of advantages. Pruned trees exhibit an amplified ability to generalize, effectively discarding overly complex branches that tend to capture noise rather than genuine patterns. Their streamlined structure also fosters simplicity and understandability, making them accessible to a diverse audience. Additionally, pruned trees facilitate quicker inference, a valuable asset in scenarios demanding real-time applications or the processing of extensive datasets.

However, prudent application of pruning is paramount. Factors to consider include selecting appropriate criteria for pruning, finding the optimal balance between complexity and simplicity to prevent underfitting, and recognizing that the relevance of pruning can differ in ensemble methods such as Random Forests.

Pruning in decision trees embodies the delicate art of harmonizing model intricacy and predictive prowess. By employing this technique judiciously, decision trees evolve into not only reliable and efficient tools but also interpretable models, equipped to tackle a wide spectrum of machine learning challenges [20].

2.4.2.3 Random Forest

The evolution of decision trees has given rise to a suite of advanced algorithms that harness their strengths while mitigating their limitations. This evolution has paved the way for powerful ensemble techniques, such as Random Forests, which address issues of bias, variance, and predictive accuracy through sophisticated aggregation strategies.

Random Forests, introduced by Leo Breiman in 2001 [21] are a cornerstone of ensemble learning that represent a profound evolution in the world of machine learning. By marrying the strengths of decision trees with ingenious techniques to mitigate their shortcomings, Random Forests have emerged as a robust and versatile approach for predictive modeling. This methodology's ability to tackle issues of overfitting, variance, and generalization makes it a staple in modern data analysis.

At the heart of Random Forests lies the principle of ensemble learning (the bagging principle involves training multiple instances of the same model on different subsets of the training data, sampled with replacement, and then combining their predictions to reduce overfitting and improve model stability, was also developed by Leo Breiman in 1996 [22]) where multiple decision trees collaborate to make collective predictions. The algorithm emphasizes two essential elements: bootstrapping and feature subsetting. By introducing randomness into the model-building process, Random Forests foster diversity among the constituent trees, thus enhancing the ensemble's robustness and predictive accuracy.

The process of constructing a Random Forest can be outlined as follows:

1. **Bootstrapping:** The concept of boosting was introduced by Robert Schapire and Yoram Singer in 1990 [23]. The AdaBoost (Adaptive Boosting) algorithm, developed by Freund and Schapire in 1995 [24]. Boosting randomly select subsets of the training data with replacement. Each subset, known as a "bootstrap sample," is used to train an individual decision tree.

2. **Feature Subsetting:** At each node split within a tree, randomly select a subset of features. This step introduces variability and prevents one feature from disproportionately influencing the tree's decisions.

3. **Build Decision Trees:** Construct multiple decision trees using the bootstrapped datasets and feature subsets. These trees are grown to their full extent without pruning.

4. **Aggregation of Predictions:** For classification tasks, the ensemble's prediction is determined through majority voting among the individual trees' predictions. For regression, the predictions are averaged.

5. **Reducing Variance:** The diversification introduced by bootstrapping and feature subsetting mitigates overfitting and reduces the variance that can plague single decision trees.

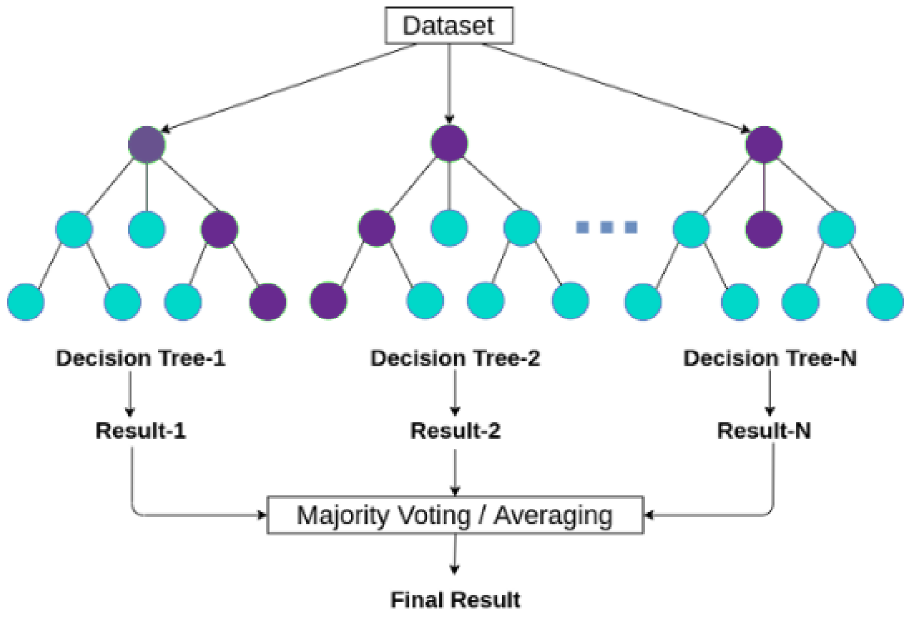


Figure 2.9: Random Forest Method

Formulae and Mechanisms:

- **Bootstrapping:** Given the training dataset with n instances, a random sample of size n is drawn with replacement. The resultant bootstrap sample serves as the training data for an individual decision tree.

$$\text{Bootstrap Sample Size} = n$$

- **Feature Subsetting:** At each split node of a tree, a random subset of features (m) is selected from the total features (p). This subset size (m) is typically smaller than p, introducing diversity and preventing excessive reliance on specific features.

$$m < p$$

The true strength of Random Forests emanates from their diversity-driven approach. By constructing an ensemble of trees that differ in training data and feature selection, the algorithm gains a remarkable ability to generalize well. This diversity shields the model from the overfitting that often plagues single decision trees, thereby endowing Random Forests with remarkable predictive accuracy.

Random Forests have revolutionized predictive modeling by overcoming the limitations of standalone decision trees. Their robustness, resistance to overfitting, and capacity to handle complex datasets make them a cornerstone in modern machine learning. This approach's applicability extends across various domains, encompassing classification, regression, feature selection, and more. As a testament to their effectiveness, Random Forests have found their place not only in research but also in numerous real-world applications, leaving an indelible mark on the landscape of machine learning and data analysis.

2.4.2.4 Gradient Boosting Trees

Gradient Boosting Trees, a pinnacle of machine learning ingenuity, have ushered in a new era of predictive prowess. By harnessing the strength of decision trees while deftly circumventing their limitations, Gradient Boosting Trees have emerged as a robust technique for predictive modeling. Its innate ability to address overfitting, bias, and accuracy imbues it with exceptional versatility and has firmly established it as a cornerstone of modern data analysis. The Gradient Boosting algorithm, including Gradient Boosting Trees, was introduced by Jerome H. Friedman, Trevor Hastie, and Robert Tibshirani. Specifically, the Gradient Boosting algorithm was outlined in 2001 [25]

At the core of Gradient Boosting Trees lies a meticulous process of iterative refinement. This approach is grounded in the amalgamation of multiple decision trees, with each subsequent tree endeavoring to correct the deficiencies of its predecessors. The process begins with a base model, often a simple decision tree. The subsequent trees are then tailored to target the residual errors or inaccuracies of the previous models, gradually enhancing the overall predictive performance of the ensemble.

Central to the technique is the concept of gradient descent, a method for optimizing a loss function by iteratively adjusting model predictions to minimize the loss. The loss function quantifies the deviation between predicted and actual values. In the context of Gradient Boosting, the objective is to systematically reduce this loss by training new trees to capture the residual errors of the ensemble's existing predictions.

The process of constructing Gradient Boosting Trees can be outlined as follows:

1. **Base Model Initialization:** Begin with a base model, often a simple decision tree.
2. **Computing Residuals:** Calculate the residuals by subtracting the predictions of the base model from the actual target values.

3. **Building a New Tree:** Construct a new decision tree with the primary goal of predicting the residuals, thereby focusing on capturing the remaining patterns in the data.
4. **Updating Ensemble Predictions:** Adjust the ensemble's predictions by aggregating the predictions of the newly constructed tree, scaled by a learning rate that controls the contribution of each tree.
5. **Iterative Refinement:** The process is reiterated, with each new tree fine-tuning the ensemble's predictions to rectify the residual errors. This iterative process progressively enhances predictive performance.

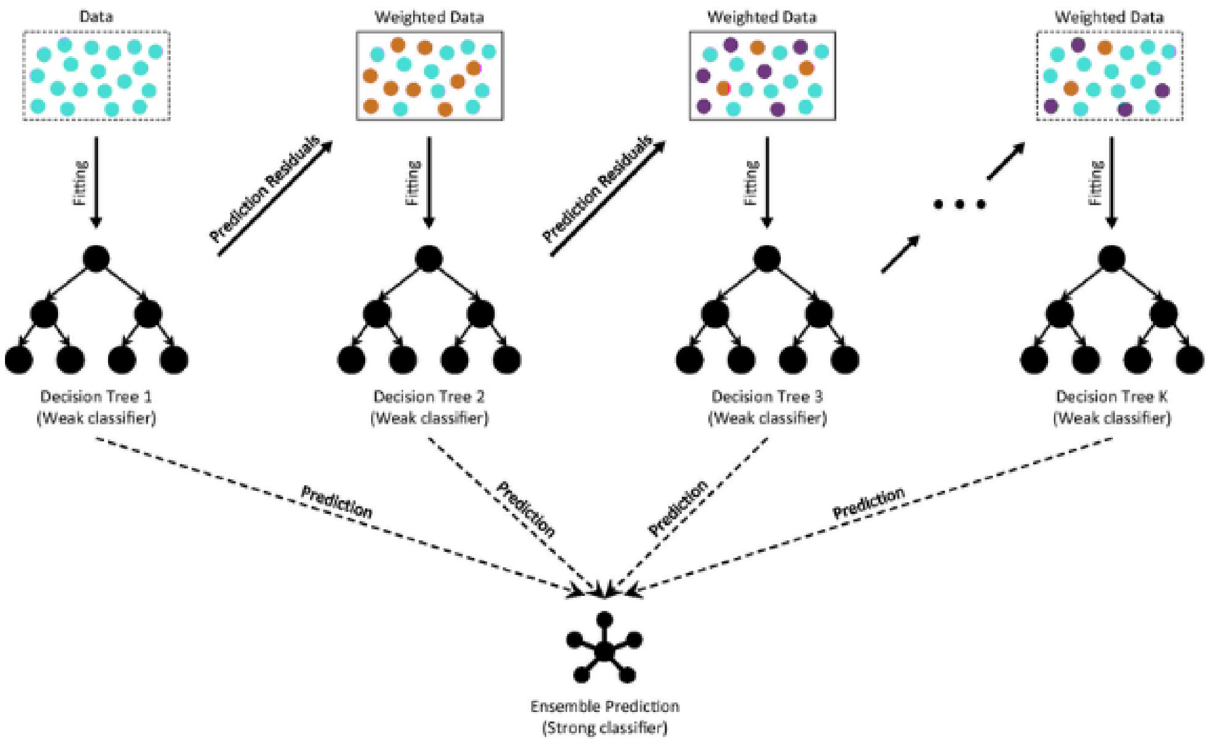


Figure 2.10: Gradient Boosting Tree

Gradient Boosting Trees effectively guard against overfitting through regularization mechanisms. The learning rate (often denoted as η) is a critical parameter that determines the magnitude of updates applied to the ensemble's predictions by each new tree. Smaller learning rates facilitate gradual convergence and diminish the risk of overfitting, while larger rates expedite learning at the expense of potential overshooting.

The strength of Gradient Boosting Trees lies in the amalgamation of individual models into a potent ensemble. Each subsequent tree endeavors to rectify the limitations of its predecessors, culminating in enhanced predictive accuracy and resilience against overfitting. The ensemble benefits from diverse trees, each capturing distinct data patterns.

Formulae used in Gradient Boosting Trees

1. **Loss Function:** A commonly used loss function is the Mean Squared Error (MSE), which quantifies the squared deviations between predicted and actual values:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.12}$$

2. **Residuals:** Residuals are calculated as the differences between actual target values (y_i) and the base model's predictions ($\hat{y}_i$):

$$\text{Residuals} = y_i - \hat{y}_i \tag{2.13}$$

3. **Gradient Descent Update:** The update rule for adjusting predictions based on gradient descent direction (∇) and learning rate (η):

$$\text{New Prediction} = \text{Old Prediction} + \eta \cdot \nabla \tag{2.14}$$

Gradient Boosting Trees have revolutionized machine learning by combining the strengths of decision trees with an iterative refinement process. This approach empowers the algorithm to produce highly accurate models that transcend the limitations of individual decision trees. The technique's adaptability extends across diverse domains, including classification, regression, and ranking tasks, making it a fundamental tool in predictive modeling. Through meticulous optimization, Gradient Boosting Trees have solidified their place as a potent technique that continues to drive breakthroughs in data analysis, research, and industry applications.

2.4.3 Neural Networks

In the realm of artificial intelligence, neural networks stand as a cornerstone, mimicking the intricate neural connections of the human brain to tackle complex tasks. These networks have redefined the boundaries of machine learning and have driven remarkable advancements in areas like image recognition, natural language processing, and autonomous systems. This section delves into the inner workings of neural networks, shedding light on their architecture, functioning, and mathematical foundations.

The base of neural networks are perceptron, which are explained in full detail in 3.4.

2.4.3.1 Multi-Neuron Perceptrons (MLPs)

While single perceptrons can make simple decisions, they are limited in their ability to handle complex tasks that require more intricate decision boundaries. Multi-neuron perceptrons, also known as multi-layer perceptrons (MLPs), are designed to overcome this limitation by stacking multiple perceptrons together in layers.

A Multi-Layer Perceptron (MLP) [26] is a type of artificial neural network that consists of multiple layers of interconnected neurons where each neuron in one layer is connected to every neuron in the subsequent layer. This connectivity allows for the propagation of information through the network. It's a fundamental architecture in deep learning and is used for a wide range of tasks, including classification, regression, and more.

1. **Input Layer:** The input layer is the first layer of the MLP and serves as the entry point for the data that the network processes. Each neuron in the input layer corresponds to a feature or attribute of the input data. The number of neurons in the input layer is determined by the dimensionality of the input data. For example, if you're working with images, each pixel might be considered a feature, so the number of neurons in the input layer would be equal to the number of pixels in the image.

2. **Hidden Layers:** Hidden layers are the layers that come after the input layer in an MLP. These layers are called "hidden" because they are not directly exposed to the input data or the final output. Hidden layers are responsible for learning and extracting complex patterns and representations from the input data. An MLP can have one or more hidden layers, each containing multiple neurons.

   The neurons in the hidden layers apply weighted sums of inputs, biases, and activation functions to produce intermediate representations of the input data. These intermediate representations become increasingly abstract and meaningful as you move deeper into the hidden layers. The number of neurons in each hidden layer and the total number of hidden layers are design choices that depend on the complexity of the problem and the available resources.

3. **Output Layer:** The output layer is the final layer of the MLP and produces the network's prediction or output. The number of neurons in the output layer depends on the type of task the network is designed to solve. For example:

   - In a binary classification task, there would be one neuron in the output layer, representing the probability of belonging to one class.
   - In a multi-class classification task, the number of neurons in the output layer matches the number of classes, and each neuron represents the probability of the input belonging to a specific class.

- In a regression task, there would be one neuron in the output layer, producing a continuous numeric value.
- The activation function used in the output layer depends on the task. For binary classification, a sigmoid or softmax activation function might be used. For regression tasks, a linear activation function might be used.

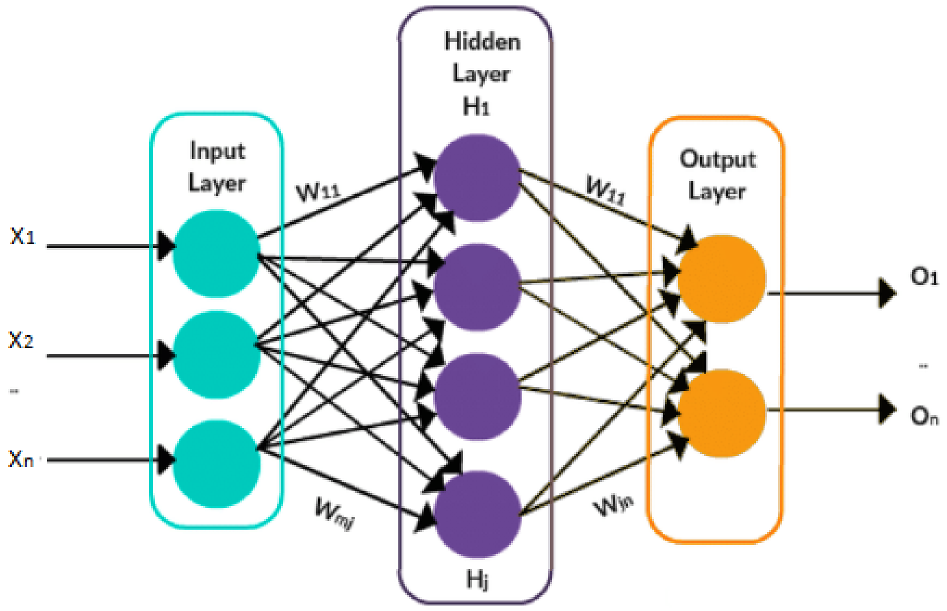


Figure 2.11: MLPs Structure

Forward propagation is the process by which input data is fed through the layers of an MLP to produce an output prediction. It involves a series of calculations that transform the input data through the network's weights, biases, and activation functions to generate the final prediction. These are the steps involved in forward propagation:

1. Forward propagation begins with the input data. Each data point is represented as a vector, and these vectors are stacked together to form a mini-batch, which is a subset of the entire dataset processed simultaneously. The input data is fed into the input layer of the MLP. Each neuron in the input layer corresponds to a feature in the input data. The input neurons don't perform any computation; they just pass the input data to the neurons in the next layer.

2. For each hidden layer each neuron in the current hidden layer calculates a weighted sum of the outputs from the previous layer. The weights associated with the connections from the previous layer's neurons are multiplied by the neuron outputs, and these products are summed up. Additionally, the bias term for each neuron is added to the sum. The weighted sum is then passed through an activation function. The activation function introduces non-linearity to the network and determines the output of each neuron. The output of this layer becomes the input for the next hidden layer, and the process repeats.

3. The process of weighted sum and activation continues in the last hidden layer until you reach the output layer. The output layer's neurons compute the final activations, which represent the network's prediction. The activations of the neurons in the output layer are the final predictions generated by the network. The specific interpretation of these predictions depends on the task the

network is solving. For example, in a binary classification task, the output might represent the probability of belonging to one class. In a regression task, the output might be a numeric value.

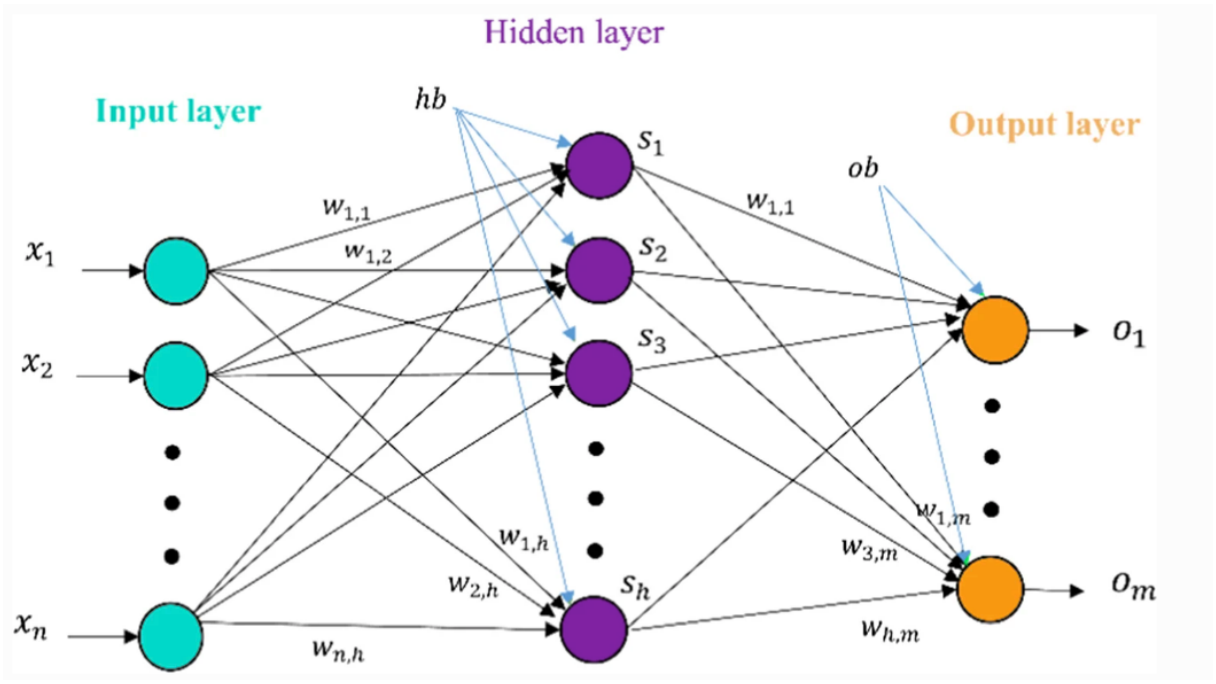


Figure 2.12: MLPs Forward propagation

This process transforms the raw input data into a prediction by leveraging the network's learned weights, biases, and activation functions. It's a fundamental step in training and using neural networks for various machine learning tasks.

Training an MLP involves adjusting the weights and biases of the entire network to minimize the error between predicted outputs and true outputs. This process is typically done using gradient-based optimization algorithms, mainly backpropagation.

Backpropagation [27] is a fundamental process used in training Multi-Layer Perceptrons and other neural network architectures. It's a method for adjusting the weights and biases of the network to minimize the difference between the predicted output and the actual target output. Let's see a detailed explanation of backpropagation process.

1. **Loss Function:** The first step in backpropagation is defining a loss function. This function quantifies the difference between the predicted output and the actual target output. The goal of training is to minimize this loss [28].

2. **Gradient Descent:** Like in Gradient Boosting Trees, Gradient Descent is performed. Backpropagation involves the use of gradient descent to update the weights and biases of the network. Gradient descent is an optimization algorithm that iteratively adjusts the model parameters in the direction that reduces the loss. The gradient of the loss function with respect to the weights and biases tells us how much and in what direction to adjust these parameters.

3. **Backward Pass:** The backward pass is the core of backpropagation. It's the process of calculating the gradients of the loss function with respect to the weights and biases, layer by layer, starting from the output layer and moving backward towards the input layer [29]. The gradient of the loss with respect to the output of the output layer is calculated first. This gradient is then used to

compute the gradients of the weights and biases in the output layer. The process is repeated for each hidden layer. The gradient at a hidden layer depends on the gradients from the subsequent layer and the weights connecting the layers.

4. **Weight and Bias Updates:** With the gradients calculated, the network's weights and biases are updated using the gradient descent algorithm. The updates are proportional to the negative gradient and a learning rate hyperparameter. This step aims to adjust the parameters to reduce the loss.

5. **Iteration:** Backpropagation involves multiple iterations (epochs) of forward and backward passes. In each iteration, the model makes predictions, calculates the loss, computes gradients, updates weights and biases, and repeats the process for a predefined number of times or until the loss converges to a satisfactory level.

6. **Hyperparameters:** Successful backpropagation involves tuning several hyperparameters, including the learning rate, the choice of activation functions, the number of hidden layers, the number of neurons in each layer, and more. These choices impact the convergence and effectiveness of the training process.

Backpropagation is a crucial process for training neural networks, enabling them to learn from data and make accurate predictions. It's the foundation of most modern deep learning algorithms.

Activation Functions in Hidden Layers [30]. Unlike the binary step function often used in single perceptrons, MLPs can use a variety of activation functions in their hidden layers. All these functions introduce non-linearity, allowing the network to model complex relationships in data. In all these activation functions z represents the weighted sum of inputs and bias. Given the inputs $x_1, x_2, \ldots, x_n$ and their associated weights $w_1, w_2, \ldots, w_n$, as well as the bias b, the weighted sum z is calculated as:

$$z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

- **Sigmoid:** Maps the weighted sum of inputs plus bias to a value between 0 and 1 2.13, providing a smooth activation response. The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.15}$$

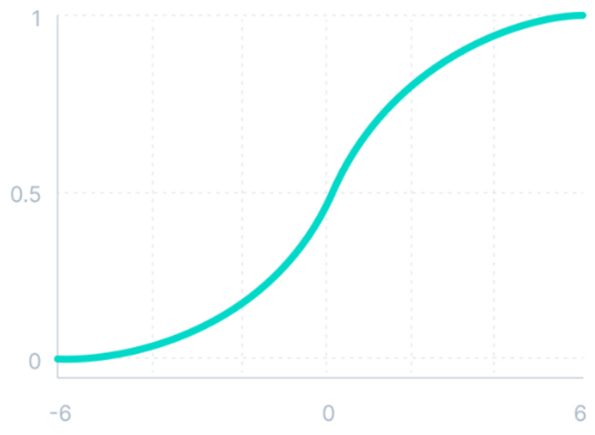


Figure 2.13: Sigmoid Activation Function

- **Hyperbolic Tangent (tanh):** It maps the weighted sum of inputs plus bias to a value between -1 and 1 2.14, providing a smooth activation response. The tanh function is defined as:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.16}$$

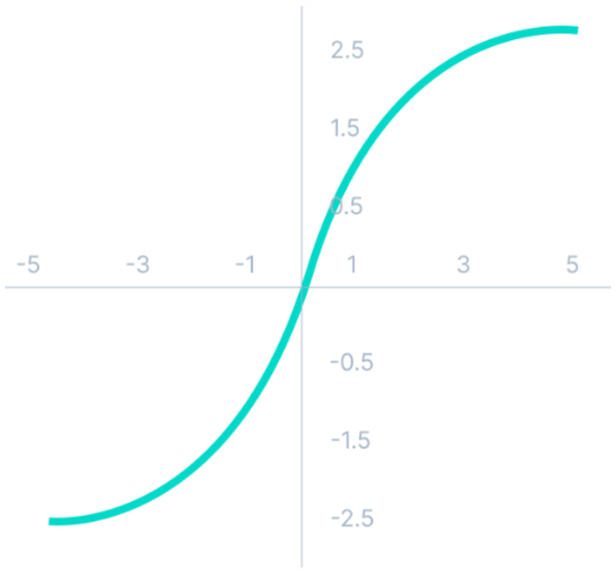


Figure 2.14: Tanh Activation Function

- **The Rectified Linear Unit (ReLU):** Maps the weighted sum of inputs plus bias to the output directly if it's positive, and otherwise, it outputs zero 2.15. The ReLU function is defined as:

$$\text{ReLU}(z) = \max(0, z) \tag{2.17}$$

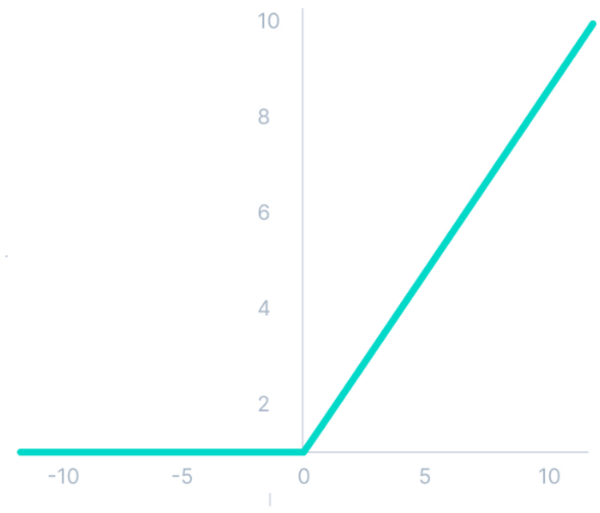


Figure 2.15: ReLu Activation Function

The result of applying the ReLU activation function is the output of the neuron. If the weighted sum z is positive, the neuron fires and produces z as the output. If z is negative, the neuron outputs zero.

- **Softplus:** The Softplus activation function produces a smooth output that resembles a smoothed version of the ReLU function 2.16. The Softplus function is defined as:

$$\text{Softplus}(z) = \ln(1 + e^z) \tag{2.18}$$

The result of applying the Softplus activation function is a positive value that increases as z becomes more positive. It introduces smoothness in the network's response, allowing it to capture complex patterns in the data.

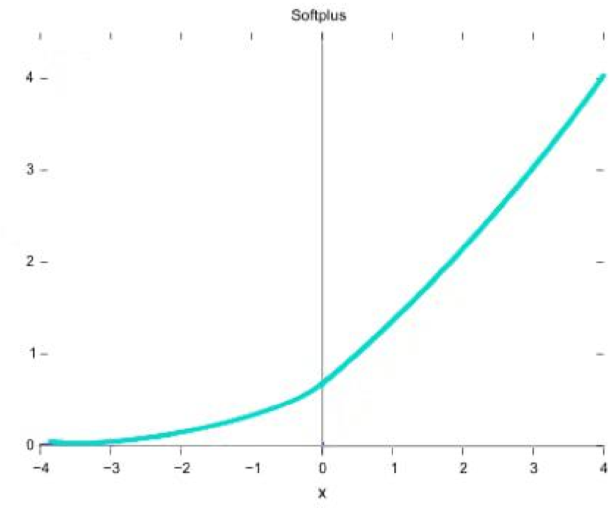


Figure 2.16: Softplus Activation Function

- **Linear:** The linear activation function is used in neural networks for specific purposes 2.17, such as regression tasks. It simply passes the weighted sum of inputs plus bias as the output without any transformation. The linear function is defined as:

$$\text{Linear}(z) = z \tag{2.19}$$

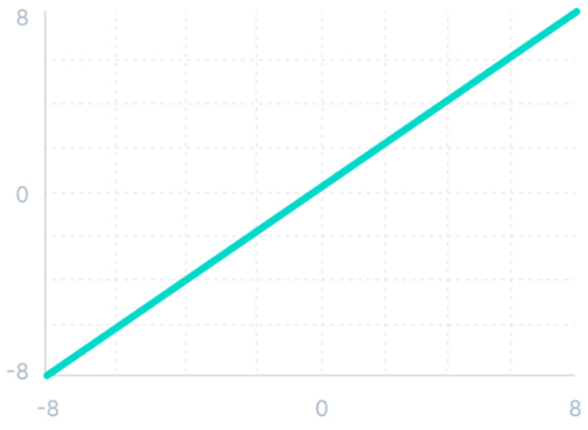


Figure 2.17: Linear Activation Function

- **Unit Step:** The unit step activation function, also known as the step function, is used in neural networks for binary classification tasks. It produces an output of 1 if the weighted sum of inputs plus bias is greater than or equal to 0, and an output of 0 otherwise 2.18. The step function is defined as:

$$\text{Step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.20}$$

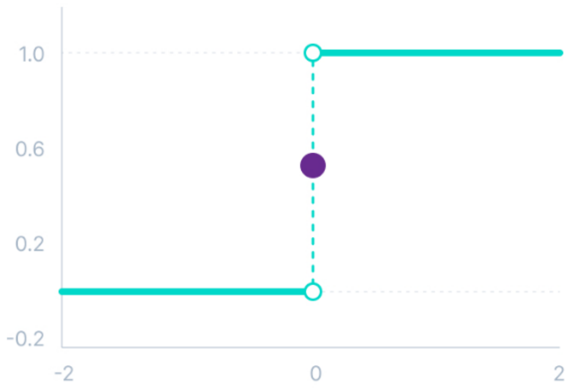


Figure 2.18: Unit Step Activation Function

The output of the neuron is either 1 or 0, depending on the value of the weighted sum z. The unit step function is often used in binary classification tasks.

- **Sign:** The sign activation function is used in neural networks to produce an output of 1 if the weighted sum of inputs plus bias is greater than 0, an output of -1 if the weighted sum is less than 0, and an output of 0 if the weighted sum is exactly 0. The sign function is defined as:

$$\text{Sign}(z) = \begin{cases} 1 & \text{if } z > 0 \\ -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \end{cases} \tag{2.21}$$

The output of the neuron is either 1, -1, or 0, depending on the value of the weighted sum z. The sign function is often used in situations where the network needs to make decisions based on the direction of the input signal.

- **Piece-Wise Linear:** The piece-wise linear activation function is used in neural networks to introduce non-linearity through different linear segments. It consists of multiple linear equations for different ranges of the input.

$$f(z) = \begin{cases} m_1 z + c_1 & \text{if } z < a_1 \\ m_2 z + c_2 & \text{if } a_1 \leq z < a_2 \\ \vdots & \vdots \\ m_n z + c_n & \text{if } a_{n-1} \leq z \end{cases} \tag{2.22}$$

Here, $a_1, a_2, \ldots, a_n$ are the points where the linear segments change, and $m_1, m_2, \ldots, m_n$ and $c_1, c_2, \ldots, c_n$ are the slopes and intercepts for each segment. The output Introduces non-linearity through different linear segments with varying slopes and intercepts.

- **Scaled Exponential Linear Unit (Selu)** The Scaled Exponential Linear Unit (SELU) activation function is a self-normalizing activation function that has gained popularity in the field of deep learning. It addresses the **vanishing and exploding gradient problems** associated with other activation functions like sigmoid, tanh, and ReLU. SELU offers the advantage of maintaining mean activation close to zero and standard deviation close to one, leading to more stable and effective training of deep neural networks.

$$f(x) = \begin{cases} \lambda\,(x) & \text{if } x > 0 \\ \alpha\,(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{2.23}$$

where λ is a scaling constant, typically set to 1.0507 to ensure that the mean and variance of the activations remain close to 0 and 1 during forward propagation 2.19.α is a negative scaling constant, typically set to 1.67326, which controls the value of the SELU function for $x \leq 0$

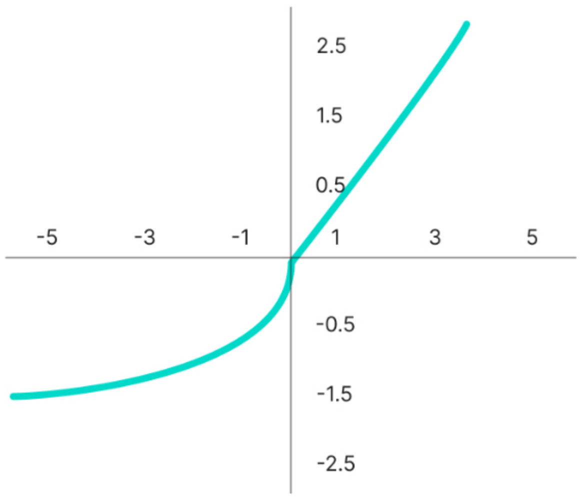


Figure 2.19: Selu Activation Function

- **Gaussian Error Linear Unit (GELU)** The Gaussian Error Linear Unit (GELU) activation function is a smooth and non-monotonic activation function that has shown promising performance in deep neural networks. It combines properties of both the ReLU and sigmoid activation functions, aiming to strike a balance between avoiding vanishing gradients and enabling efficient training 2.20. The GELU activation function is defined as follows:

$$f(x) = \frac{1}{2}x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right) \tag{2.24}$$

GELU has shown strong performance on a variety of tasks and benchmarks, making it a competitive alternative to traditional activation functions like ReLU and sigmoid. Its smoothness and balanced behavior contribute to improved convergence rates and better generalization capabilities.

- **Swish:** The Swish activation function is a novel activation function that has gained attention due to its promising performance in deep neural networks. It introduces a non-linearity that is smooth and differentiable, contributing to improved training while avoiding some of the limitations of traditional activation functions 2.21. The Swish activation function is defined as follows:

$$f(x) = x \cdot \sigma(\beta x) \tag{2.25}$$

where $\sigma(\cdot)$ is the sigmoid activation function and β controls the saturation behavior of the function.

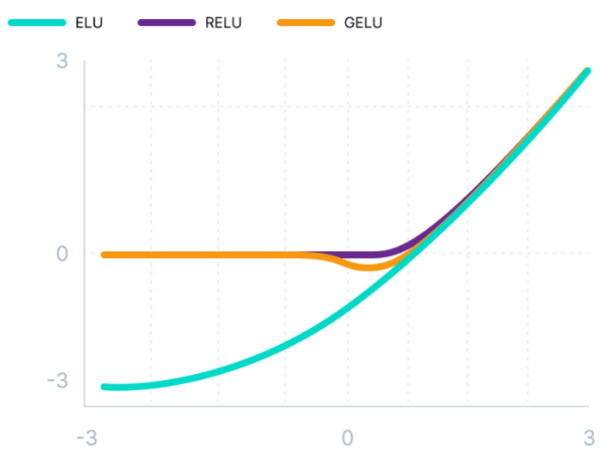


Figure 2.20: GELU Activation Function

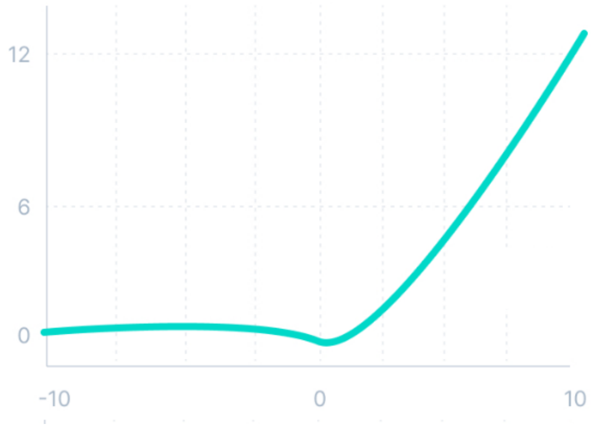


Figure 2.21: Swish Activation Function

Swish has demonstrated competitive performance on various tasks, making it a strong candidate for activation functions in deep networks. Its adaptiveness and smoothness contribute to stable convergence and better generalization.

The transition from a single perceptron to Multi-Layer Perceptrons (MLPs) brings several advantages, enabling neural networks to model and learn more complex relationships in data. Here are the key advantages of using MLPs over single perceptrons:

Single perceptrons are limited to learning linear relationships in data. In contrast, MLPs can learn non-linear relationships and complex patterns in data by incorporating multiple layers with non-linear activation functions. This allows them to model intricate relationships that linear models cannot capture.

MLPs with multiple hidden layers can learn to abstract and extract hierarchical features from the input data. Each hidden layer learns to represent more abstract and high-level features that are combinations of lower-level features. This ability to learn hierarchical representations is crucial for understanding complex data.

MLPs with one or more hidden layers are universal function approximators. This means that they can approximate any continuous function to a desired degree of accuracy, given enough neurons and appropriate weights. This property makes MLPs extremely versatile and capable of handling a wide range of tasks. MLPs can be designed with multiple hidden layers and varying numbers of neurons per layer. This flexibility allows them to adapt to different types of data and tasks. Deeper architectures allow for more complex feature learning and higher-level abstractions.

While single perceptrons are mainly suited for binary classification tasks, MLPs can be used for various tasks, including classification, regression, and even more complex tasks like image and speech recognition. The ability to model diverse relationships makes MLPs applicable to a wide array of problems.

In cases where data is not linearly separable, MLPs can learn complex decision boundaries that fit the data well. This makes them effective for tasks where the relationship between inputs and outputs is highly intricate.

MLPs are capable of learning rich internal representations of the input data, enabling them to capture underlying structures. This feature makes them highly suitable for tasks involving data with high dimensionality or complex interactions between features.

Multi Layer Perceptrons can benefit from regularization techniques such as **dropout, weight decay, and batch normalization**. These techniques help prevent overfitting, improve generalization, and enhance the model's ability to perform well on new, unseen data. They can also be combined to form ensemble models, where multiple MLPs work together to make predictions. Ensembles improve the overall performance by reducing variance and providing more robust predictions.

In summary, the advantages of transitioning from a single perceptron to Multi-Layer Perceptrons (MLPs) include the ability to model non-linear relationships, extract hierarchical features, handle diverse tasks, learn complex decision boundaries, and achieve better generalization. These advantages have contributed to the widespread use of MLPs in various domains of machine learning and deep learning.

Multi-neuron perceptrons, represent an evolution from single perceptrons, allowing neural networks to solve more complex problems. Their architecture, training procedures, and choice of activation functions contribute to their ability to capture and represent intricate data patterns, making them a foundational component of modern neural network systems.

Neural networks, built upon the foundation of multi-layer perceptrons, are at the forefront of modern machine learning. Their interconnected layers of neurons and sophisticated training algorithms enable them to model intricate relationships in data, making them a versatile tool for a wide range of tasks across various domains. Creating the known as Artificial Neural Networks (ANNs).

ANNs have ushered in a new era for supervised classification, redefining the boundaries of pattern recognition and decision-making. These networks, inspired by the intricate connections within the human brain, have brought unparalleled advancements to the field. ANNs excel at capturing complex relationships in data, transforming raw input into meaningful insights. With layers of interconnected neurons and adaptive weights, ANNs can learn to discriminate between classes with remarkable accuracy. From image recognition to natural language processing, their adaptability has revolutionized supervised classification, enabling machines to comprehend and classify intricate patterns and features that were previously challenging for traditional methods.

Within the realm of ANNs, a myriad of types exists, each tailored to different data structures and tasks. Multi-Layer Perceptrons (MLPs) thrive on structured data, deciphering intricate relationships through hidden layers and non-linear activations. Convolutional Neural Networks (CNNs), designed for images and grid-like data, detect spatial hierarchies and visual features. Meanwhile, Recurrent Neural Networks (RNNs) excel with sequential data, unveiling temporal dependencies in sequences. Specialized architectures such as Long Short-Term Memory (LSTM) networks bolster memory retention, crucial for tasks like language modeling. As the field advances, hybrid models merge these paradigms, tapping into the strengths of each architecture. The diversity of ANN types is a testament to the adaptability and innovation that underpin their transformative impact on supervised classification.

- **Convolutional Neural Networks (CNNs):** Convolutional Neural Networks (CNNs) [31], [32] represent a breakthrough in the realm of deep learning, fueling remarkable advancements in computer vision and image analysis. Built on the foundational principles of neural networks, CNNs have been meticulously tailored to process grid-like data, making them particularly adept at understanding visual information. Their architecture is inspired by the human visual system, employing layers of convolutional, pooling, and fully connected units that mimic the process of feature extraction in the brain.

  At the heart of CNNs lies the convolutional layer, which operates by sliding small filters over the input data to capture local patterns, edges, and textures. This hierarchical approach empowers CNNs to automatically learn meaningful visual features without explicit feature engineering. Pooling layers then downsample the spatial dimensions, consolidating the most pertinent information while reducing computational complexity. The integration of non-linear activation functions, like ReLU, imparts CNNs with the ability to capture complex relationships within the data.

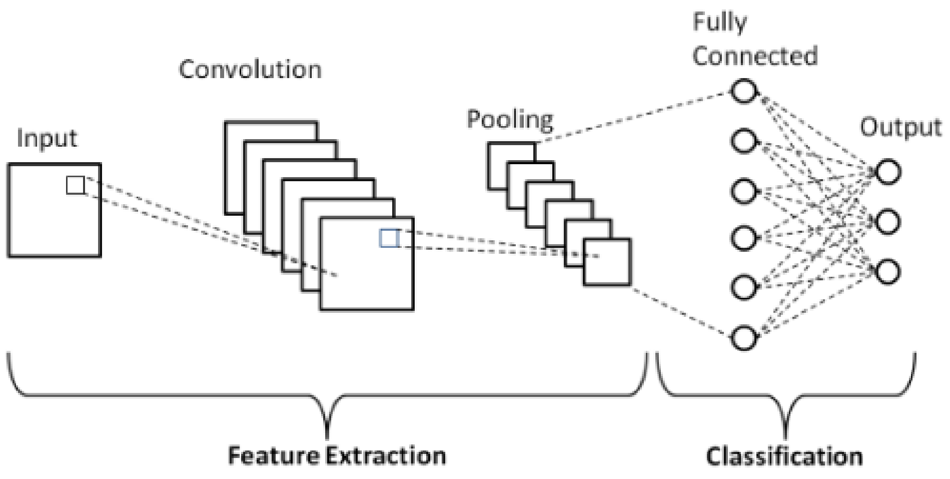


Figure 2.22: CNNs Structure

  CNNs have revolutionized numerous computer vision tasks, including image classification, object detection, image segmentation, and more. Their prowess lies in their capacity to detect spatial hierarchies and identify intricate patterns, regardless of their location within an image. This adaptability and efficiency have propelled CNNs into the forefront of modern technology, playing an integral role in applications ranging from self-driving cars to medical image analysis, and opening the door to previously unattainable feats in understanding and interpreting visual data.

- **Recurrent Neural Networks (RNNs):** Recurrent Neural Networks (RNNs) [33] ,mainly developed by Geoffrey Hinton and Ronald J. Williams [27], stand as a foundational innovation in deep learning, particularly tailored for tasks involving sequential data analysis. Unlike traditional feedforward neural networks, RNNs introduce a dynamic feedback mechanism that allows them to retain information from previous time steps, enabling them to capture temporal dependencies in sequences.

  The essence of RNNs lies in their ability to maintain hidden states that store contextual information as data sequences unfold. Each time step processes input along with the hidden state, effectively encoding the sequence's history. This intrinsic memory capability makes RNNs exceptionally adept at tasks such as language modeling, time series prediction, and speech recognition.

However, traditional RNNs face the challenge of the vanishing gradient problem, where information diminishes over long sequences due to repeated multiplicative interactions. To address this, advanced architectures like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) were introduced. These architectures incorporate gating mechanisms that regulate the flow of information, allowing relevant information to persist over longer distances and mitigating the vanishing gradient issue.

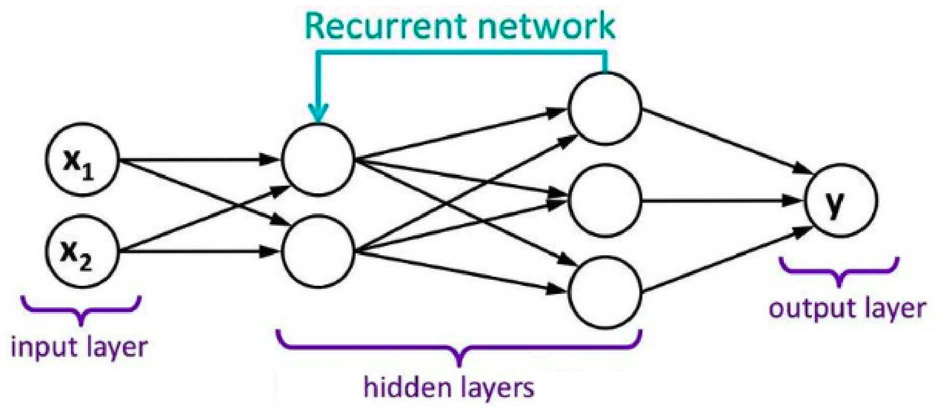


Figure 2.23: RNNs Structure

RNNs have brought transformative capabilities to a multitude of domains, from natural language processing to music generation. Their ability to understand and model sequences has unlocked new levels of sophistication, empowering machines to grasp intricate temporal patterns and revolutionizing applications ranging from chatbots that understand context to algorithms that predict stock market trends.

- **Long Short-Term Memory (LSTM):** Long Short-Term Memory networks (LSTMs), first introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997 [34], stand as a pivotal advancement within the realm of Recurrent Neural Networks (RNNs), addressing one of the most significant challenges: the vanishing gradient problem. LSTMs introduce a sophisticated memory mechanism that allows them to retain and process information over extended sequences, making them exceptionally suited for tasks involving long-term dependencies.

  At the core of LSTMs are memory cells equipped with gating units that control the flow of information. These gates, including the input, forget, and output gates, enable LSTMs to regulate the flow of information, selectively retaining or discarding relevant information at each time step. This mechanism empowers LSTMs to capture and preserve crucial context throughout sequences, ensuring that distant relationships are not lost due to vanishing gradients.

  LSTMs have revolutionized various fields, from natural language processing to speech recognition. Their ability to remember and recall intricate patterns within sequences has propelled them to the forefront of deep learning. By overcoming the limitations of traditional RNNs, LSTMs have unlocked the potential to tackle complex tasks such as language translation, sentiment analysis, and even generating creative content like poetry or music.

- **Generative Adversarial Networks (GANs):** Generative Adversarial Networks (GANs) have emerged as a groundbreaking paradigm in deep learning, revolutionizing the field of generative modeling and creative content creation. Conceived by Ian Goodfellow in 2014 [35], where Yoshua Bengio's contributions played a key role, GANs introduce an innovative framework where two neural networks—the generator and the discriminator—engage in a captivating adversarial dance.

The generator aims to fabricate data samples that closely resemble real data, while the discriminator endeavors to differentiate between genuine and generated data. This adversarial interplay leads to a dynamic equilibrium, where the generator constantly refines its skill in producing increasingly convincing data, and the discriminator hones its ability to make more precise judgments.

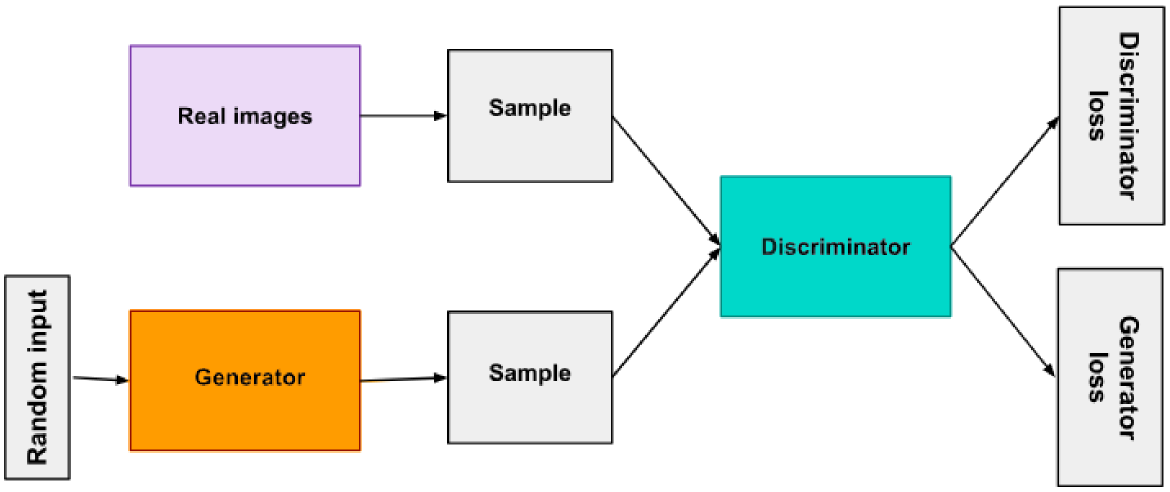


Figure 2.24: GANs Structure

The hallmark of GANs is their ability to create remarkably realistic and high-quality data samples, from images and music to text and videos. By learning from raw data distributions, GANs excel in mimicking the intricate patterns, textures, and nuances present in the training data. Their applications span a wide spectrum, ranging from image synthesis, where GANs craft lifelike images of non-existent faces or scenes, to style transfer, which transforms images to resemble artistic styles or different visual domains. Despite their unparalleled success, GANs present challenges such as training instability and mode collapse, where the generator produces limited diversity in generated outputs. These challenges continue to drive research, inspiring innovations that further propel GANs into the vanguard of AI creativity and generative modeling.

Lets see a small example:

When training begins, the generator produces obviously fake data, and the discriminator quickly learns that it's false.

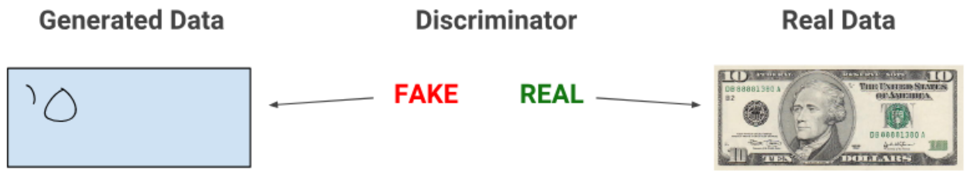


Figure 2.25: GANs 1'st aproximation

As training progresses, the generator approaches the outcome that can deceive the discriminator.

Finally, if the training of the generator is successful, the discriminator deteriorates in indicating the difference between real and fake. It starts classifying fake data as real and its accuracy decreases.

- **Autoencoders:** Autoencoders, a foundational concept in unsupervised learning, have bestowed the field of neural networks with the power to unveil hidden patterns and reduce data dimension-

Figure 2.26: GANs 2'nd aproximation

Figure 2.27: GANs n'th aproximation

ality. Comprising an encoder and a decoder, an autoencoder seeks to compress input data into a lower-dimensional latent space representation and then reconstruct it with minimal loss. The encoder compresses the input into a compact form, often referred to as a bottleneck layer, where critical features are preserved, and the decoder endeavors to reconstruct the original input from this compressed representation.

Autoencoders hold versatile applications, from denoising noisy images to feature learning for downstream tasks. Denoising autoencoders, introduced by Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol [36], act as robust feature extractors by learning to recover clean data from corrupted samples, making them adept at handling noisy data. Variational Autoencoders (VAEs), introduced by Diederik P. Kingma and Max Welling in 2013 [37] delve deeper, imbuing generative properties by modeling the latent space distribution and enabling the generation of novel data samples. Autoencoders play a pivotal role in dimensionality reduction, anomaly detection, and even data compression. Their underlying mechanism, characterized by the interplay of compression and reconstruction, has fostered a multitude of variations and applications, demonstrating their indispensability in modern machine learning paradigms.

- **Transformers:** Transformers have revolutionized natural language processing and beyond, introducing a novel architecture that excels in capturing contextual relationships and dependencies within sequential data. Developed by Vaswani et al in 2017 [38], the transformer architecture centers around self-attention mechanisms, enabling the network to weigh the significance of different words in a sentence based on their contextual relevance. This departure from traditional recurrent and convolutional models has unlocked unprecedented accuracy and efficiency in processing sequences.

  The heart of transformers lies in the self-attention mechanism, where each word interacts with every other word in the sequence, capturing long-range dependencies and contextual nuances. The introduction of multi-head attention further empowers the model to focus on different parts of the input, enabling it to discern various aspects simultaneously. Transformers have become the cornerstone of modern natural language processing, driving progress in machine translation, text generation, sentiment analysis, and question answering. Moreover, their capabilities extend beyond language, as transformers are applied to image generation, protein folding, and even video analysis. While their computational demands can be substantial, transformers' exceptional performance and adaptability continue to reshape the landscape of deep learning.

Chapter 3

Development

3.1 Introduction

In the upcoming section, we will delve deeper into describing the intricate process of developing the project. The central aim here is to meticulously program algorithms for supervised classification, ensuring a lucid comprehension of their inner workings. These algorithms have been deliberately designed with simplicity in mind, serving as didactic tools for educational purposes. However, caution should be exercised when contemplating their utilization with substantial datasets or in the realm of actual projects. Despite the algorithms demonstrating correct functionality and theoretically feasible implementation, it's crucial to recognize that there exist alternative methodologies capable of accomplishing this task with superior efficiency in computational and temporal aspects.

It's worth noting that the foundation of all these algorithms rests predominantly on the theoretical framework covered in preceding sections. Consequently, a thorough familiarity with this background is strongly recommended. Practical examples will also be included to illustrate their functioning in simplified real-world scenarios. The manner of implementation deviates slightly from the aforementioned approach, as certain tailored adjustments and simplifications have been introduced. This strategic adaptation has been introduced to facilitate a clearer visualization of outcomes and to engender a more profound understanding of the methodological approach. Furthermore, it's pivotal to note that some of these algorithms aren't exclusively engineered to achieve data classification goals. Rather, they offer a pathway to apprehending the construction of data structures that will subsequently form the bedrock of the classification process.

For each of these algorithms, a detailed presentation of the accompanying code and a conceptual explanation of their computational mechanics will be provided. It's crucial to emphasize that to truly grasp the nuances of each algorithm, it's highly recommended to engage in a thorough examination of the code snippets. The complexity of these algorithms can pose a challenge to comprehension, given the intricate nature of some of the implemented processes. Thus, dedicating time to scrutinizing the code, using the package and understanding its intricate components will be needed to fully comprehend the algorithm's inner workings and the adequate way to utilize the visualization mechanisms.

Without further preamble, let's delve more deeply into each of the developed methods, examining them with greater scrutiny.

3.2 Regression

Regression is the study of dependence. It is used to answer questions as: Can we predict the eruption of a geyser based on the length of the recent eruptions? Do countries with higher per person income have lower birth rates than countries with lower income? Do changes in diet result in changes in cholesterol level, and if so, do the result depend on other characteristics such as age, sex, and exercise?

Regression analysis is a central part of many research projects. In this section we study the important instance of regression methodology, called linear regression. As with most statistical analyses, the goal of regression is to summarize observed data as simply, usefully and elegantly as possible.

3.2.1 Linear Regression

Linear regression is a fundamental statistical technique used to analyze and model the relationship between two or more variables. It provides a framework for understanding the dependency between a dependent variable (also called the target) and one or more independent variables (also known as predictors or features). Linear regression assumes that this relationship can be approximated by a linear equation, making it a powerful tool for prediction, inference, and understanding underlying patterns in the data.

3.2.1.1 Theoretical approach

The core concept of linear regression is to find the best-fitting line that minimizes the difference between the predicted values and the actual values in the dataset. This line, often referred to as the regression line or the best-fit line, represents the linear equation that describes the relationship between the variables. In simple linear regression, which deals with a single predictor variable, the equation takes the form:

$$y = \beta_0 + \beta_1 x + \varepsilon \tag{3.1}$$

Where:

- y is the dependent variable (target).
- x is the independent variable (predictor).
- β_0 is the y-intercept, representing the value of y when x is 0.
- β_1 is the coefficient of the independent variable, indicating the change in y for a unit change in x.
- ε represents the residual or error term, accounting for the variability that the model doesn't capture.

To represent all this data, the fundamental graphical tool used to look at regression data is the scatterplot. In regression problems with one predictor and one response, the scatterplot of the response versus the predictor is the starting point for regression analysis. In problems with many predictors, several simple grapsh will be required at the beginning of the analysis. A scatterplot matrix is a convenient way to organize looking at many scatterplots at once.

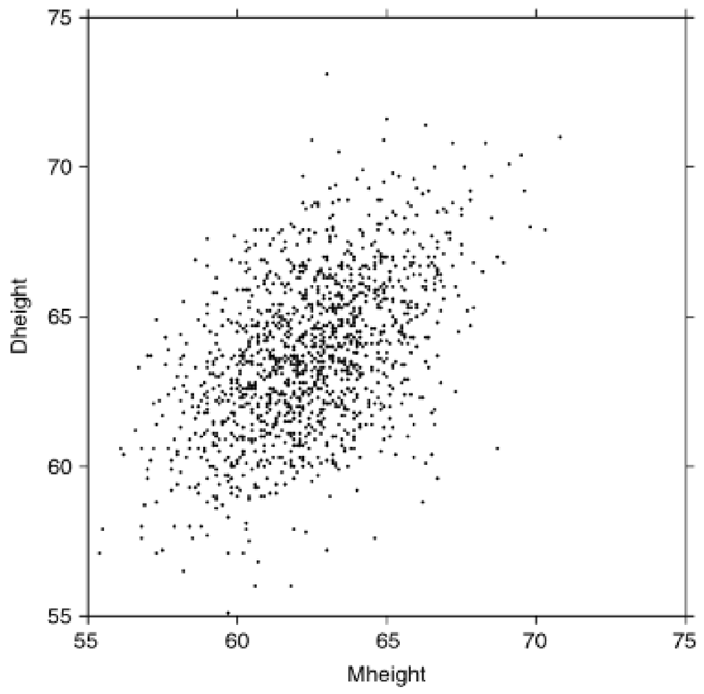


Figure 3.1: simple scatterplot

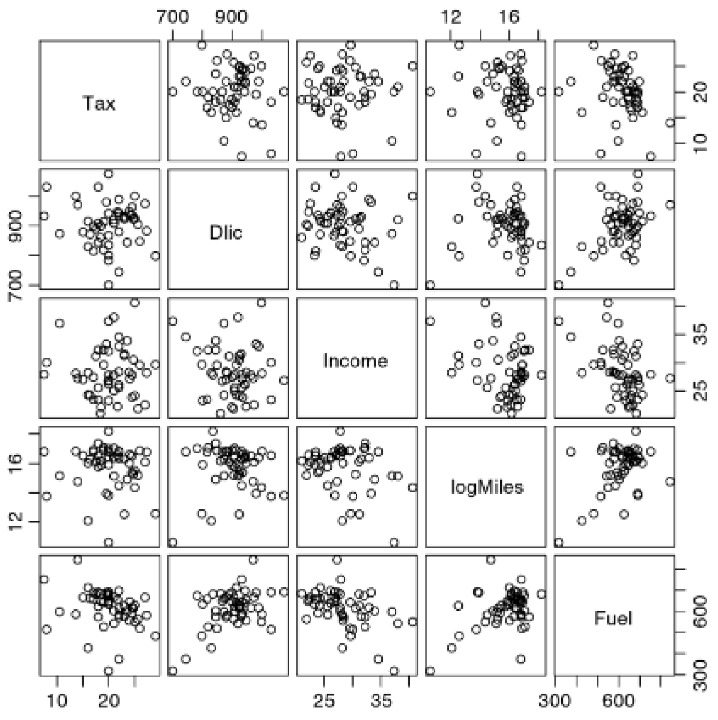


Figure 3.2: scatterplot matrix

Linear regression 3.3 stands as a cornerstone in the realm of statistical techniques, offering a systematic approach to model the intricate relationships between variables. At its core, it postulates a linear connection between the dependent variable (y) and one or more independent variables (x), facilitating the understanding of how variations in the independent variables impact the dependent variable. This technique serves as a fundamental building block for more sophisticated models and enjoys widespread adoption across diverse domains, including economics, social sciences, and machine learning.

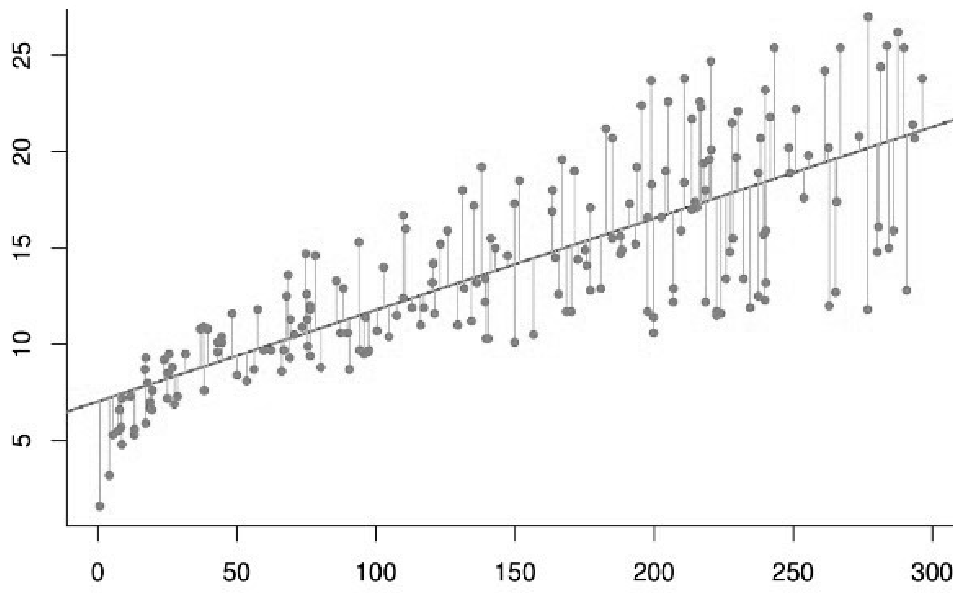


Figure 3.3: Linear Regression

In the realm of simple linear regression, the objective is to model the connection between a solitary independent variable (x) and the dependent variable (y). This relationship is encapsulated within the equation:

The efficacy of linear regression pivots on a set of assumptions, including the linearity of relationships, independence of errors, homoscedasticity (constant error variance), and the normal distribution of errors. The validation of these assumptions often involves the examination of residual plots and statistical tests. To gauge the adequacy of the model, metrics such as the coefficient of determination (R^2), adjusted R^2, and the standard error of the estimate are utilized.

The outcomes of linear regression are imbued with valuable insights. The coefficients $\beta_1, \beta_2, \ldots, \beta_p$ signify the change in the dependent variable for each unit change in the corresponding independent variable, all the while holding other variables constant. The p-values associated with these coefficients offer a measure of their statistical significance. Lower p-values indicate a stronger likelihood that the respective independent variable significantly influences the dependent variable.

Linear regression is a widely used statistical method with several notable advantages. Its simplicity and interpretability make it an ideal choice for introductory modeling and for situations where a clear understanding of the relationship between variables is essential. Linear regression's efficiency and ability to handle large datasets with numerous features are additional strengths, making it a practical choice in various fields. Furthermore, it can be robust in the face of minor violations of its assumptions, such as the independence of errors. However, it also has limitations, primarily stemming from its linearity assumption. When the true relationship between variables is nonlinear or complex, linear regression may perform poorly, leading to underfitting. It is also sensitive to outliers, with a single extreme data point significantly affecting model results. Additionally, when independent variables are highly correlated, multicollinearity can lead to unstable coefficient estimates.

In conclusion, linear regression is an elegant and robust technique, furnishes a comprehensive framework for deciphering relationships within data. It bestows practitioners with the power to untangle the intricate web of interactions between variables, thereby unveiling meaningful insights. While simple and multiple linear regressions might falter in capturing complex nonlinear associations, their simplicity, interpretability, and efficacy continue to render them invaluable tools for both novice and seasoned data analysts.

3.2.1.2 Coded implementation

The function facilitates simple linear regression analysis. This analysis aims to uncover a linear relationship between two or more variables and visualize the resulting regression lines, along with relevant statistics. The function expects a dataset as input, with the first n columns representing the independent variables (x) and the last column representing the dependent variable (y).

```
multivariate_linear_regression <- function(data, details = FALSE, waiting = TRUE) {
  oldpar <- par(no.readonly = TRUE)
  on.exit(par(oldpar))

  num_columns <- ncol(data)

  if(details){
    console.log("\nEXPLANATION (for each independent variable)")
    hline()
    hline()
    console.log("\nStep 1:")
    console.log("    - Calculate mean of the dependent and independet variables.")
    console.log("    - Calculate covariance and the variance of the dependent variable.")
    console.log("      If covariance = 0, print error message.")
    console.log("Step 2:")
    console.log("    - Calculate the intercept and the slope of the equation.")
    console.log("Step 3:")
    console.log("    - Calculate the sum of squared residuals and the sum of squared
    deviations")
    console.log("      of the independent variable.")
    console.log("    - Calculate the coefficient of determination.")
    console.log("Step 4:")
    console.log("    - Plot the line equation\n")
    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
    hline()
    hline()

    par(mfrow = c(1, 1))
    console.log("\nAn empty plot is created with appropiate limits\n\n")
    plot(1, type = "n", xlim = range(data[, num_columns]),
         ylim = range(data[, 1:(num_columns - 1)]),
         main = "Multivariate Linear Regression",
         xlab = colnames(data)[num_columns],  # Use dependent variable as x-axis label
         ylab = "Variables")  # Use "Variables" as y-axis label

    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  # Initialize empty vectors for legends
  legend_labels <- character(num_columns - 1)
  legend_colors <- integer(num_columns - 1)
  variable_names <- character(num_columns - 1)
```

```

reg_params <- list(num_columns - 1)

dependent_var <- data[, num_columns]
mean_y <- mean(dependent_var)
if (details){
  console.log(paste("The mean of ",colnames(data)[num_columns]," is", mean_y,"\n\n"))
}
# Iterate through each column (except the last one) as the independent variable
for (i in 1:(num_columns - 1)) {
  independent_var <- data[, i]

  mean_x <- mean(independent_var)
  covar <- cov(independent_var, dependent_var)
  var_x <- var(independent_var)

  if (details) {
    hline()
    console.log("\nStep 1:")
    console.log(paste(colnames(data)[i],":"))
    console.log(paste("     - Mean =", round(mean_x,3)))
    console.log(paste("     - Covariance =", round(covar,3)))
    console.log(paste("     - Variance =",round(var_x,3), "\n\n"))
    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  if (covar != 0) {
    # Calculate the slope and intercept
    b <- covar / var(dependent_var)
    a <- mean_x - b * mean_y

    ssr <- sum((a + b * dependent_var - mean_x)^2)
    ssy <- sum((independent_var - mean_y)^2)
    rcua <- ssr / ssy

    if (details){
      hline()
      console.log("\nSteps 2 and 3")
      console.log(paste(colnames(data)[i],":"))
      console.log(paste("     - Intercept (a) =", round(a,3)))
      console.log(paste("     - Slope (b) =", round(b,3)))
      console.log(paste("     - Sum of squared residuals (ssr) =", round(ssr,3)))
      console.log(paste("     - Sum of squared deviations of y (ssy) =", round(ssy,3)))
      console.log(paste("They are used to calculate: Coefficient of determination (r^2) =",
  round(rcua,3), "\n\n"))
      console.log("")
      if (waiting){
        invisible(readline(prompt = "Press [enter] to continue"))
        console.log("")
      }

      # Plot the points and regression line for each column
      points(dependent_var, independent_var, pch = 16, cex = 1, col = i)
      abline(a, b, col = i, lty = i)

      # Store legend labels, colors, and variable names
```

```
        legend_labels[i] <- paste(" f(x) =", round(a, 3), "+", round(b, 3), "x")
        legend_colors[i] <- i
        variable_names[i] <- colnames(data)[i]

        legend_text <- paste(variable_names, ": ", legend_labels, sep = "")
        legend("topleft", legend = legend_text, col = legend_colors,
               pch = 1, lty = 1, bty = 'n', xjust = 1, cex = 0.8)

        hline()
        console.log("\nStep 4")
        console.log(paste(colnames(data)[i],":"))
        console.log(paste("Data is plotted and the equation is represented in the legend\n"))
        if (i != num_columns - 1){
          if (waiting) {
            invisible(readline(prompt = "Press [enter] to continue"))
            console.log("")
          }
        }
      }
      reg_params[[i]]<- list(var_name = colnames(data)[i], a = a ,b = b)
    } else {
      stop("Covariance = 0 for column ", i, " infinite slope, no line fits the given data.\n")
    }
  }
  return(reg_params)
}
```

Listing 3.1: Linear regression function

At first the current graphical parameters are saved so that at the end of the function, when it is exited, those parameters are restablished.

Upon receiving the dataset, the function calculates the means of x and y, denoted as `meanx` and `meany` respectively. Subsequently, it computes intermediate values, namely `covar` and `varx`, which are pivotal for subsequent calculations. The `apply()` function is employed to iterate through the dataset, calculating the covariance (`covar`) and the squared difference of x from its mean (`varx`). The final covariance and variance calculations involve adjusting `covar` and dividing `varx` by the number of data points.

If the calculated covariance (`covar`) 3.2 is non-zero, indicating a potential linear relationship, the function proceeds with the regression analysis. It calculates the slope (`b`) 3.4 of the regression line by dividing `covar` by `varx` 3.3, rounded to three decimal places. The intercept (`a`) 3.5 of the regression line is then computed using the equation $a = \bar{y} - b \times \bar{x}$, rounded to three decimal places.

To assess the quality of fit of the regression line, the function calculates the sum of squared residuals (`ssr`) 3.6 and the sum of squared deviations of y (`ssy`) 3.7. It then derives the coefficient of determination (`rcua`) by dividing `ssr` by `ssy`, quantifying the proportion of total variability explained by the regression line.

In terms of visualization, if the `details` parameter is set to `TRUE`, the function employs the `plot()` function to generate a scatter plot of the data points, using blue dots to represent the data. Additionally, it adds the regression line to the plot using the `abline()` function, specifying the calculated intercept and slope, and coloring the line red. The visualization is complemented by a legend positioned in the top-left corner of the plot. This legend includes information about the regression equation ($f(x) = a+bx$) and the computed coefficient of determination (R^2) 3.8.

If the covariance is be determined as zero, implying a lack of linear relationship, the function displays a message indicating that no line can fit the given data. In summary, the `linear_regression` function encapsulates the fundamental steps of linear regression analysis, culminating in a visualization of the regression line and pertinent statistics, rendering it a valuable tool for comprehending relationships between variables.

All this process is calculated for as many independent variables as there where provided in the input dataset, assigning different colors for each independent variable.

Throughout the entire function, if the `details` parameter is set to `TRUE` many explanations and clarifications about the process are detailed through the console. Additionally, if the `waiting` parameter is set to `TRUE`, the code will wait for the user to press the `enter` key, in every code explanation block, so that the reading and understanding of the process is clearer.

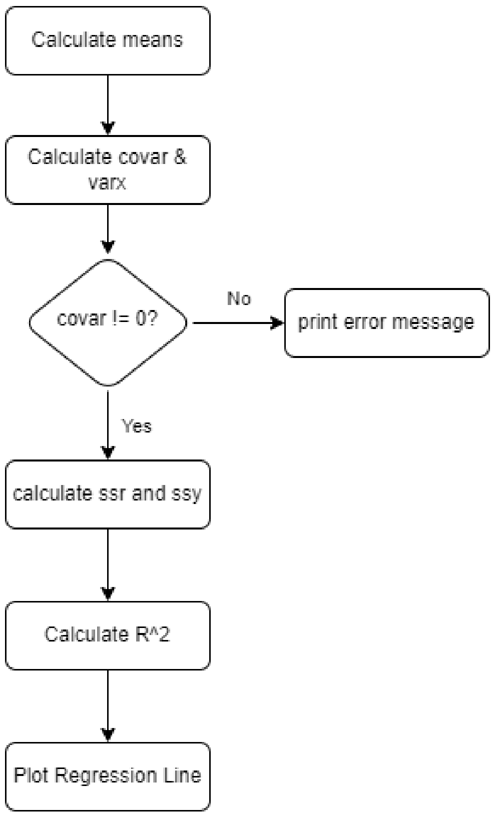


Figure 3.4: Regression Line Flowchart

3.2.1.3 Results

In this section, we present the results of the linear regression analysis performed using the `linear_regression` function. We delve into the details of how the function calculates key parameters such as the slope, intercept, and coefficient of determination for a given input. Additionally, we showcase the visual representation of the regression line plotted alongside the data points, offering an intuitive way to grasp the observed relationship. Finally, we discuss the significance of the results and their implications in understanding the underlying dynamics between the variables.

Without further ado, let's explore the outcomes of the linear regression analysis and gain a deeper understanding of the relationships inherent in the dataset. First lets see the input data. It is simply a small dataframe containing the needed information.

Table 3.1: input data

X	Y
10	8.04
8	6.95
13	7.58
9	8.81
11	8.33
14	9.96
6	7.24
4	4.26
12	10.84
7	4.82
5	5.68

Covariance:

$$\operatorname{cov}(X,Y) = \frac{\sum_{i=1}^{n} x_i \times y_i}{n} - \bar{x} \times \bar{y} \tag{3.2}$$

$$\operatorname{cov}(X,Y) = \frac{10 \times 8.04 + 8 \times 6.95 + \cdots + 5 \times 5.68}{11} - 9 \times 7.5 \approx 5$$

Variance:

$$\operatorname{var}(X) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n} \tag{3.3}$$

$$\operatorname{var}(X) = \frac{(10-9)^2 + (8-9)^2 \cdots + (5-9)^2}{11} = 10$$

Slope:

$$\mathrm{b} = \frac{\operatorname{cov}(X,Y)}{\operatorname{var}(X)} \tag{3.4}$$

$$\mathrm{b} = \frac{5}{10} = 0.5$$

Intercept:

$$\mathrm{a} = \bar{y} - \mathrm{b} \times \bar{x} \tag{3.5}$$

$$\mathrm{a} = 7.5 - 0.5 \times 9 = 3$$

Sum of squared residuals:

$$\mathrm{ssr} = \sum_{i=1}^{n} (\mathrm{a} + \mathrm{b} \times x_i - \bar{y})^2 \tag{3.6}$$

$$\mathrm{ssr} = (3 + 0.5 \times 10 - 7.5)^2 + (3 + 0.5 \times 8 - 7.5)^2 + \cdots + (3 + 0.5 \times 5 - 7.5)^2 = 27.5$$

Sum of squared deviations:

$$\mathrm{ssy} = \sum_{i=1}^{n} (y_1 - \bar{y})^2 \tag{3.7}$$

$$\mathrm{ssy} = (8.04 - \bar{y})^2 + (6.95 - \bar{y})^2 + \cdots + (5.68 - \bar{y})^2 = 41.2727$$

Coefficient of determination:

$$R^2 = \frac{\text{ssr}}{\text{ssy}} \tag{3.8}$$

$$R^2 = \frac{27.5}{41.2727} = 0.666$$

This is the plotted regression line 3.5.

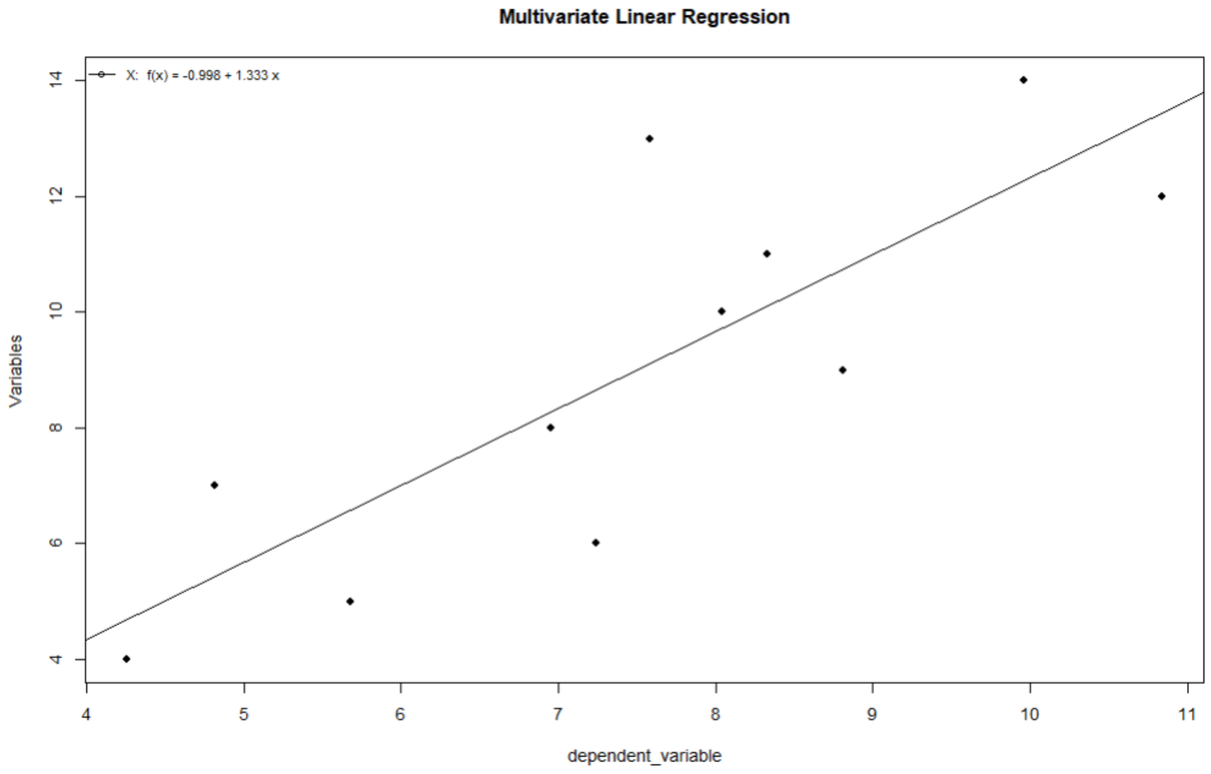


Figure 3.5: Linear Regression Result

This would be the result with a multivariate input 3.6:

3.2.2 Polynomial Regression

Linear regression, as previously explained, is a powerful technique for modeling relationships between variables when those relationships are linear, meaning they can be adequately described by a straight line. However, many real-world scenarios involve complex, non-linear relationships that cannot be effectively captured by a simple linear model. In such cases, we turn to Polynomial Regression, an extension of linear regression that allows us to model and analyze more intricate and non-linear relationships between variables.

3.2.2.1 Theoretical approach

Polynomial regression is a versatile and valuable tool in statistics and machine learning. It enables us to explore the subtleties of data by fitting polynomial functions to the observed relationships, accommodating curves, peaks, and troughs that a linear model cannot represent. By doing so, it broadens the scope of problems we can solve, from predicting housing prices based on square footage to understanding how environmental factors affect crop yields.

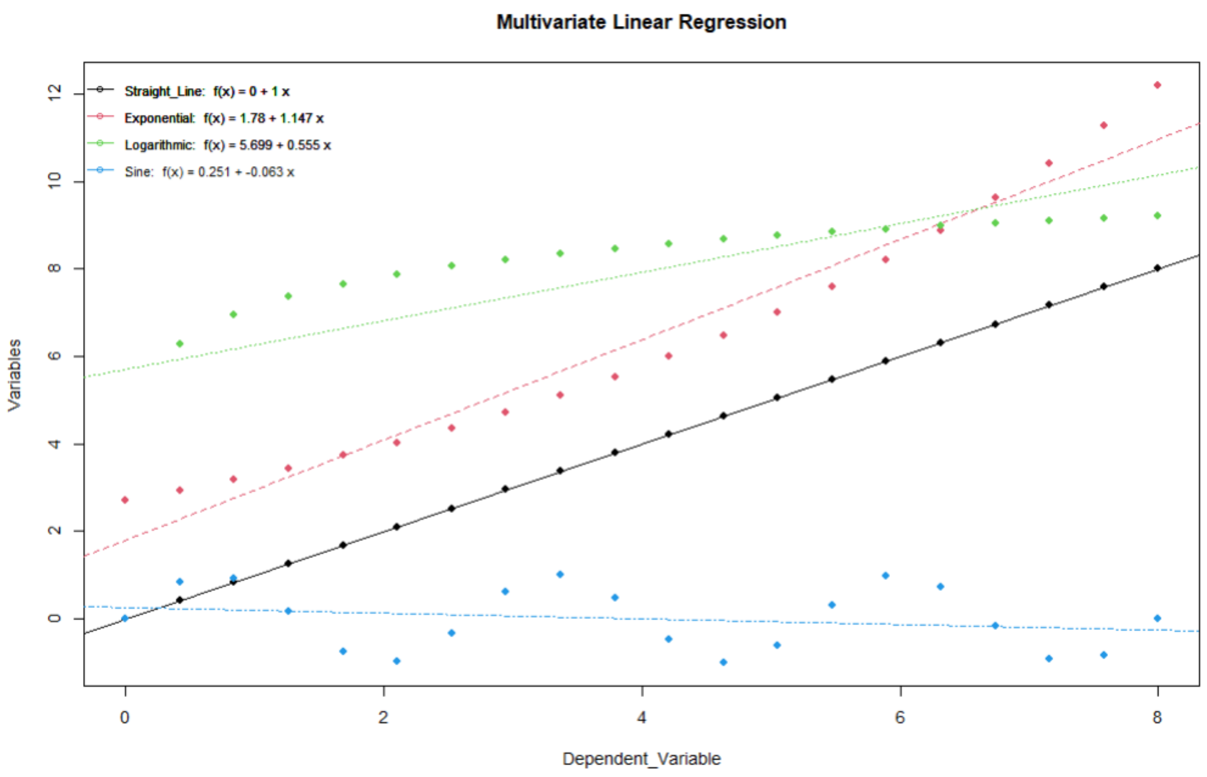


Figure 3.6: Multivariate Linear Regression Result

This approach extends the equation 3.1 to encompass:

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \ldots + \beta_k X^k + \varepsilon \tag{3.9}$$

In this context, $x_1, x_2, \ldots, x_p$ denote the array of independent variables, while $\beta_0, \beta_1, \ldots, \beta_p$ encapsulate the associated coefficients. The degree of the polynomial represents the highest power of the independent variable (X) in the polynomial equation. For example, in a second-degree polynomial, k is 2, and the equation includes X^2 terms.

The cornerstone of polynomial regression is the determination of the coefficients $\beta_0, \beta_1, \ldots, \beta_p$. To do this, the least squares method is a crucial, serving as the foundation for estimating the coefficients that best fit the regression line to the observed data points. It aims to minimize the sum of the squared differences between the actual dependent variable values and the predicted values from the linear regression equation.

The essence of the least squares method lies in minimizing the residuals, which are the vertical distances between each data point and the regression line. The residuals, denoted as e_i, are given by:

$$e_i = y_i - (\beta_0 + \beta_1 x_i) \tag{3.10}$$

The sum of squared residuals (SSR) is defined as:

$$SSR = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))^2 \tag{3.11}$$

The objective is to find the values of β_0 and β_1 that minimize SSR.

Deriving the Coefficients. To determine the optimal coefficients β_0 and β_1, we differentiate SSR with respect to each coefficient and set the derivatives equal to zero. Solving these equations leads to the least squares estimators for β_0 and β_1:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{3.12}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1\bar{x} \tag{3.13}$$

Where $\bar{x}$ and $\bar{y}$ are the means of the independent and dependent variables, respectively.

Interpreting the Coefficients. The estimated coefficient $\hat{\beta}_1$ represents the change in the dependent variable y for a unit change in the independent variable x, assuming all other factors are constant. The intercept $\hat{\beta}_0$ denotes the value of y when x is zero, which might or might not hold practical meaning based on the context.

The least squares method is extremely important in polynomial regression, enabling us to find the coefficients that yield the line of best fit for our data. By minimizing the sum of squared residuals, this method captures the inherent relationship between variables while accounting for measurement errors. This technique empowers us to make informed predictions and gain insights into real-world phenomena.

Determining the correct degree of the equation is also crucial. Polynomial regression can be susceptible to overfitting, this means that if a too high degree is chosen for the polynomial, the model may fit the training data perfectly but generalize poorly to new, unseen data. Proper model selection and regularization techniques can help mitigate this issue.

Polynomial regression addresses some of the limitations of linear regression by offering increased flexibility in modeling nonlinear relationships. Its ability to capture complex data patterns makes it advantageous in scenarios where linearity assumptions do not hold. By using higher-degree polynomial equations, polynomial regression can provide a better fit to data that follows a nonlinear trend, resulting in more accurate predictions. Moreover, it allows for feature engineering, where new polynomial terms are introduced to enhance model performance. However, polynomial regression is not without its drawbacks. Its flexibility can lead to overfitting, where the model captures noise in the data rather than the underlying pattern, thus limiting its generalization ability. The increased complexity of higher-degree polynomials can also make interpretation challenging. Additionally, polynomial regression may require larger datasets to avoid overfitting, especially when employing high-degree polynomials, which can be computationally intensive. In summary, the choice between linear and polynomial regression depends on the nature of the data and the specific modeling requirements of the problem at hand, balancing simplicity and interpretability against the need to capture nonlinear relationships.

3.2.2.2 Coded implementation

The following function is designed to perform polynomial regression analysis on a given dataset. This function allows users to specify the degree of the polynomial they want to fit to the data and provides optional details and interactive waiting prompts for stepwise explanation.

```r
polynomial_regression <- function(data, degree, details = FALSE, waiting = TRUE) {
  oldpar <- par(no.readonly = TRUE)
  on.exit(par(oldpar))
  num_columns <- ncol(data)

  if (details) {
    console.log("\nEXPLANATION (for each independent variable)")
    hline()
    hline()
    console.log("\nStep 1:")
    console.log("    - Create an empty plot with the appropiate limits.")
    console.log("Step 2:")
    console.log("    - Aproximate an equation line that approximates the given values.")
    console.log("      using the lm() function. It employs the least squared error method.")
    console.log("Step 3:")
    console.log("    - Plot the line and the legend.")
    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
    hline()
    hline()
    par(mfrow = c(1, 1))

    plot(1, type = "n", xlim = range(data[, num_columns]),
      ylim = range(data[, 1:(num_columns - 1)]),
      main = "Polynomial Regression",
      xlab = colnames(data)[num_columns],
      ylab = "Variables")
    console.log("\nStep 1:")
    console.log("\nAn empty plot is created with appropiate limits\n\n")
    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
    console.log("The aproximations of the following equations to the provided values")
    console.log("are done adjusting the coefficients of the line to make it the best-fit
    possible.\n\n")
  }

  # Initialize empty vectors for legends
  legend_labels <- character((num_columns - 1))
  legend_colors <- integer((num_columns - 1))

  # Iterate through each column (except the last one) as the independent variable
  coefs_list <- list(num_columns - 1)
  for (i in 1:(num_columns - 1)) {
    independent_var <- data[, i]
    dependent_var <- data[, num_columns]

    # Fit polynomial regression
    poly_fit <- lm(independent_var ~ poly(dependent_var, degree, raw = TRUE))

    # Generate points for the regression line
    y_range <- range(dependent_var)
    y_pred <- seq(y_range[1], y_range[2], length.out = 100)
    x_pred <- predict(poly_fit, newdata = data.frame(dependent_var = y_pred))

    # Extract coefficients of the polynomial
    poly_coefs <- coef(poly_fit)

    # Create legend labels with the full equation
    equation <- paste(
      "f(x) =", round(poly_coefs[1], 3),
      ifelse(poly_coefs[2] >= 0, "+", "-"), abs(round(poly_coefs[2], 3)), "x")

    for (d in 3:(degree + 1)) {
      equation <- paste(equation, ifelse(poly_coefs[d] >= 0, "+", "-"), abs(round(poly_coefs[d
    ], 3)), "x^", (d - 1), sep = "")
    }
    coefs <- list(degree + 1)
    coefs[1] <- colnames(data)[i]
    for (d in 1:(degree + 1)){
      coefs[d+1] <- poly_coefs[d]
    }
    coefs_list[[i]] <- coefs
    legend_labels[i] <- paste(colnames(data)[i], ":", equation)
    legend_colors[i] <- i

    if (details) {
      points(dependent_var, independent_var, pch = 16, cex = 1, col = i)
      lines(y_pred, x_pred, col = i, lty = i)

      # Create the legend
      legend("topleft", legend = legend_labels, col = legend_colors,
             pch = 1, lty = 1, bty = 'n', xjust = 1, cex = 0.8)
      hline()
      console.log("Steps 2 and 3:")
      console.log(paste("Equation ( degree",degree,") aproximation for"
                       ,colnames(data)[i],"--> ",equation,"\n\n"))
      if (i != num_columns - 1){
        if (waiting){
          invisible(readline(prompt = "Press [enter] to continue"))
          console.log("")
        }
      }
    }
  }
  return (coefs_list)
}
```

Listing 3.2: Linear regression function

At first the current graphical parameters are saved so that at the end of the function, when it is exited, those parameters are reestablished. The function iterates through each column in the dataset (except the last one), treating each column as an independent variable. For each independent variable, it performs the following steps:

1. **Polynomial Regression Model:** Using the R `lm()` function, the function fits a polynomial regression model to the data. The ‘degree‘ parameter, specified when calling the function, determines the degree of the polynomial equation. This step involves finding the coefficients that best approximate the relationship between the independent and dependent variables.

2. **Regression Line Generation:** To visualize the fitted model, the function generates a series of data points for the regression line. It does this by defining a range of values for the dependent variable based on its minimum and maximum observed values. Then, it uses the fitted polynomial model to predict the corresponding values of the independent variable. This produces a set of points that form the regression line.

3. **Coefficient Extraction:** The function extracts the coefficients of the polynomial equation, which represent the relationship between the independent and dependent variables. These coefficients include the intercept and various terms for different powers of the dependent variable, up to the specified degree.

4. **Plotting:** If the `details` parameter is set to `TRUE`, the function plots the original data points and overlays the regression line on the same graph. Each independent variable is plotted with a distinct color, and corresponding legend labels are added to the plot for clarity.

If `details` is set to `TRUE`, the function provides a step-by-step explanation of the process. This includes creating an empty plot, approximating the equation using the `lm()` function (least squared error method), plotting the regression line, and prompting the user to continue if `waiting` is set to `TRUE`.

3.2.2.3 Results

After executing the function described earlier with `degree = 4`, which implements polynomial regression analysis, we obtained results for the same input dataset as depicted in the figure 3.6 3.7.

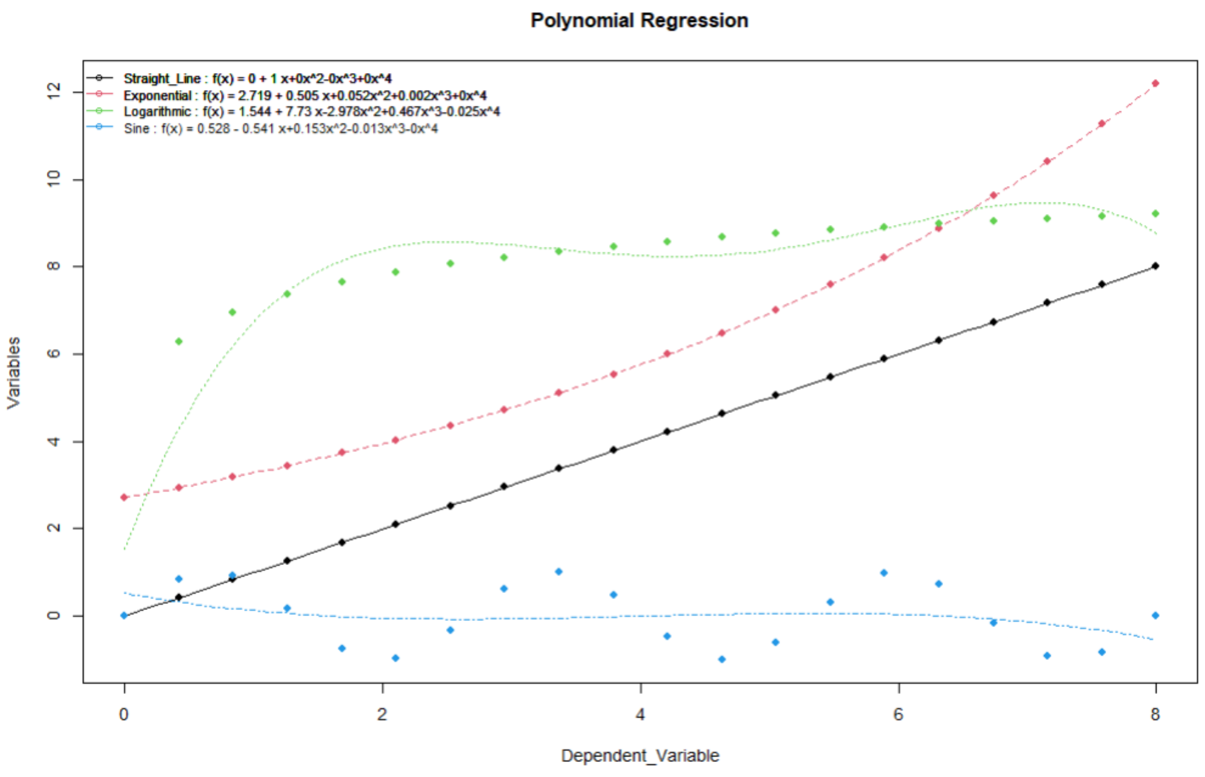


Figure 3.7: Multivariate Polynomial Regression Result (Degree = 4)

For `degree = 8` this would be result 3.8:

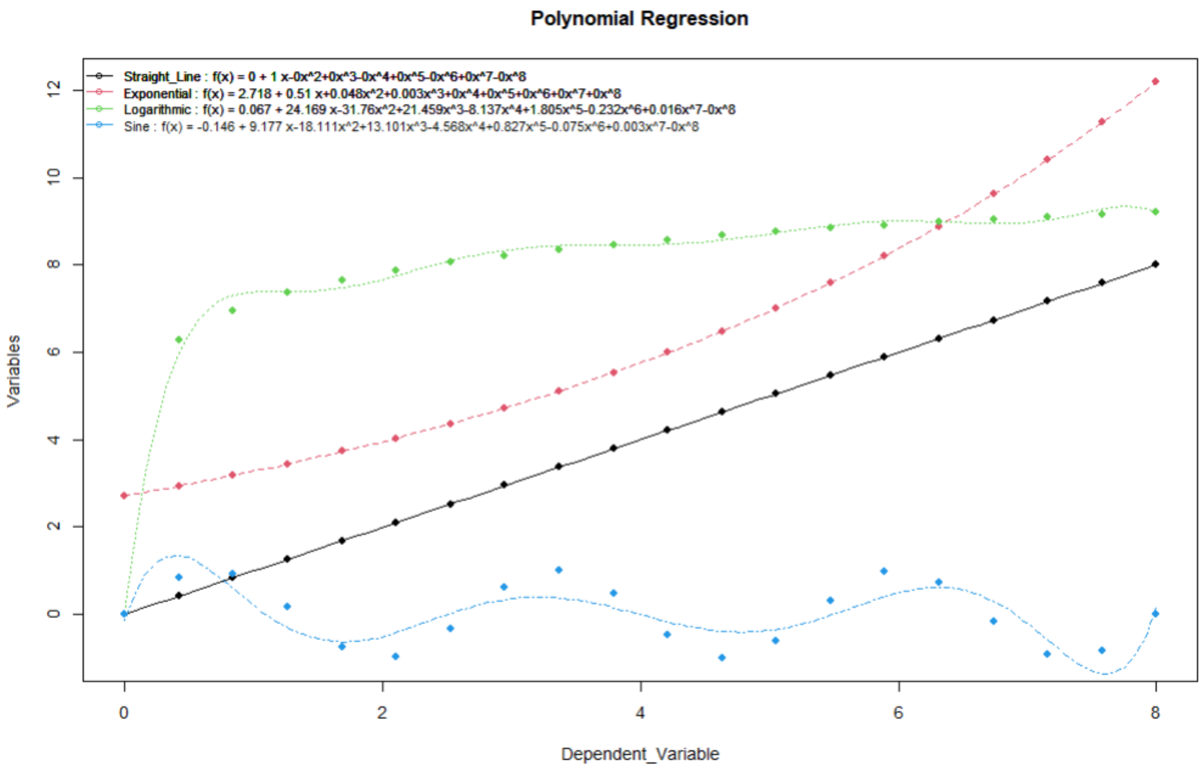


Figure 3.8: Multivariate Polynomial Regression Result (Degree = 8)

3.3 k-NN

3.3.1 Theoretical approach

The k-Nearest Neighbors (k-NN) algorithm is a fundamental and widely used classification and regression technique in machine learning. Thomas M. Cover and Peter E. Hart in 1967 [39] introduced and discussed the concept of nearest neighbor classification, which forms the basis of the K-Nearest Neighbors (K-NN) algorithm. The paper played a significant role in establishing the foundation for distance-based classification methods, including K-NN, in the field of pattern recognition and machine learning. K-NN operates on the principle that data points with similar features tend to belong to the same class or exhibit similar behaviors. k-NN is a non-parametric, instance-based learning algorithm, meaning it doesn't make strong assumptions about the underlying data distribution. Instead, it relies on the training data to classify new instances or predict outcomes.

3.3.1.1 Training Phase

In the training phase of the k-NN algorithm, the model simply stores the labeled training dataset. This dataset consists of tuples, where each tuple includes a feature vector $\mathbf{x}_i$ and a corresponding class label or target value y_i. The algorithm doesn't learn explicit parameters as in parametric models; rather, it memorizes the training data to be used during predictions.

3.3.1.2 Prediction Phase

The prediction phase involves the following steps:

1. **Distance Metric:** The algorithm starts by choosing a distance metric, such as Euclidean distance, to quantify the similarity between data points in the feature space. These are the main distance metrics:

   (a) **Euclidean Distance**: Fundamental geometric metric that calculates the straight-line distance between two points in a Euclidean space 3.9, employing the Pythagorean theorem. By summing the squared differences between corresponding coordinates and taking the square root of the sum, it captures both small and large differences, making it widely used in fields such as image analysis, clustering, and machine learning algorithms like k-nearest neighbors. Despite its effectiveness, the Euclidean distance can be sensitive to outliers and may not generalize well to high-dimensional spaces or non-Euclidean domains, yet it remains a pivotal concept for assessing spatial relationships and proximity between data points.

$$\sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{3.14}$$

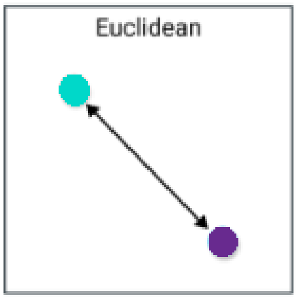


Figure 3.9: euclidean distance

   (b) **Manhattan Distance**: Also known as the L1 distance 3.10 or city block distance, calculates the distance between two points by summing the absolute differences of their coordinates. Inspired by city block navigation, it excels in scenarios where movement is restricted to grid-like paths. This metric's resilience to outliers and its ability to handle data with varying scales make it valuable in computer vision, transportation planning, genetics, and other fields where quantifying directional movement or feature differences is essential.

$$\sum_{i=1}^{n} |p_i - q_i| \tag{3.15}$$

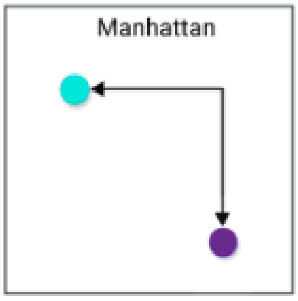


Figure 3.10: Manhattan Distance

(c) **Chebyshev Distance**: Known as the L∞ distance 3.11 or maximum value distance, gauges the dissimilarity between two points by measuring the greatest difference along any dimension. This metric is particularly useful when identifying the dimension in which two points differ the most. It effectively captures the most significant difference while disregarding less impactful variations in other dimensions. Chebyshev distance's applications range from anomaly detection in data analysis to pathfinding algorithms in robotics, where pinpointing the largest deviation is crucial for making informed decisions. Its ability to emphasize the most differing dimension makes it a powerful tool in scenarios where outliers or major discrepancies need special attention.

$$\max_{i=1}^{n} |p_i - q_i| \tag{3.16}$$

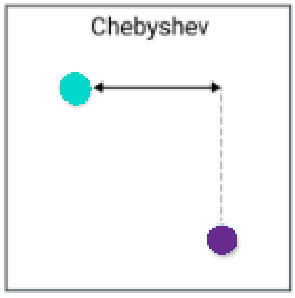


Figure 3.11: Chebyshev Distance

(d) **Minkowski Distance**: Versatile distance metric that unifies both Euclidean and Manhattan distances by introducing a parameter r that controls the distance's behavior 3.12. When r = 2, it reduces to the familiar Euclidean distance, emphasizing balanced distances across all dimensions. On the other hand, when r = 1, it becomes equivalent to the Manhattan distance, focusing on the absolute differences along each dimension. By adjusting r, one can tune the metric's sensitivity to different aspects of the data distribution, making it adaptable to various scenarios. Minkowski distance's flexibility renders it useful in diverse applications such as pattern recognition, feature selection, and optimization, where the choice of r enables tailored distance calculations suited to specific needs.

$$\left(\sum_{i=1}^{n} |p_i - q_i|^r\right)^{\frac{1}{r}} \tag{3.17}$$

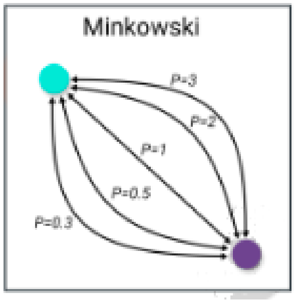


Figure 3.12: Minkowski Distance

(e) **Canberra Distance**: Offers a specialized approach to measuring dissimilarity between two points by considering not just the absolute differences, but also the relative magnitudes of their coordinates. It calculates the distance by summing the absolute differences divided by the sum of the absolute values of the coordinates. This unique normalization accounts for data with varying scales, making it particularly effective in scenarios where different dimensions contribute unequally to the overall distance. Canberra distance is valuable in fields like biology, ecology, and economics, where such relative differences hold significance. It helps identify meaningful relationships in data sets with diverse scales and magnitudes, contributing to more accurate analyses and informed decision-making.

$$\sum_{i=1}^{n} \frac{|p_i - q_i|}{|p_i| + |q_i|} \tag{3.18}$$

(f) **Hamming Distance**: Also known as Binary distance 3.13. It is a binary-specific distance metric that quantifies the dissimilarity between two strings of equal length by counting the positions at which their corresponding symbols differ. Primarily used for comparing sequences of discrete values like binary strings or DNA sequences, Hamming distance provides a straightforward measure of the number of substitutions needed to transform one string into another. It's particularly relevant in error detection and correction, cryptography, and DNA sequence analysis. Due to its simplicity and focus on individual symbol differences, Hamming distance is efficient for tasks where only substitutions are of interest, making it an essential tool in areas where discrete data comparison is paramount.

$$\sum_{i=1}^{n} \delta(s_i, t_i) \tag{3.19}$$

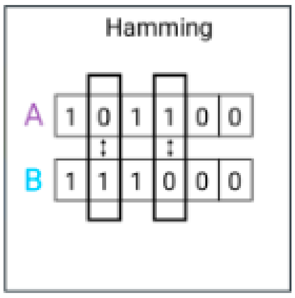


Figure 3.13: Hamming Distance

(g) **Octile Distance**: Distance metric often used in pathfinding algorithms (mainly in robotics), particularly in grid-based environments where diagonal movement is allowed, mostly known as A* 3.14 heuristic. It calculates the distance between two points p1 and p2 by combining the maximum absolute difference and a scaled diagonal step. The formula is given by:

$$\max(|p1 - p2|) + \Big((\sqrt{2} - 1) \times \min(|p1 - p2|)\Big) \tag{3.20}$$

(h) **Jaccard Distance**: The Jaccard Distance 3.15 serves as a valuable measure to evaluate the dissimilarity between sets, especially in contexts where item presence matters. It quantifies the difference between the size of shared elements and the overall union of two sets A and B, with values ranging from 0 (identical sets) to 1 (no common elements). Frequently employed in applications like text analysis and recommendation systems, the Jaccard Distance helps assess document similarity, keyword overlap, or user preferences. Its simplicity and applicability in clustering scenarios further emphasize its relevance across domains requiring efficient set-based

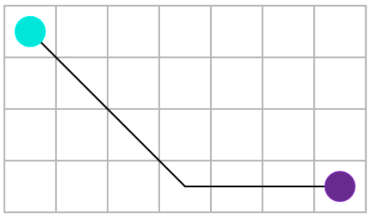


Figure 3.14: Octile Distance

similarity evaluations.

$$1 - \frac{|A \cap B|}{|A \cup B|} \tag{3.21}$$

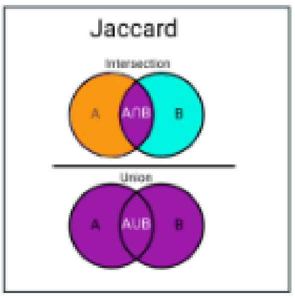


Figure 3.15: Jaccard Distance

(i) **Cosine Distance**: Derived from the cosine similarity concept, offers a powerful means to quantify dissimilarity between vectors in a multi-dimensional space. Evaluates the angle between vectors A and B 3.16, encapsulating their alignment rather than their magnitudes. This makes it particularly effective in scenarios where magnitude differences between vectors might be irrelevant, such as text analysis and recommendation systems. Its values range from 0, indicating parallel vectors, to 2, signifying orthogonal vectors. Frequently harnessed in natural language processing, information retrieval, and collaborative filtering, the Cosine Distance facilitates similarity assessments across diverse domains, enabling the comparison of document vectors, user preferences, or feature sets.

$$\frac{A \cdot B}{\|A\| \cdot \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{3.22}$$

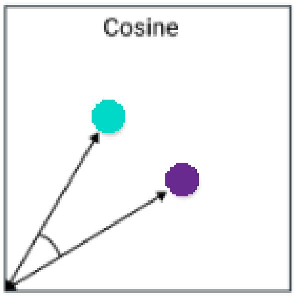


Figure 3.16: Cosine Distance

2. **Calculating Distances:** For a new, unlabeled data point **x**, the algorithm calculates the distances between **x** and all training data points.

3. **Choosing k:** The parameter k specifies the number of nearest neighbors to consider. Selecting an appropriate k is crucial, as it affects both bias and variance. A small k may lead to overfitting, while a large k may cause oversmoothing. For example, for the following example "dataset" we will see the different results as we vary the value of k.

   - **For k = 1:** The new test value will be classified as red as the closest dot is red 3.17.

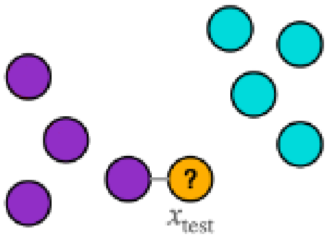


Figure 3.17: K = 1

   - **For k = 3:** The new test value will be classified as blue as 2 of the 3 dots are blue and only 1 is red 3.18.

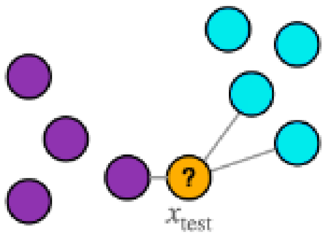


Figure 3.18: K = 3

   - **For k = 4:** the new test value cannot be properly classified as among those 4 closest dots 2 are blue and 2 are red, so another criteria must be used to determine the class the new value belongs to 3.19.

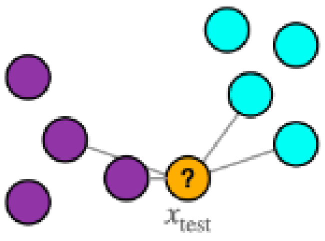


Figure 3.19: K = 4

4. **Finding Neighbors:** The k training data points with the smallest distances to $\mathbf{x}$ are selected as the nearest neighbors.

5. **Majority Vote (Classification) / Average (Regression):**

- For classification tasks, the class label of $\mathbf{x}$ is determined by a majority vote among the class labels of its k nearest neighbors.

$$y = \arg\max_{y_i} \sum_{i=1}^{k} [y_i = y] \tag{3.23}$$

- For regression tasks, the predicted value $\hat{y}$ is the average of the target values of the k nearest neighbors.

$$\hat{y} = \frac{1}{k} \sum_{i=1}^{k} y_i \tag{3.24}$$

3.3.1.3 Mathematical Formulation

Assume we have a training dataset with N data points, where each data point is represented by a feature vector $\mathbf{x}_i$ and a corresponding class label or target value y_i. For a new, unlabeled data point $\mathbf{x}$, the Euclidean distance $d(\mathbf{x}_i, \mathbf{x})$ between $\mathbf{x}$ and each training data point $\mathbf{x}_i$ is given by:

$$d(\mathbf{x}_i, \mathbf{x}) = \sqrt{\sum_{j=1}^{D} (x_{ij} - x_j)^2} \tag{3.25}$$

Here, D is the number of features. After calculating distances, we identify the k nearest neighbors of $\mathbf{x}$ based on the smallest distances.

3.3.1.4 Choosing the Right k

Selecting the appropriate value for the parameter k is a pivotal aspect of employing the k-NN algorithm effectively. The choice of k significantly influences the algorithm's behavior, affecting both the model's ability to capture fine-grained patterns and its susceptibility to noise and outliers.

- **Bias-Variance Trade-off:** The selection of k is rooted in the classical bias-variance trade-off. Smaller values of k tend to lead to low bias but high variance. This means that the predictions can be highly influenced by individual data points, resulting in a model that fits the training data closely but might perform poorly on unseen data due to its sensitivity to noise.

  On the other hand, larger values of k yield high bias and low variance. The model becomes smoother, as predictions are averaged over a larger number of neighbors. This can lead to a more generalizable model, but it might struggle to capture intricate patterns present in the data.

- **Cross-Validation:** To navigate this delicate balance, practitioners often employ techniques like cross-validation. Cross-validation involves partitioning the dataset into training and validation subsets. The model is trained on the training subset and evaluated on the validation subset for different values of k. This process is repeated multiple times to obtain an average performance measure, allowing you to select the k that minimizes the validation error.

- **Grid Search:** Another common approach is grid search, where a predefined range of k values is explored systematically. For each k value, the model's performance is assessed using a chosen metric (such as accuracy or mean squared error), allowing you to identify the k value that optimizes the chosen metric.

- **Impact of Dataset Size and Complexity:** The choice of k should also consider the size of the dataset and the complexity of the underlying relationships. In larger datasets, a smaller k value might be suitable, as the local structure is better captured. Conversely, in smaller datasets, a larger k value can help mitigate the effects of noise.

  Additionally, the complexity of the data's underlying patterns should guide your decision. If the data exhibits intricate boundaries, a smaller k might be preferred to capture fine details. Conversely, when the patterns are relatively smooth, a larger k can help avoid overfitting.

- **Domain Knowledge:** Domain expertise can play a crucial role in selecting k. If you have a deep understanding of the problem domain, you might have insights into the scale of patterns and the potential impact of noise, which can guide your choice of k.

- **Regularization Techniques:** Regularization techniques offer a means to fine-tune the influence of neighbors in the k-NN algorithm. They aim to strike a balance between the simplicity of a smoother decision boundary and the model's capacity to capture intricate patterns. The overarching goal is to mitigate the potential shortcomings associated with extreme values of k while preserving the algorithm's robustness.

  - **Distance-Weighted Voting:** One common regularization technique involves incorporating a distance-weighted approach to the majority vote. Instead of assigning equal importance to all k neighbors, each neighbor's contribution is scaled by its inverse distance to the target point. Intuitively, this approach emphasizes the influence of closer neighbors while reducing the impact of farther ones. The weighted majority vote for classification can be expressed as:

    $$y = \arg\max_{y_i} \sum_{i=1}^{k} \frac{1}{d(\mathbf{x}_i, \mathbf{x})^p} [y_i = y] \tag{3.26}$$

    Here, $d(\mathbf{x}_i, \mathbf{x})$ represents the distance between the new data point $\mathbf{x}$ and its ith nearest neighbor $\mathbf{x}_i$, and p is a tunable parameter that controls the strength of the weighting.

  - **Kernel Density Estimation:** Another approach involves the use of kernel density estimation (KDE) to smooth the influence of neighbors. KDE assigns a continuous density value to each data point, effectively allowing the influence of neighbors to spread smoothly across the feature space.

    By replacing the binary weight in the majority vote with a continuous density function, the classification decision is influenced by the distribution of neighbors rather than just their count. This can lead to more nuanced predictions that account for the local density of neighbors.

    $$f(\mathbf{x}_i) = \frac{1}{N} \sum_{j=1}^{N} K(\mathbf{x}_i - \mathbf{x}_j) \tag{3.27}$$

    The weighted majority vote can be expressed using the density estimates:

    $$y = \arg\max_{y_i} \sum_{i=1}^{k} f(\mathbf{x}_i)[y_i = y] \tag{3.28}$$

    Here, $K(\cdot)$ is typically a Gaussian kernel function, and the probability density is estimated based on the distribution of neighbors within the feature space.

  - **Adaptive k-NN:** Adaptive k-NN techniques aim to adjust the value of k based on the local density of the data. In regions with high data density, a smaller k value is used to capture fine-grained details. Conversely, in sparse regions, a larger k value is employed to counteract the effects of noise.

Such techniques often involve defining a neighborhood radius and setting k to be a function of the number of points within that radius. This adaptive nature allows the algorithm to automatically adjust to the complexity of the data, making it more robust to variations in density.

– **Harnessing Regularization:** Regularization techniques provide a toolbox for tailoring the behavior of the k-NN algorithm to suit specific datasets and contexts. Distance-weighted voting, kernel density estimation, and adaptive k-NN offer nuanced approaches to balance the trade-off between underfitting and overfitting.

  These techniques demonstrate that the k-NN algorithm is not set in stone; rather, it can be molded and refined to achieve optimal performance. The choice of regularization technique hinges on the intricacies of the data, the desired trade-offs, and the goals of the analysis. By embracing regularization, practitioners unlock the potential to enhance the robustness and adaptability of the k-NN algorithm, propelling it from a foundational tool to a tailored solution.

3.3.1.5 Advantages and Limitations

The k-Nearest Neighbors (k-NN) algorithm is a machine learning technique that comes with its own set of advantages and limitations. One of the most prominent advantages of the k-NN algorithm is its simplicity and ease of implementation. Its intuitive concept makes it accessible to those new to machine learning, providing a straightforward approach to solving classification and regression tasks. Additionally, k-NN doesn't require an explicit training phase, as it learns directly from the training data during the prediction stage. This "lazy learning" property enables the algorithm to adapt to changes in data patterns without needing a full retraining cycle, which is particularly beneficial when dealing with dynamic datasets.

Another advantage of k-NN is its flexibility in handling various data types, whether they are numerical or categorical. Unlike some other algorithms that assume linear relationships, k-NN is capable of capturing non-linear relationships, making it applicable to a wide range of problem domains. Moreover, k-NN can be particularly useful when dealing with complex decision boundaries in classification tasks. Its ability to model intricate patterns allows it to excel in scenarios where traditional linear classifiers might struggle.

However, the k-NN algorithm also comes with its share of limitations. One significant limitation is its computational cost during prediction. The algorithm requires calculating distances between the query point and all training data points. As the dataset grows larger, this process becomes computationally expensive, making k-NN less efficient for datasets with a substantial number of instances. Additionally, k-NN is sensitive to noisy data points and outliers. These outliers can significantly influence the algorithm's predictions, potentially leading to inaccurate results. Preprocessing techniques and outlier handling are often required to mitigate this issue.

In classification tasks with imbalanced class distributions, k-NN tends to favor the majority class due to the higher likelihood of finding neighbors from that class. This can result in biased predictions and reduced accuracy for minority classes. Furthermore, the "curse of dimensionality" is another limitation of k-NN. As the number of dimensions increases, the concept of distance between data points becomes less meaningful, diminishing the algorithm's performance. Additionally, selecting the optimal value of the parameter k (number of neighbors) is critical. An inappropriate choice of k can lead to underfitting or overfitting, affecting the model's predictive performance.

3.3.1.6 Conclusion

The k-Nearest Neighbors algorithm is a versatile and powerful technique used in various machine learning applications. Its simplicity, flexibility, and ability to handle complex patterns make it a valuable tool in a practitioner's toolkit. However, careful consideration of the parameter k and preprocessing steps is necessary to maximize its effectiveness.

In closing, the k-NN algorithm stands as an embodiment of the quintessential trade-off in machine learning – the balance between interpretability and predictive power. It exemplifies the idea that sophisticated solutions often emerge from harnessing the inherent patterns within data, steering us towards a deeper understanding of the intricate interplay between similarity and prediction.

The k-NN algorithm's simplicity, adaptability to different data types, and capability to model complex decision boundaries make it a valuable tool in various machine learning applications. However, considerations such as computational cost, sensitivity to noise, optimal parameter selection, and class imbalance need to be addressed to maximize its effectiveness in real-world scenarios.

3.3.2 Coded implementation

The following R function, `knn`, implements the k-Nearest Neighbors algorithm for classification tasks. The k-NN algorithm classifies a new data point by identifying the k training examples (data points with known class labels) that are closest to the new data point and assigning it the class that is most common among these k neighbors.

```r
knn <- function(data,ClassLabel ,p1, d_method = "euclidean", k, p = 3, details = FALSE,
    waiting = TRUE) {
  dist <- apply(
    data[, 1:(length(data) - 1)],
    1,
    distance_method,
    p1 = p1,
    d_method = d_method,
    p = p
  )

  neighbors <- sort(dist, index.return = TRUE)$ix[1:k] # k closest values position
  tags <- data[neighbors, length(data)] # class names in k values
  clas <- table(tags)

  my_string <- "New_Value"
  my_list <- list()
  my_list<- c(my_list, p1)
  my_list <- c(my_list, my_string)
  data <- rbind(data, my_list)

  # Extract the features (columns except the last one, which is the class)
  features <- data[, 1:(length(data) - 1)]
  num_dimensions <- ncol(features)

  prediction <- names(clas)[clas == max(clas)][1] #most repeated class

  if(details){
    console.log("\nEXPLANATION")
    hline()
    hline()
    console.log("\nStep 1:")
    console.log("    - Calculate the chosen d_method from the value we want to classify to
    every other one.")
    console.log("Step 2:")
    console.log("    - Select the k closest neighbors and get their classes.")
    console.log("Step 3:")
    console.log("    - Create a scatterplot matrix with the provided values for visualization
    purpose")
    console.log("Step 4:")
    console.log("    - Select the most repeated class among the k closest neighbors classes.\n
    ")
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
    hline()
    hline()

    console.log("\nStep 1:")
    console.log("\nDistance from p1 to every other p.")
    print(dist)
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }

    hline()
    console.log("\nStep 2:")
    console.log("\nThese are the first k values classes:")
    print(tags)
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }

    # Create a scatterplot matrix with different colors for each class
    colors <- c("red", "blue", "green", "purple", "orange", "cyan", "magenta", "brown", "gray"
    , "pink")
    class_colors <- colors[match(data[[ClassLabel]], unique(data[[ClassLabel]]))]

    pairs(features, col = class_colors)
    legend("topleft", legend = unique(data[[ClassLabel]]), fill = colors, cex = 0.7, xpd =
    TRUE, ncol = 1)

    hline()
    console.log("\nStep 3:")
    console.log("\nPlot values.")
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }

    hline()
    console.log("\nStep 4:")
    console.log(paste("\nThe most represented class among the k closes neighbors is",
    prediction))
    console.log("therefore, that is the new value's predicted class.")
  }
  return (prediction)
}
```

Listing 3.3: K-NN Function

The function takes several parameters as input:

- `data`: The input dataset, represented as a data frame where each row corresponds to a data point, and the last column contains the class labels.
- `ClassLabel`: A string specifying the name of the column containing the class labels.
- `p1`: The data point for which the class label is to be predicted.
- `d_method`: The distance metric used to measure the distance between data points (default: "euclidean").
- `k`: The number of nearest neighbors to consider for classification.
- `p`: The power parameter for the Minkowski distance (used if `d_method` is "minkowski," default: 3).

- `details`: A boolean flag indicating whether to display detailed steps and visualizations.
- `waiting`: A boolean flag indicating whether to pause execution and wait for user input after displaying each step.

The function performs the following steps:

1. **Distance Calculation**: Calculate the distance between the input data point `p1` and all other data points in the dataset using the specified distance metric (`d_method`). Common distance metrics include Euclidean distance and Minkowski distance. The result is stored in the `dist` variable.
2. **Neighbor Selection**: Identify the indices of the `k` nearest neighbors based on the calculated distances. These indices are sorted based on distance values and stored in the `neighbors` variable.
3. **Class Label Extraction**: Retrieve the class labels of the `k` nearest neighbors from the dataset. These labels are stored in the `tags` variable.
4. **Class Counting**: Create a table (`clas`) that counts the occurrences of each class label among the `k` nearest neighbors. This table will be used to determine the most common class among the neighbors.
5. **Adding the New Value**: Create a new data point with the label "New_Value" and add it to the dataset. This allows for the inclusion of the data point being classified within the dataset.
6. **Feature Extraction**: Extract the features from the dataset, which are all columns except the last one (which contains class labels). This set of features is stored in the `features` variable and is used for visualization.
7. **Prediction**: Determine the predicted class label by selecting the class with the highest count among the `k` nearest neighbors. The predicted label is stored in the `prediction` variable.
8. **Details and Visualization**: If the `details` parameter is set to `TRUE`, the function displays detailed explanations and visualizations of the k-NN algorithm's steps. This includes visualizations such as scatterplot matrices and explanations of the algorithm's behavior.
9. **Return Prediction**: The function returns the predicted class label as the final output.

This implementation is intended for educational purposes, providing a clear explanation and visualizations of the k-NN algorithm's steps to aid in understanding its operation and behavior with different datasets and parameters.

In cases of tie votes, the algorithm chooses randomly from the tied classes. This approach ensures that the algorithm does not introduce bias by favoring one class over another when they are equally likely.

```r
distance_method <- function(p1, p2, d_method = "euclidean", p = 3){
  switch (tolower(d_method),
          "euclidean" = euclidean_d(p1, p2),
          "manhattan" = manhattan_d(p1, p2),
          "chebyshev" = chebyshev_d(p1, p2),
          "minkowski" = minkowski_d(p1, p2, p),
          "canberra"  = canberra_d(p1, p2),
          "octile"    = octile_d(p1, p2),
          "hamming"   = hamming_d(p1, p2),
          "binary"    = hamming_d(p1, p2),
          "jaccard"   = jaccard_d(p1, p2),
```

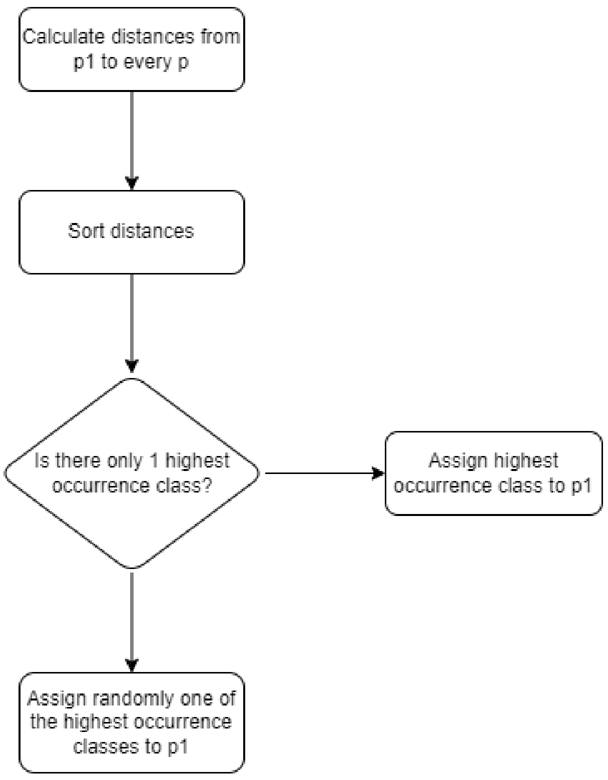

Calculate distances from p1 to every p
Sort distances
Is there only 1 highest occurrence class?
Assign highest occurrence class to p1
Assign randomly one of the highest occurrence classes to p1

Figure 3.20: KNN Flowchart

```
          "cosine"    = cosine_d(p1, p2),
          stop("Unknown distance method")
  )
}

euclidean_d <- function(p1, p2 = p1)  sqrt(sum((p1 - p2)^2))

manhattan_d <- function(p1, p2 = p1)  sum(p1 - p2)

chebyshev_d <- function(p1, p2 = p1)  max(abs(p1 - p2))

minkowski_d <- function(p1, p2 = p1, p = 3)  (sum(abs(p1 - p2)^p)^(1/p))

canberra_d <- function(p1, p2 = p1)  sum(abs(p1 - p2)/(abs(p1) + abs(p2)))

octile_d <- function(p1, p2 = p1)  ((max(abs(p1 - p2))) + ((sqrt(2)-1) * (min(abs(p1-p2)))))

hamming_d <- function(p1, p2 = p1)  sum(p1 != p2)

jaccard_d <- function(p1, p2 = p1)  (1 - (length(intersect(p1, p2)) / length(union(p1, p2))))

cosine_d <- function(p1, p2 = p1)  (sum(p1 * p2) / (sqrt(sum(p1^2)) * sqrt(sum(p2^2))))
```

Listing 3.4: Distance method functions

Notably, the code further enhances its flexibility by providing an array of distance metrics through the `distance_method` function. This comprehensive set includes metrics like Euclidean, Manhattan, Chebyshev, and others, each designed to capture different aspects of data dissimilarity or similarity.

3.3.3 Results

Lets see the insightful outcomes obtained through the application of the `knn` function to a sample input dataset.

For illustrative purposes, let us consider a scenario where we aim to classify a specific data point (`p1`) using the k-NN algorithm. The dataset (`data`) contains a collection of labeled instances, each characterized by features that define its attributes. By leveraging the k-NN approach, we explore how this algorithm can discern patterns within the data and provide a prediction for the classification of the target data point.

We will showcase how the algorithm's decision-making process unfolds, discuss the impact of different parameters, and present the final classification prediction for the chosen input. By elucidating these outcomes, we aim to offer a comprehensive understanding of the k-NN algorithm's effectiveness in capturing data relationships and generating meaningful predictions.

To exemplify the operation of the k-Nearest Neighbors (k-NN) algorithm and showcase its results, let's consider a hypothetical scenario involving a dataset of flowers with distinct attributes. In this illustration, we will use the `knn` function to predict the class of a specific flower based on its features.

Our dataset (`data`) comprises several flower instances, each characterized by features such as petal length, petal width, sepal length, and sepal width. These attributes encapsulate the unique characteristics of each flower and serve as the basis for classification using the k-NN algorithm.

Lets set a flower based dataframe (`data`) to "train" the algorithm containing some features of the flowers in order to identify unseen values:

Table 3.2: Flower Features

Petal Length	**Petal Width**	**Sepal Length**	**Sepal Width**	**Class Label**
1.516451	0.2382057	4.114058	3.597033	setosa
1.667244	0.1508315	4.703633	3.363405	setosa
1.508161	0.2566923	5.431131	3.739184	setosa
1.134878	0.1310670	5.368672	3.068208	setosa
1.600494	0.2159157	5.199039	3.608380	setosa
3.940833	1.270623	6.210460	2.715029	versicolor
4.631613	1.295003	5.200539	3.639288	versicolor
4.862320	1.182477	5.701241	2.104026	versicolor
4.532310	1.442793	6.043681	3.143762	versicolor
4.058404	1.292536	5.791786	3.128121	versicolor
5.831143	1.973074	5.430434	3.036410	virginica
5.648877	1.850175	5.307665	3.082098	virginica
4.814389	1.912310	5.367006	2.527141	virginica
5.258882	2.465356	5.554396	2.749967	virginica
6.089334	1.665120	5.480960	3.712928	virginica
3.284192	0.9323125	5.687376	3.130439	unknown
2.719208	1.0221394	5.924469	2.525883	unknown
2.105287	0.3667641	6.689834	2.674247	unknown
2.630711	0.5491226	6.197555	3.720731	unknown
1.910485	0.8324152	7.176522	2.486113	unknown

Let's focus on a particular flower of interest (`p1`) that possesses the following feature values:

- Petal Length: 4.7
- Petal Width: 1.2
- Sepal Length: 5.3
- Sepal Width: 2.1

With this example input, we will explore how the `knn` function classifies the flower by identifying its nearest neighbors and making a prediction based on their class labels. The distance calculation method (`d_method`), in this case will be the chebyshev distance 3.16; the number of neighbors to consider (`k`), will be set at 5, and (`p`) will be set at its default value, 3.

By dissecting the k-NN algorithm's execution for this specific input, we aim to provide a hands-on understanding of its mechanics, decision-making criteria, and the resulting prediction. This example serves as a practical demonstration of how the k-NN algorithm can leverage data relationships to make informed classifications.

Now, let's dive into the k-Nearest Neighbors results for our flower classification example.

- **calculating distances:** The Chebyshev Distance method is applied being p1 the value to classify and p2 each of the values of the (`data`) dataframe 3.2. These are the results of the calculated distances:

```
[1] 3.183549 3.032756 3.191839 3.565122 3.099506 0.910460 1.539288 0.401241 1.043762
1.028121 1.131143 0.982098 0.712310 1.265356  1.612928 1.415808 1.980792 2.594713
2.069289 2.789515
```

Listing 3.5: Calculated distances

- **Obtaining the k nearest neighbors and their classes:** We obtain the position of the k nearest neighbors from the value and identifying their classes:

```
[1]  8 13  6 12 10
```

Listing 3.6: k Nearest Neighbors indexes

```
[1] "versicolor" "virginica"  "versicolor" "virginica"  "versicolor"
```

Listing 3.7: k Nearest Neighbors classes

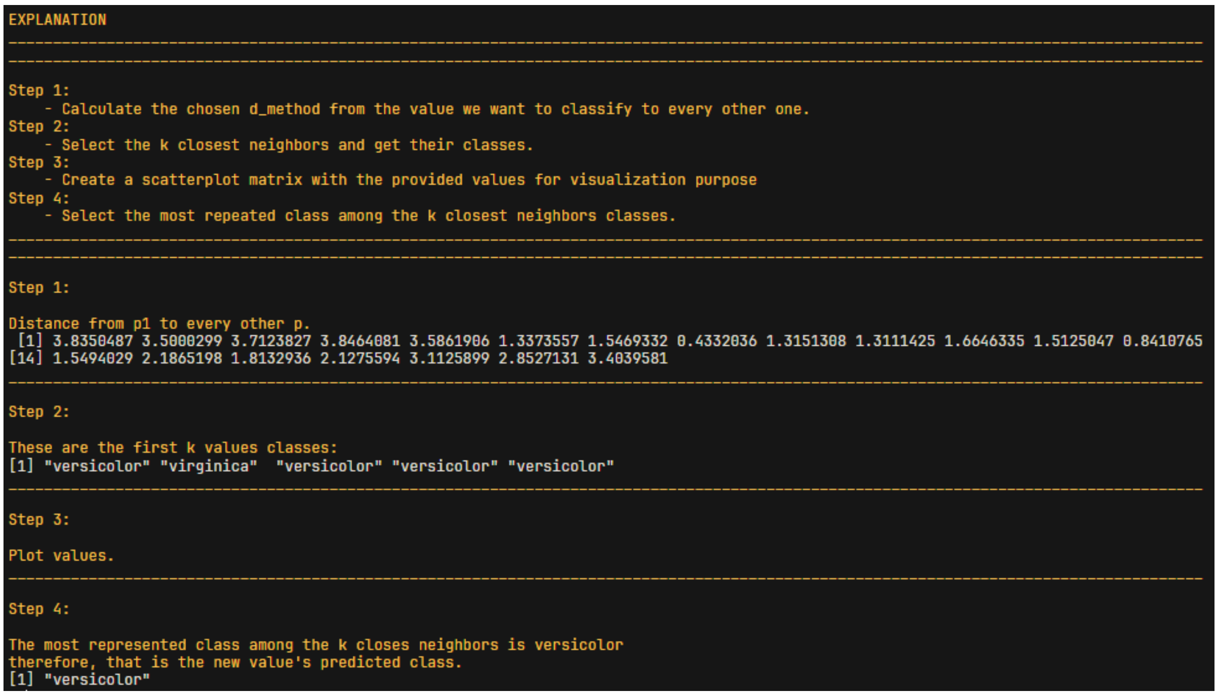
- **Obtaining most repeated class (prediction):** The sum of each class is made and the most represented class is selected as the prediction class:

```
[1] "versicolor"
```

As we can see the prediction is correct because 3 of the 5 nearest neighbors are from the "versicolor" class.

This are the results 3.21 if the `details` parameter is set to `TRUE`.

This is the outputted scatterplot matrix 3.22 if the `details` parameter is set to `TRUE`:



```
EXPLANATION
------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------

Step 1:
    - Calculate the chosen d_method from the value we want to classify to every other one.
Step 2:
    - Select the k closest neighbors and get their classes.
Step 3:
    - Create a scatterplot matrix with the provided values for visualization purpose
Step 4:
    - Select the most repeated class among the k closest neighbors classes.
------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------

Step 1:

Distance from p1 to every other p.
 [1] 3.8350487 3.5000299 3.7123827 3.8464081 3.5861906 1.3373557 1.5469332 0.4332036 1.3151308 1.3111425 1.6646335 1.5125047 0.8410765
[14] 1.5494029 2.1865198 1.8132936 2.1275594 3.1125899 2.8527131 3.4039581
------------------------------------------------------------------------------------------------------------------------------------------

Step 2:

These are the first k values classes:
[1] "versicolor" "virginica"  "versicolor" "versicolor" "versicolor"
------------------------------------------------------------------------------------------------------------------------------------------

Step 3:

Plot values.
------------------------------------------------------------------------------------------------------------------------------------------

Step 4:

The most represented class among the k closes neighbors is versicolor
therefore, that is the new value's predicted class.
[1] "versicolor"
```

Figure 3.21: KNN Result Details

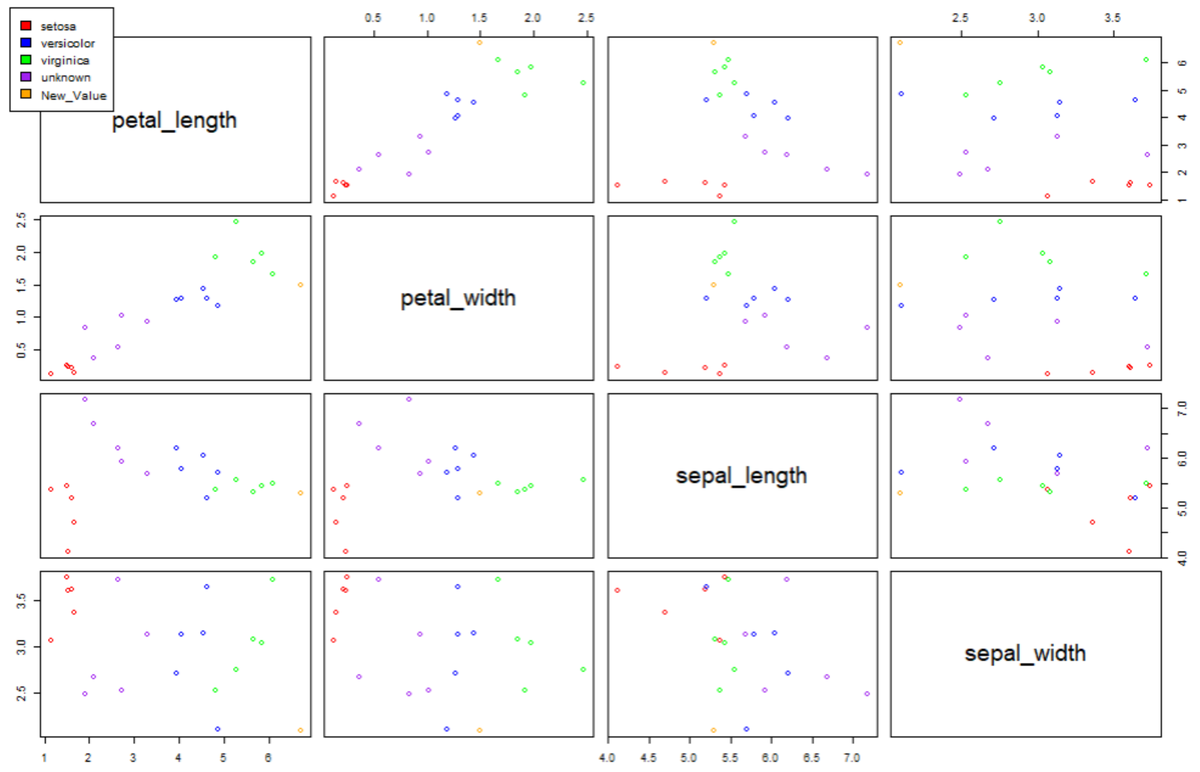


Figure 3.22: KNN Result Scatterplot Matrix

3.4 Perceptron

3.4.1 Theoretical approach

The perceptron was introduced by Frank Rosenblatt in 1957 [40]. It is inspired by the structure and function of biological neurons in the human brain. A neuron receives inputs from multiple sources, processes them, and generates an output signal. Similarly, a perceptron takes numerical inputs, processes them through a set of weights and biases, and produces an output.

A perceptron consists of the following components:

- **Inputs (x):** Inputs are the values or signals that the perceptron receives from the external environment or from other neurons in a more complex neural network. Each input is associated with a weight that determines its importance in influencing the perceptron's output. Inputs could represent features of a dataset or signals from other neurons in a more complex network.
- **Weights (w):** Weights are parameters associated with each input. They indicate the strength of the connection between the input and the perceptron's output. A larger weight means the associated input has a stronger influence on the output. Conversely, a smaller weight reduces the input's impact. The process of learning in a perceptron involves adjusting these weights to minimize the error between the actual output and the desired output (target). Learning algorithms, like the **perceptron learning rule** or **gradient descent**, are used to update the weights during training.
- **Bias (b):** The bias is an additional parameter in a perceptron that represents the threshold towards which the sum of weighted inputs must exceed for the perceptron to activate (fire) and produce an output. In essence, it allows the perceptron to control when it should activate even if the weighted sum of inputs is not extremely high. The bias shifts the decision boundary of the perceptron.
- **Activation Function (f):** The activation function defines the output of the perceptron based on the weighted sum of inputs and the bias. It introduces non-linearity to the perceptron's behavior, enabling it to model complex relationships in data. Without an activation function, the perceptron would only be able to perform linear transformations.

The operation of a perceptron can be described in the following steps:

1. Multiply each input (x) by its corresponding weight (w) to obtain weighted inputs ($w \cdot x$).
2. Sum up the weighted inputs and add the bias (b) to get the net input (z): $z = (w_1 \cdot x_1) + (w_2 \cdot x_2) + \ldots + (w_n \cdot x_n) + b$.
3. Pass the net input through the activation function (f) to produce the output (y): $y = f(z)$.

The process of training a perceptron involves adjusting its weights and bias to optimize its performance on a given task. This is typically done using a supervised learning approach, where the perceptron is provided with input-output pairs (training data) and learns to adjust its parameters to minimize the error between its predicted output and the true output.

The learning algorithm often used for training perceptrons is the **perceptron learning rule**, which updates the weights and bias based on the error between the predicted output and the desired output. This iterative process continues until the perceptron reaches a state where the error is sufficiently minimized.

While perceptrons are the foundation of neural networks, they have limitations. One notable limitation is their inability to handle nonlinearly separable data. This led to the development of more complex

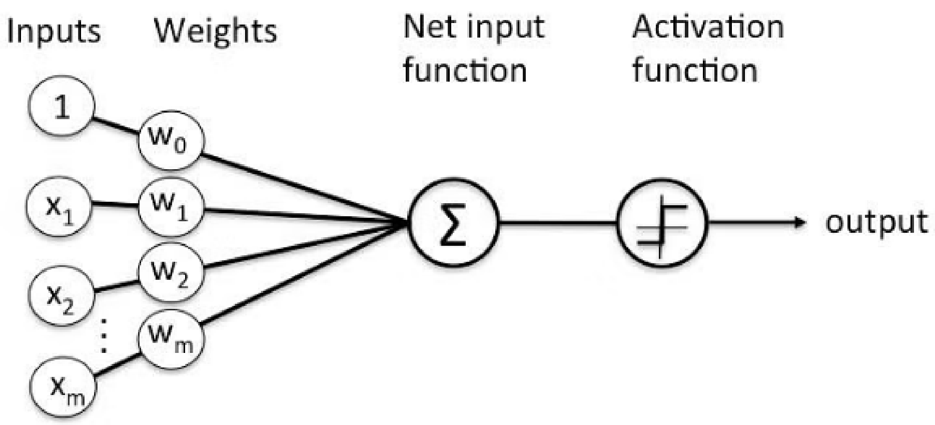


Figure 3.23: Perceptron Structure

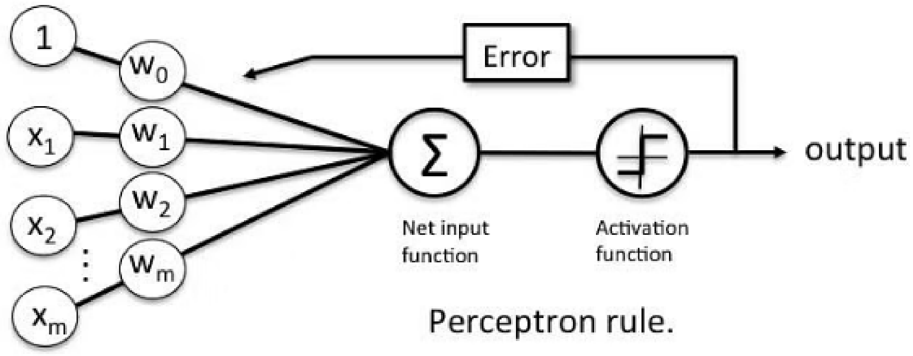


Figure 3.24: Perceptron Learning Rule

architectures like multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs), which can capture intricate relationships in data.

The perceptron, with its inspiration from biology and simple mathematical operations, forms the bedrock of neural network architecture. While the perceptron itself has limitations, its evolution has paved the way for more advanced and powerful neural network models that have revolutionized artificial intelligence and machine learning.

3.4.2 Coded implementation

The following code offers an implementation of the perceptron learning algorithm, A foundational technique in machine learning employed for binary classification tasks. Alongside the core algorithm, auxiliary functions for training and specifying activation methods for the perceptron are defined. The algorithm operates as follows:

At the heart of the code is the `perceptron` function, serving as the primary interface to employ the perceptron algorithm. The function expects several essential inputs:

- `training_data`: A dataframe used for training the perceptron. Each row of the dataset represents an instance with associated features, and the final column signifies the class label.
- `to_classify`: A vector containing the feature values of a new instance awaiting classification.

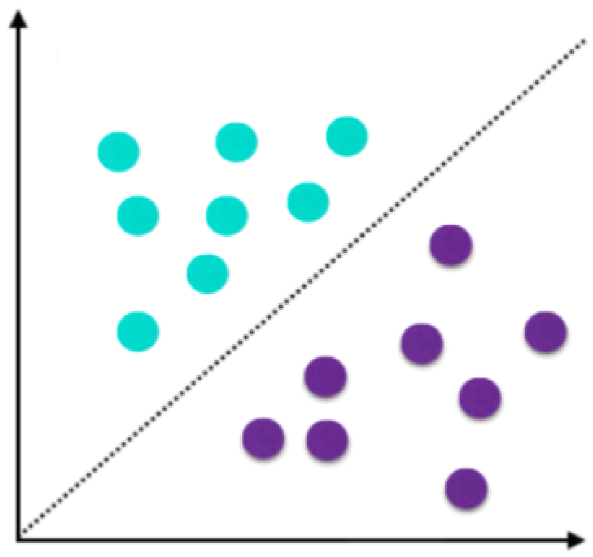


Figure 3.25: Linear Separability

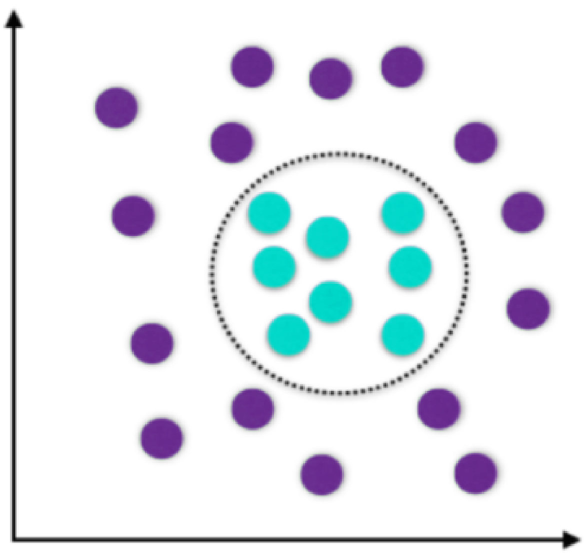


Figure 3.26: Non-Linear Separability

- `activation_method`: A parameter enabling the selection of an activation method for the perceptron's operation, they can be: "step", "sine","tangent", "relu", "gelu", "swish" or "linear."
- `max_iter`: The maximum number of iterations that the training process will undertake. This value should be varied depending on the learning rate. values between 300 and 3000 are recommended.
- `learning_rate`: A parameter determining the rate at which the weights are adjusted during training. Depending on the dataset this value should be changed. Anyway, values between 0.005 and 0.3 are recommended because bigger values will make the training really unstable and smaller values would make the training take way too many iterations

```
perceptron <- function(training_data, to_clasify, activation_method, max_iter, learning_rate,
    details = FALSE, waiting = TRUE){
  if(details){
    console.log("\nEXPLANATION")
    hline()
    hline()
    console.log("\nStep 1:")
    console.log("    - Generate a random weight for each variable.")
    console.log("Step 2:")
    console.log("    - Check if the weight classify correctly. If they do, go to step 4")
    console.log("Step 3:")
    console.log("    - Adjust weights based on the error between the expected output and the
    real output.")
    console.log("    - If max_iter is reached go to step 4. If not, go to step 2.")
    console.log("Step 4:")
    console.log("    - Return the weigths and use them to classigy the new value\n")
    hline()
    hline()
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }
  weigths <- per_training(training_data, activation_method, max_iter, learning_rate, details,
    waiting)
  clasificacion <- as.numeric(act_method(activation_method,sum(weigths * to_clasify)) > 0.5)
  if (details){
    hline()
    console.log("\nStep 4:\n")
    console.log(paste("Predicted value:", clasificacion, "\n"))
    console.log("Final weigths:")
    print(weigths)
  }
  return(weigths)
}
```

Listing 3.8: Perceptron Function

Inside the `perceptron` function:

The `per_training` function is invoked to train the perceptron using the `training_data` and specified parameters. The calculated weights are stored in the variable `weights` (note the typo; it should be `weights`) and subsequently printed. Then the function calculates the classification for the new instance (`to_classify`) using the trained perceptron. Finally the classification outcome is displayed.

```
per_training <- function(training_data, activation_method, max_iter, learning_rate, details,
    waiting){
  env <- new.env()
  env$weigths <- runif(ncol(training_data)-1, min = -1, max = 1)
  if (details){
    console.log("\nStep 1:")
    console.log(paste("Random weights between -1 and 1 are generated for each variable:"))
    print(env$weigths)
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
    hline()
    console.log("\nSteps 2 and 3:")
  }
  env$is_correct <- FALSE
  sapply(
    1:max_iter,
    function(a){
      if (!env$is_correct){# If every element is classified, we are done
        env$is_correct <- TRUE
        # Verify if every value is correctly classified
        apply(
          training_data,
          1,
          function(b){
            if (env$is_correct){
              inputs <- b[1:length(b)-1]
              expected_output <- b[length(b)]
              output <- act_method(activation_method,sum(env$weigths * inputs))
              if (as.numeric(output > 0.5) != expected_output) {env$is_correct <- FALSE}
            }
          }
        )
        if (!env$is_correct){
          # select a random value from training_data
          row_num <- sample(1:nrow(training_data), 1)
          inputs <- training_data[row_num, 1:ncol(training_data)-1]
          expected_output <- training_data[row_num, ncol(training_data)]

          # calculate output and update weights
          output <- act_method(activation_method,sum(env$weigths * inputs))
          error <- expected_output - output
          env$weigths <- env$weigths + learning_rate * error * inputs
          if(details){
            console.log("Weights do not classify correctly so they get adjusted:")
            print(env$weigths)
            if(waiting){
              invisible(readline(prompt = "Press [enter] to continue"))
              console.log("")
            }
          }
        }
      }
    }
  )
  console.log("")
  return(env$weigths)
}
```

Listing 3.9: Perceptron Auxiliar Function

Inside the per_training function, the iterative training process of the perceptron unfolds as follows. Initially, the function generates a set of random weights using the runif function (values between -1 and 1), with the count of weights aligning with the number of features in the training data, excluding the class label. Subsequently, the training process commences, continuing for a maximum of max_iter iterations or until all training examples are accurately classified. During each iteration, a check is performed to determine whether all examples have been correctly classified. If so, the training halts. If not, the function traverses each example in the training data. For each example, the inputs and expected output are updated based on the current instance. The perceptron's output is then computed, employing the chosen activation method and the current weights. If the perceptron's output does not match the expected output, a flag is_correct is set to FALSE. A random example (row) is sampled from the training data, and its inputs and expected output are stored for reference. Subsequently, the perceptron's output is recalculated, and the error (err) is determined. The weights are then adjusted using the error correction learning rule. Once all iterations are completed, the trained weights are returned. Check the 3.27 for a simple flowchart of the algorithm implementation.

If the details parameter is set to TRUE, the updated weights are displayed, and optionally, the function waits for user input to proceed.

Overall, the per_training function trains the perceptron by iteratively adjusting the weights based on the classification errors until all training examples are correctly classified or until reaching the maximum number of iterations.

The perceptron function then uses these learned weights to classify new input data and returns the result. If details is set to TRUE, it also displays the predicted value and the final weights.

The loop has to be always fully done in order to use the apply() function, which is significantly faster than a for loop. Tests have been performed and this way, although seeming slower, performs better in most situations. It is also worth mentioning that the algorithms training stops when the perceptron correctly classifies every value or when the max_iter value is reached. This has been done for simplification but other stopping metrics could be implemented, such as a percentage of correctly classified values.

```
act_method <- function(method, x){
```

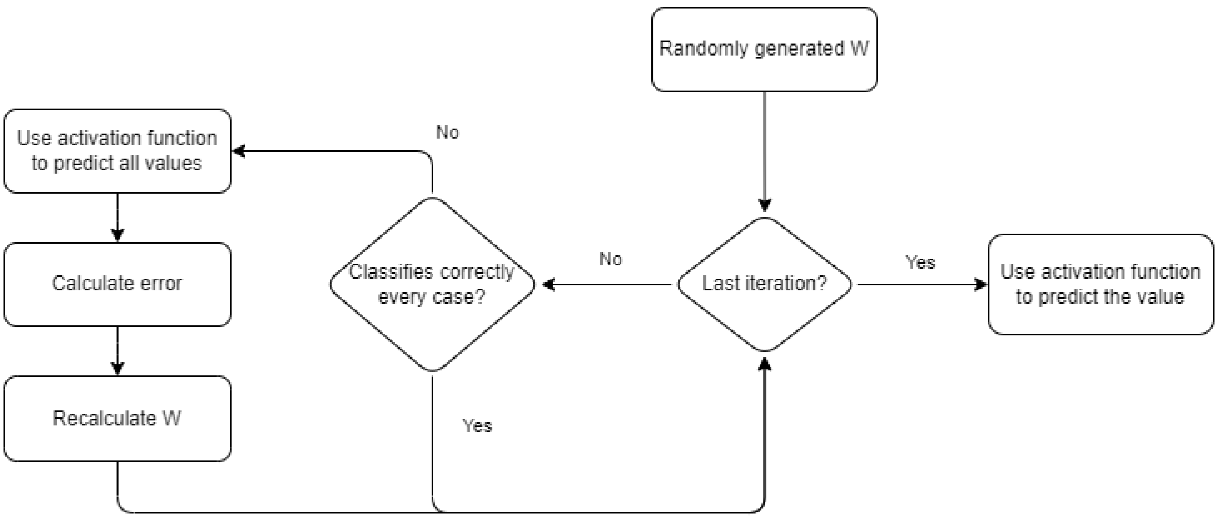

Randomly generated W
Use activation function to predict all values
No
Calculate error
Classifies correctly every case?
No
Last iteration?
Yes
Use activation function to predict the value
Recalculate W
Yes

Figure 3.27: Perceptron Flowchart

```
    switch (tolower(method),
        "step"      = as.numeric(x > 0.5),
        "sine"      = (exp(x) - exp(-x)) / 2,
        "tangent"   = (exp(x) - exp(-x)) / (exp(x) + exp(-x)),
        "linear"    = x,
        "relu"      = pmax(x, 0),
        "gelu"      = 0.5 * x * (1 + tanh(sqrt(2 / pi) * (x + 0.044715 * x^3))),
        "swish"     = x / (1 + exp(-x)),
        stop("Unknown method")
    )
}
```

Listing 3.10: Activation Method Function

The `act_method` function provides various activation methods that the perceptron can employ. This function accepts the `method` parameter and an input value `x`. Depending on the selected `method`, it returns the outcome of distinct activation functions: step function, sine function, cosine function, tangent function, or linear function. If the specified `method` is not recognized, the function raises an error.

In essence, the provided R code encapsulates the mechanics of the perceptron learning algorithm, allowing the iterative adjustment of weights through error correction to classify new instances. Users have the flexibility to choose from various activation methods for the perceptron's operations.

3.4.3 Results

In the upcoming section, we will delve into the results of our implementation of the perceptron algorithm. It's important to note that the perceptron algorithm involves an element of randomness in its learning process. As a result, we won't be able to provide a highly detailed account of each individual run. Instead, we will present a collection of different results obtained from multiple runs of the algorithm, along with some illustrative input examples.

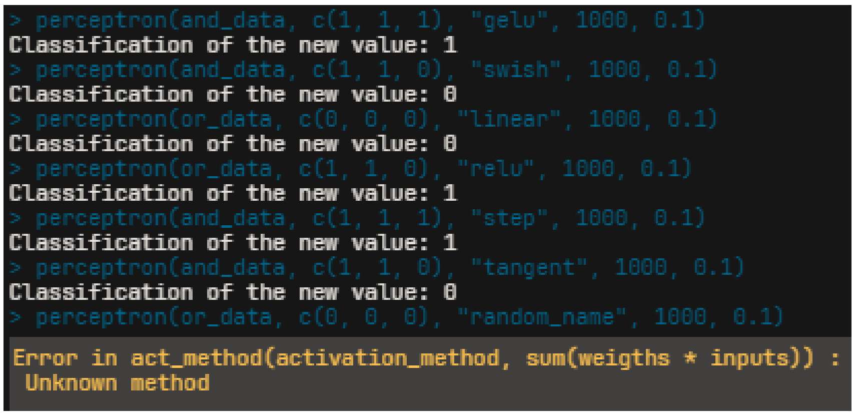
By presenting a variety of results and highlighting key input examples, we aim to provide a comprehensive understanding of the perceptron algorithm's capabilities and limitations. Lets check this 2 small datasets:

Table 3.3: And_data.

x1	x2	x3	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 3.4: Or_data.

x1	x2	x3	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



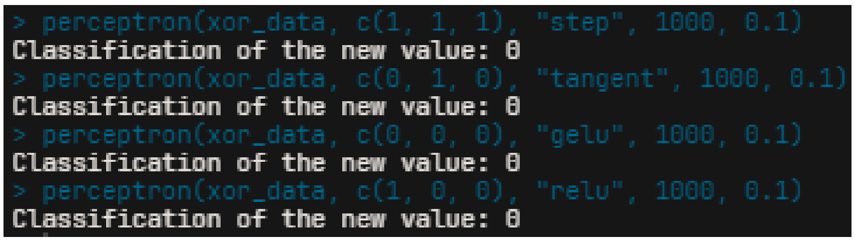
```
> perceptron(and_data, c(1, 1, 1), "gelu", 1000, 0.1)
Classification of the new value: 1
> perceptron(and_data, c(1, 1, 0), "swish", 1000, 0.1)
Classification of the new value: 0
> perceptron(or_data, c(0, 0, 0), "linear", 1000, 0.1)
Classification of the new value: 0
> perceptron(or_data, c(1, 1, 0), "relu", 1000, 0.1)
Classification of the new value: 1
> perceptron(and_data, c(1, 1, 1), "step", 1000, 0.1)
Classification of the new value: 1
> perceptron(and_data, c(1, 1, 0), "tangent", 1000, 0.1)
Classification of the new value: 0
> perceptron(or_data, c(0, 0, 0), "random_name", 1000, 0.1)
Error in act_method(activation_method, sum(weigths * inputs)) :
  Unknown method
```

Figure 3.28: Perceptron function predictions

As we can see the perceptron classifies correctly every single value we try using linearly separable input datasets. But if we try with a non linearly separable dataset such as 3.5, the predicted values are correctly rated at a random rate 3.29.

Table 3.5: Xor_data.

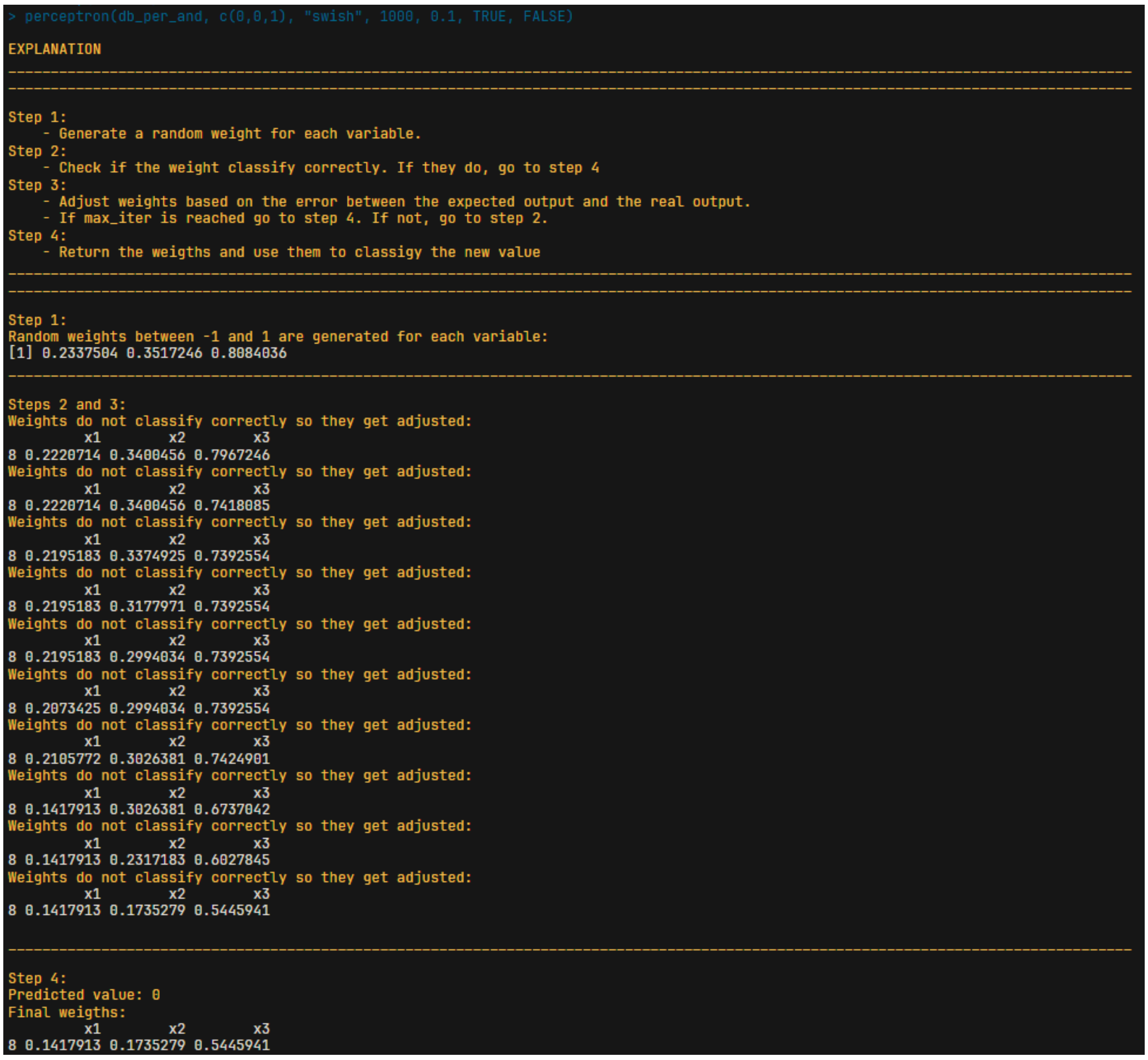
x1	x2	x3	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0



```
> perceptron(xor_data, c(1, 1, 1), "step", 1000, 0.1)
Classification of the new value: 0
> perceptron(xor_data, c(0, 1, 0), "tangent", 1000, 0.1)
Classification of the new value: 0
> perceptron(xor_data, c(0, 0, 0), "gelu", 1000, 0.1)
Classification of the new value: 0
> perceptron(xor_data, c(1, 0, 0), "relu", 1000, 0.1)
Classification of the new value: 0
```

Figure 3.29: Perceptron function, Xor dataset predictions

In case the user invoked this function setting `details = TRUE`, this would be the output 3.30



```
> perceptron(db_per_and, c(0,0,1), "swish", 1000, 0.1, TRUE, FALSE)

EXPLANATION
------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------

Step 1:
    - Generate a random weight for each variable.
Step 2:
    - Check if the weight classify correctly. If they do, go to step 4
Step 3:
    - Adjust weights based on the error between the expected output and the real output.
    - If max_iter is reached go to step 4. If not, go to step 2.
Step 4:
    - Return the weigths and use them to classigy the new value
------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------

Step 1:
Random weights between -1 and 1 are generated for each variable:
[1] 0.2337504 0.3517246 0.8084036
------------------------------------------------------------------------------------------------------------------------------------

Steps 2 and 3:
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.2220714 0.3400456 0.7967246
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.2220714 0.3400456 0.7418085
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.2195183 0.3374925 0.7392554
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.2195183 0.3177971 0.7392554
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.2195183 0.2994034 0.7392554
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.2073425 0.2994034 0.7392554
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.2105772 0.3026381 0.7424901
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.1417913 0.3026381 0.6737042
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.1417913 0.2317183 0.6027845
Weights do not classify correctly so they get adjusted:
         x1        x2        x3
8 0.1417913 0.1735279 0.5445941


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

Step 4:
Predicted value: 0
Final weigths:
         x1        x2        x3
8 0.1417913 0.1735279 0.5445941
```

Figure 3.30: Perceptron output explanations

3.5 Decision Tree

3.5.1 Theoretical approach

Decision trees are a popular machine learning algorithm used for both classification and regression tasks ([16], [41] and [42] are recommended reading). They represent a decision-making model in the form of a tree structure, where each internal node represents a feature or attribute, each branch represents a decision rule or outcome, and each leaf node represents a class label or a numerical value. The goal of a decision tree is to recursively split the data into subsets based on the most informative features, effectively creating a hierarchical set of rules for decision-making. These rules are learned from the training data and can be easily interpreted by humans, making decision trees valuable for both predictive modeling and understanding the underlying patterns in data.

3.5.1.1 Key Components of a Decision Tree

- **Root Node:** The top node of the decision tree, representing the entire dataset at the beginning of the decision-making process. It is the starting point for the tree's traversal and contains the initial set of data that needs to be divided into subsets.

- **Internal Nodes:** Nodes between the root node and the leaf nodes. Each internal node represents a decision point based on a specific feature and its associated condition. These nodes guide the traversal of the tree by branching the data into different subsets according to the conditions.

- **Leaf Nodes:** The final nodes of the tree. These nodes do not further split the data but contain the predicted output or class label for the given input. The predictions in leaf nodes are made based on the majority class in the case of classification tasks or the mean (or other statistic) of the target values in regression tasks.

- **Edges/Branches:** Connections between nodes that represent the flow of decisions. Edges connect the parent node to its child nodes, indicating the path the data takes through the tree based on the feature conditions.

- **Splitting Criteria:** Rules used to decide how to split the data at each internal node. In classification tasks, common splitting criteria include Gini impurity and entropy. In regression tasks, mean squared error (MSE) is commonly used. The splitting criterion measures the impurity or error in the data and guides the tree to select the best feature and threshold for splitting.

- **Predicted Output:** The outcome predicted by the decision tree for a given input. For classification, the predicted output is the class label associated with the majority of samples in the leaf node. For regression, the predicted output is the statistical measure (e.g., mean) of the target values in the leaf node.

- **Depth of the tree:** The depth of the tree refers to the length of the longest path from the root node to a leaf node. A deeper tree can capture more complex relationships in the data but might also be prone to overfitting. We will see this in more detail.

- **Stopping criteria:** Conditions that determine when to stop the tree-building process. Common stopping criteria include reaching a maximum depth, having a minimum number of samples in a node, or reaching a threshold impurity level.Stopping criteria help prevent overfitting and lead to a more generalized model.

- **Pruning:** Pruning involves removing parts of the tree to reduce its complexity and improve generalization. Pruning can help avoid overfitting by removing nodes that do not significantly contribute to the model's predictive power.

Understanding these key components is essential for grasping the mechanics of decision trees and their role in making predictions based on feature conditions and impurity measures.

As seen in the diagram figure above, a decision tree starts with a root node, which has no incoming branches. The outgoing branches from the root node feed into internal nodes, also known as decision nodes. Based on the available features, both types of nodes perform evaluations to create homogeneous subsets, indicated by leaf nodes or terminal nodes. Leaf nodes represent all possible outcomes within the dataset.

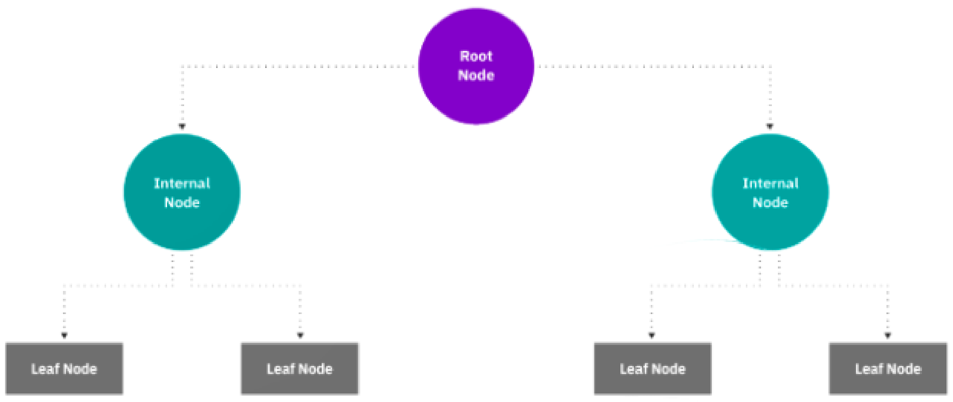


Figure 3.31: Decision Tree Structure

The process of building a decision tree involves recursively partitioning the dataset into subsets based on feature conditions. The goal is to create nodes that maximize information gain or minimize impurity at each step.

This type of flowchart structure also creates an easily understandable representation of decision-making, allowing different groups within an organization to better grasp why a decision was made.

Decision tree learning employs a divide-and-conquer strategy by using a greedy search to identify optimal split points within a tree. This splitting process is recursively repeated from top to bottom until all or most records are classified under specific class labels. Whether all data points are classified as homogeneous sets or not depends largely on the complexity of the decision tree. Smaller trees are easier to create pure leaf nodes, meaning data points in a single class. However, as a tree grows in size, it becomes increasingly challenging to maintain this purity and usually results in very few data points within a specific subtree. When this happens, it's known as data fragmentation and often can lead to overfitting.

As a result, decision trees prefer smaller trees, which aligns with the principle of parsimony in Occam's Razor. In other words, "entities should not be multiplied beyond necessity." In simpler terms, decision trees should add complexity only when necessary, as the simplest explanation is usually the best.

There are different algorithms for constructing decision trees:

- **ID3 algorithm:** Ross Quinlan's ID3 algorithm (1986) [43] is often credited as one of the first formal introductions of decision trees in machine learning. The ID3 (Iterative Dichotomiser 3) algorithm stands as a seminal method for constructing decision trees in the realm of machine learning. The algorithm employs a top-down, recursive approach to partition a given dataset into subsets based on attributes that demonstrate the most significant information gain. This algorithm is particularly well-suited for problems involving categorical attributes and discrete class labels, allowing for the creation of human-readable decision trees that capture underlying patterns in the data.

  The ID3 algorithm encompasses several key steps that culminate in the formation of a decision tree:

  1. **Input and Termination Criteria:** The algorithm begins with a dataset containing instances, each characterized by a set of attributes and corresponding class labels. The process halts when one all instances belong to the same class, leading to the creation of a pure leaf node or when no attributes remain available for further splitting.
  2. **Attribute Selection:** Central to the ID3 algorithm is the strategic choice of attributes that will serve as nodes for splitting the dataset. The attribute selected is the one that maximizes

the Information Gain (IG) or Gain Ratio (GR). Information Gain quantifies the reduction in uncertainty achieved through a particular split, while Gain Ratio normalizes this gain by taking into account the intrinsic information associated with the attribute.

3. **Creating Child Nodes:** Having identified the attribute with the highest Information Gain or Gain Ratio, the dataset is partitioned into subsets based on the attribute's distinct values. For each subset, a new internal node is introduced in the decision tree, thus initiating a recursive process. This recursive approach continues until the termination criteria are met for the subsets.
4. **Pruning and Optimization:** ID3, while proficient, tends to construct deep decision trees that are prone to overfitting. Post-processing pruning techniques are frequently applied to mitigate overfitting and foster generalization. Pruning involves removing branches or nodes from the tree that may lead to undesirable complexity.

The ID3 algorithm possesses several notable advantages that have contributed to its popularity and use in various applications. One of its primary strengths lies in its simplicity and intuitive nature. The algorithm's straightforward methodology makes it accessible to both novice and experienced practitioners, facilitating its implementation and understanding. Furthermore, the ID3 algorithm excels when applied to datasets with a moderate number of instances and categorical attributes. It is particularly efficient in scenarios where class labels are discrete and well-defined.

A significant advantage of the ID3 algorithm is its capacity to generate decision trees that are inherently interpretable. The resulting trees present a human-readable representation of decision-making processes, aiding analysts and stakeholders in comprehending the underlying patterns captured by the model. This interpretability is crucial in domains where transparency and justification of decisions are paramount.

Despite its merits, the ID3 algorithm is not without limitations, and these shortcomings have driven the development of more advanced tree-building techniques. Notably, the algorithm struggles when confronted with continuous attributes. Since it inherently works with categorical attributes, the direct incorporation of continuous variables can be challenging. This limitation restricts its applicability to datasets containing both categorical and continuous attributes, which are common in real-world scenarios.

Additionally, the ID3 algorithm's eagerness to create deep decision trees can lead to a notable drawback: overfitting. The algorithm's tendency to generate intricate trees that perfectly fit the training data might result in poor generalization to new, unseen data. As the tree becomes increasingly complex, it could capture noise and anomalies present in the training dataset, ultimately hampering its predictive performance on previously unseen instances.

Furthermore, the ID3 algorithm exhibits a bias towards attributes with many distinct values. Such attributes can yield higher Information Gain due to their capacity to overfit the training data. This bias may not align with the actual importance of attributes in the underlying data distribution and can result in suboptimal tree structures.

The ID3 algorithm's strengths, including its simplicity, efficacy on categorical attributes, and generation of interpretable decision trees, are counterbalanced by limitations such as its struggle with continuous attributes and its predisposition to overfitting and bias towards certain attribute types. These drawbacks have spurred the development of more advanced algorithms that aim to address these shortcomings while preserving the algorithm's core principles.

In summation, the ID3 algorithm holds significance as a foundational technique for constructing decision trees. It operates iteratively, selecting attributes that maximize Information Gain and foster data partitioning. Although it bears certain limitations, it paved the way for the development of more sophisticated algorithms that refine and expand upon the principles established by ID3.

- **C4.5 algorithm** The C4.5 algorithm, an evolution of the ID3 method, was also developed by Ross Quinlan in 1993 [44] as an improved approach to constructing decision trees. C4.5 addresses the limitations of its predecessor, ID3, by introducing techniques to handle continuous attributes, mitigate overfitting, and improve the overall robustness of the resulting decision trees. This algorithm has had a profound impact on the field of machine learning and remains a foundational technique for building accurate and interpretable classification models.

  The C4.5 algorithm builds decision trees using a top-down, recursive approach, similar to ID3. However, C4.5 introduces several crucial enhancements to the tree-building process:

  1. **Handling Continuous Attributes:** Unlike ID3, which works primarily with categorical attributes, C4.5 is designed to handle continuous attributes seamlessly. It achieves this by converting continuous attributes into discrete intervals. The algorithm identifies potential split points, evaluates the Information Gain for each interval, and selects the best split, allowing for more flexible and accurate tree construction.
  2. **Gain Ratio:** C4.5 introduces the concept of Gain Ratio to address the bias of Information Gain towards attributes with many values. The Gain Ratio divides the Information Gain by the intrinsic information associated with the attribute, offering a normalized measure of the attribute's discriminatory power. This prevents the algorithm from favoring attributes with numerous values and promotes a fairer attribute selection process.
  3. **Pruning for Reduced Overfitting:** A critical advancement of C4.5 is its incorporation of post-processing pruning techniques. Pruning helps prevent overfitting by removing branches of the tree that do not contribute significantly to its predictive power. The algorithm constructs the complete decision tree and then prunes nodes based on a statistical test, optimizing the tree's complexity and enhancing its generalization to unseen data.
  4. **Continuous Split Selection:** C4.5 evaluates potential splits for continuous attributes by considering multiple split points and choosing the one that maximizes Information Gain or Gain Ratio. This approach improves the algorithm's ability to capture relationships within continuous attributes, making it well-suited for datasets with mixed attribute types.
  5. **Missing Values Handling:** C4.5 can handle instances with missing attribute values during both tree construction and classification. It intelligently distributes instances with missing values across multiple branches, ensuring accurate decision-making without excluding incomplete data.

  The C4.5 algorithm has left a significant imprint on the landscape of machine learning, fostering advancements that have rippled across various domains. Its notable contributions have augmented the algorithmic toolbox with enhancements that go beyond its predecessor, the ID3 algorithm. One of the paramount advantages lies in the algorithm's enriched robustness through the inclusion of continuous attribute handling and the introduction of the Gain Ratio. These pivotal improvements transcend the limitations of ID3 and extend the algorithm's applicability to datasets characterized by mixed attribute types, thereby broadening its reach and utility.

  Moreover, C4.5's introduction of post-processing pruning techniques has garnered a profound impact by combating the overfitting tendency prevalent in decision tree construction. This facet substantially bolsters the generalization capability of the resulting decision trees, endowing them with the ability to offer more accurate predictions on unseen data. The outcome of this advancement is twofold: the generated models attain enhanced reliability while circumventing the pitfalls of overfitting that might otherwise hinder their predictive performance.

Furthermore, the transparency and interpretability that C4.5's decision trees offer stand as invaluable contributions to the realm of machine learning. By yielding models that are readily comprehensible to both technical practitioners and non-technical stakeholders, the algorithm fosters informed decision-making and engenders a deeper understanding of the underlying processes shaping the classification outcomes.

The ripple effects of C4.5's innovations extend beyond the algorithm itself. It serves as a cornerstone upon which subsequent methodologies and techniques have been built. Enabling the evolution of sophisticated ensemble methods, such as Random Forests and Gradient Boosting, C4.5 has been instrumental in laying the groundwork for the broader advancement of machine learning. In essence, the C4.5 algorithm has not only provided an immediate leap forward in decision tree construction but has also catalyzed a cascade of developments that continue to shape the landscape of modern machine learning.

In summary, the C4.5 algorithm represents a pivotal advancement in the evolution of decision tree construction. By addressing the limitations of its predecessor and introducing techniques for handling continuous attributes and pruning, C4.5 has significantly contributed to the creation of accurate, interpretable, and generalizable classification models, while also setting the stage for more sophisticated ensemble methods.

- **CART Algorithm** The CART (Classification and Regression Trees) algorithm is a versatile and widely used technique for creating decision trees that cater to both classification and regression tasks. Proposed by Breiman, Friedman, Olshen, and Stone in 1984 [45], the algorithm offers a flexible framework that adapts to various types of data and objectives, making it a foundational tool in machine learning.

  The CART algorithm employs a binary recursive partitioning approach to construct decision trees. It starts with the entire dataset and recursively divides it into subsets based on the selected attribute and threshold value, optimizing a specific criterion at each step. The algorithm's key features include:

  1. **Splitting Criteria:** For classification tasks, CART employs the Gini impurity as the criterion to evaluate the homogeneity of a subset with respect to class labels. The Gini impurity measures the probability of an incorrect classification if an instance is randomly assigned a class label from the subset. For regression tasks, the mean squared error (MSE) is used to assess the variability of target values within a subset.
  2. **Binary Splitting:** CART's binary splitting nature ensures that each internal node is split into two child nodes, leading to a binary tree structure. At each step, the algorithm seeks the best attribute and threshold combination that optimizes the chosen splitting criterion, thereby minimizing impurity for classification or variability for regression.
  3. **Pruning for Generalization:** After constructing the full decision tree, CART applies a pruning process to eliminate branches that contribute little to improving the model's generalization to unseen data. Pruning involves iteratively removing nodes and evaluating their impact on a validation dataset, guided by performance metrics like cross-validation error.
  4. **Regression and Classification Adaptability:** One of CART's remarkable strengths is its adaptability to both classification and regression tasks. By altering the choice of impurity

measure (Gini impurity for classification or MSE for regression), the algorithm seamlessly tailors its approach to the nature of the problem.

The CART (Classification and Regression Trees) algorithm stands as a significant and transformative methodology within the realm of machine learning. Its contributions and impacts reverberate across diverse domains, rendering it a cornerstone of modern data analysis. A distinguishing feature of CART lies in its exceptional versatility, seamlessly adapting to both classification and regression tasks with equal efficacy. This adaptability endows practitioners with a unified framework that can be applied to a spectrum of challenges, bridging the gap between distinct problem types.

One of its standout strengths is its capacity to yield highly accurate models. Through its binary recursive partitioning approach, it strategically dissects the dataset into subsets, enabling the algorithm to discern intricate relationships inherent in the data. As a consequence, the resultant decision trees possess a predictive accuracy that often rivals more complex machine learning models.

Moreover, CART's decision trees are renowned for their inherent interpretability and transparency. These models are capable of delivering insights into the decision-making process, providing a window into the logic governing classification or regression outcomes. This transparency is particularly vital in applications where comprehensible explanations are of paramount importance.

The concept of pruning, integral to the CART algorithm, bolsters its impact further. By trimming branches that do not significantly contribute to model performance, pruning strikes a delicate equilibrium between prediction accuracy and the risk of overfitting, ultimately fostering better generalization to unseen data. This characteristic enhances the algorithm's applicability to real-world scenarios where robustness and adaptability are essential.

The legacy of this algorithm extends beyond its immediate applications. Its foundational ideas have catalyzed the development of ensemble methods, most notably Random Forests, which harness the collective strength of multiple decision trees. This lineage of techniques further accentuates the algorithm's enduring influence on the evolution of machine learning methodologies.

The CART algorithm's significance lies in its adaptability, accuracy, interpretability, and role in inspiring subsequent advancements. Its impact reverberates across academia, industry, and data science practice, and its versatility continues to make it a cornerstone of machine learning endeavors, continuing to shape and enrich the landscape of modern data analysis.

In essence, the CART algorithm has left an indelible mark on the landscape of machine learning. Its adaptability, robustness, and contribution to both classification and regression tasks have not only empowered practitioners with an effective tool but have also laid the groundwork for further advancements, ultimately shaping the trajectory of modern machine learning practices.

This article is recommended reading if you want to see a comparison between the 3 of them [46]

3.5.1.2 Number of max child node selection

The number of sons (or child nodes) each internal node can have in a decision tree depends on the splitting strategy employed and the nature of the data. Generally, decision trees can have binary splits (two child nodes per parent node) or multiway splits (more than two child nodes per parent node).

- **Binary Splits:** In binary splits, each internal node divides the data into two subsets based on a feature condition. This is the most common approach in decision trees, as it simplifies the decision-making process and tree structure. Binary splits are often used in algorithms like CART (Classification and Regression Trees).

- **Multiway Splits:** In some cases, decision trees can have more than two child nodes per parent node, resulting in multi way splits. This approach might be used when dealing with categorical features with more than two categories or when there are multiple potential branches based on a feature condition.

Selecting the number of maximum sons on the tree affects heavily in complexity, computation time, memory and overall understanding of the tree:

- **Tree Structure Complexity:** A higher number of maximum child nodes increases the potential branching in the tree, leading to a more complex tree structure. Complexity refers to the number of nodes and branches in the tree. A complex tree might capture intricate relationships but could also be prone to overfitting [47].
- **Computation Time:** Building and training a decision tree with more maximum child nodes can require more computational time. The algorithm needs to evaluate more potential feature conditions and thresholds, resulting in increased processing time [48].
- **Memory Usage:** A decision tree with more maximum child nodes can potentially consume more memory due to the increased number of nodes and branches. Each node requires storage for feature conditions, impurity measures, and pointers to child nodes.
- **Generalization vs. Overfitting:** A tree with a higher number of maximum child nodes might capture intricate patterns in the training data, potentially leading to better fitting of training data (lower training error). However, a very complex tree is more likely to overfit, performing poorly on unseen data (higher test error). Check [49] for more detailed information.
- **Prone to Instability:** A higher number of maximum child nodes can make the decision tree more sensitive to small variations in the data. Small changes in the training data could lead to different branches being selected, resulting in instability.
- **Pruning Impact:** Pruning, the process of removing unnecessary branches from the tree, becomes more critical in controlling complexity when the tree has a higher number of maximum child nodes. Pruning helps prevent overfitting by simplifying the tree without significantly sacrificing predictive power. There are several prunning techniques, see [50] for more information.

In practice, the choice of the number of maximum child nodes is often guided by a balance between capturing important relationships and preventing overfitting [51]. Hyperparameter tuning techniques can help determine the optimal tree complexity that generalizes well to unseen data while avoiding excessive complexity that hampers model performance, computation time, and memory usage as seen in [52].

3.5.2 Coded implementation

The following code implements a decision tree construction algorithm, particularly suited for categorical classification tasks (done for visualization purposes). It operates by recursively splitting the dataset based on features to maximize information gain, applying the Gini impurity, the entropy impurity or by classification error reduction, depending on the chosen method. This implementation is only meant for categorical variables so to make it work correctly with continuous variables, they should be bucketed first. The code consists of several functions:

3.5.3 The **decision_tree** Function

```
decision_tree <- function (data, classy, m, method = "entropy", details = FALSE, waiting =
    TRUE){

  if(details){
    console.log("\nEXPLANATION")
    hline()
    hline()
    console.log("\nStep 0:")
    console.log("    - Set the dataframe as parent node. The original dataframe is set as node
     0.")
    console.log("\nStep 1:")
    console.log("    - If data is perfectly classified, go to step 4.")
    console.log("    - If data is not classified, create all the possible combinations of
    values for each variable.")
    console.log("      Each combination stablishs the division of the son nodes, being \"m\"")
    console.log("      numbers of divisions performed.")
    console.log("Step 2:")
    console.log("    - Calculate the information gain for each combination.")
    console.log("      The \"method\" method is used to calculate the information gain.")
    console.log("Step 3:")
    console.log("    - Select the division that offers the most information gain for each
    variable.")
    console.log("    - Select the division that offers the most information gain among the
    best of each variable.")
    console.log("    - For each son of the division add the node to the tree and go to step 1
    with the filtered dataset.")
    console.log("Step 4:")
    console.log("    - This branch is finished. The next one in preorder will be evaluated\n\n
    ")
    console.log("Step 5:")
    console.log("    - Print results\n\n")
    hline()
    hline()
    console.log("\n IMPORTANT!!\n\n")
    console.log("    - The objective is to understand how decission trees work. The stopping
    condition is to have PERFECT LEAFES.")
    console.log("      If \"data\" is not perfectly classifiable, the code WILL NOT FINISH!!\n
    \n")
    console.log("    - It is important to understand that the code flow is recursive,")
    console.log("      meaning the tree is traversed in preorder (first, the root node is
    visited, then the children from left to right).")
    console.log("      So, when the information is categorized in step 1, this order will be
    followed. \n\n")
    hline()
    hline()
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  tree_strctr <- list(list(0))
  result <- aux_decision_tree(data,classy, m, method, tree_strctr, id = 0, id_f = 0, h = 0,
    details, waiting)
  result <- result[2:length(result)]
  result <- structure(result, class = "tree_struct")

  if(details){
    hline()
    console.log("\nStep 5:")
    console.log("This is the structure of the decission tree:")
    print(result)
  }
  return(result)
}
```

Listing 3.11: decision_tree Function

The `decision_tree` function is the one in charge of initializing and displaying a decision tree. Here's a detailed explanation:

1. **Parameters**:
   - `data`: The dataset containing feature values and the class variable.
   - `classy`: The name of the class or target variable in the dataset, it must be the name of one of the columns in `data`.
   - `m`: The maximum number of child nodes for each node of the decision tree.
   - `method`: The impurity measure method used for tree construction (entropy by default).
2. **Initialization**:
   - It initializes an empty list `chosen` to store selected feature combinations.
   - It initializes a `tree_strctr` list with a single element representing the root node (ID 0) This list will contain a list for each of the tree levels, each of this inner lists will contain lists with the information of each node. All the information needed to reconstruct the tree will be here.
3. **Decision Tree Construction**: The decision tree construction is carried out by calling the `aux_decision_tree` function. The result of the decision tree construction is stored in the `result` variable. It contains the `tree_strctr` after the tree has been constructed.
4. **Displaying Tree Information**: The function iterates over the generated `result` to display information about the decision tree. For each level it prints the number of children (nodes) and the feature used for splitting. It then provides details about each child node, including its ID, parent ID, filtering criteria, and the data subset associated with the child node.

3.5.4 The `aux_decision_tree` Function

```
aux_decision_tree <- function (data, classy, m, method, tree_strctr, id, id_f, h, details,
    waiting){
  if (length(unique(data[, classy])) < 2){
    if (details){
      console.log("\nSteps 1 and 4:")
      console.log("Data is classified.")
    }
    return (tree_strctr)
  }

  if (details){
    console.log("\nStep 0:")
    console.log("\nData:")
    print(data)
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  if (details){
    hline()
    console.log("\nSteps 1 and 2:")
  }
  candidates <- mapply(
    function (n, name){
      df <- all_combs(unique(n),m)
      df <- gain_method(df, data, classy, method, name)
      df$classifier <- name
      if (details){
        console.log(paste("Combinations for", name))
        print(df)
        console.log("")
      }
      m <- which(df$Gain == max(df$Gain))
      indice_fila_max_dashes <- which.min(sapply(df[m, ], function(column) sum(column == "---"
    )))
      max_value <- df[m, ][indice_fila_max_dashes, ]
      columnas_con_dashes <- colnames(max_value)[apply(max_value == "---", 2, all)]
      max_value <- max_value[, !colnames(max_value) %in% columnas_con_dashes]
      candidato <- list(c(max_value[ 1 : (length(max_value) - 2) ]), as.numeric(max_value[
    length(max_value)-1]), as.character(max_value[length(max_value)]))
    },
    data[, !colnames(data) %in% classy],  #todas las columnas menos classy
    colnames(data)[!colnames(data) %in% classy], #los nombres de las columnas
    SIMPLIFY = TRUE,
    USE.NAMES = FALSE
  )
  if (details){
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  candidates <- t(data.frame(candidates))
  colnames(candidates) <- c("Sons", "Gain", "Classifier")
  max_gain <- list(candidates[which.max(candidates[, "Gain"]), ])

  if(details){
    hline()
    console.log("\nStep 3:")
    console.log("List of best candidates (1 for each variable):")
    print(candidates)
    console.log("\nThe division with the most information gain is chosen:")
    console.log(paste("    - Classifier =",max_gain[[1]][[3]][1]))
    console.log(paste("    - Information gain =",round(max_gain[[1]][[2]][1],3)))
    console.log("    - Sons =")
    print(unlist((max_gain[[1]][[1]])))
    if (waiting){
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  h <- h + 1
  if (length(tree_strctr)-1 < h){
    tree_strctr <- append(tree_strctr, list(list()))
  }

  name <- max_gain[[1]][[3]]
  for (i in 1:length(max_gain[[1]][[1]])){
    df <- subset(data, get(name) %in% strsplit(as.character(max_gain[[1]][[1]][i]), " ")[[1]])
    rownames(df) <- NULL
    tree_strctr[[1]][[1]]<- tree_strctr[[1]][[1]]+1
    tree_strctr[[h+1]] <- append(tree_strctr[[h+1]], list(list(df, tree_strctr[[1]][[1]], id_f
    , h, max_gain[[1]][[3]],max_gain[[1]][[1]][[i]],  max_gain[[1]][[2]])))

    tree_strctr <- aux_decision_tree(df, classy, m, method, tree_strctr, tree_strctr
    [[1]][[1]], tree_strctr[[1]][[1]], h, details, waiting)
  }
  return (tree_strctr)
}
```

Listing 3.12: aux_decision_tree Function

The `aux_decision_tree` function is a recursive function that constructs the decision tree by selecting the best feature combinations for splitting. It considers impurity measures and recursively splits the dataset into subsets. Here's an extensive explanation:

1. **Stopping Condition**: The function checks if there are fewer than two unique class values in the current dataset. If so, it returns the current `tree_strctr`, indicating that no further splitting is needed. We have decided that for educational purpouses (make it clearer) the leaf nodes must be perfect.

2. **Feature Combination Selection**: The function generates feature combinations using the `all_combs` function and calculates the information gain (impurity reduction) for each combination using the `gain_method` function. It selects the combination with the highest gain, representing the best feature to split on. In case 2 values provide the same gain value, the one that creates the less sons for that node will be selected. If both create the same amount of sons, one will be chosen randomly.

3. **Tree Structure Update**: The function updates the structure of the decision tree by incrementing the child count for the parent node and adding a new node to the tree. The new node contains information about the data subset associated with it, the parent node ID, the filtering criteria, and the feature used for splitting.

4. **Recursion for Splitting**: For each unique value in the selected feature, the function splits the dataset into subsets based on that value. It recursively calls itself for each subset, constructing child nodes of the current node. Once the recursion is done, the final child node returns the `tree_strctr` as the result of the function, which has been updated throughout the process.

3.5.5 The `gain_method` Function

```
gain_method <- function(da, data, classy, method, name){
  if (is.character(da)){
```

```r
    df <- as.data.frame(matrix(da, nrow = 1, ncol = 1))
    df$Gain <- 0
    return (df)
  }
  da$Gain <- apply(
    da,
    1,
    function(n, data, classy){
      n <- as.vector(n[n != "---"])
      switch (tolower(method),
              "entropy" = entropy(n, data, classy, name),
              "gini"    = gini(n, data, classy, name),
              "error"   = error(n, data, classy, name),
              stop("Unknown method")
      )
    },
    data = data,
    classy = classy,
    simplify = TRUE
  )
  return(da)
}
```

Listing 3.13: gain_method Function

The gain_method function is responsible for calculating the gain (impurity reduction) for a given set of feature combinations. It determines which feature combination is the best split for constructing a decision tree based on a specified impurity measure. Here's a detailed explanation:

1. **Parameters**:
   - da: A data frame containing feature combinations.
   - data: The original dataset.
   - classy: The name of the class or target variable in the dataset.
   - method: The impurity measure to be used (entropy, gini or error).
   - name: The name of the feature being evaluated.
2. **Loop Over Feature Combinations**: The function uses the apply function to loop over each row (feature combination) in the da data frame. For each combination, it extracts the non-dash values (actual values) from the combination and calls the impurity measure function (entropy, gini, or error) based on the specified method. The apply function returns a vector with all the gain values.
3. **Return**: The da data frame is updated with an additional column named Gain, which contains the gain values calculated for each feature combination. Finally, the updated da data frame is returned as the result.

3.5.6 The entropy, gini, and error Functions

These three functions (entropy, gini, and error) are used to calculate impurity measures for a given set of class values. Each function computes a different impurity measure, which is used to evaluate the impurity or disorder of a dataset or a subset of data.

3.5.6.1 The entropy Function

```r
entropy <- function(n, data, classy, name){
  ent_hijos <- vector(mode="integer", length = length(n))
  entp <- 0
  valores <- table(data[,classy])
  for (i in 1:length(valores)){
    entp <- entp - (valores[i]/sum(valores)* log2(valores[i]/sum(valores)))
  }
  ganancia = entp[[1]]
  for (i in 1:length(n)){
    subset_data <- table(data[data[[name]] %in% unlist(strsplit(n[i], " ")), ][,classy])
    for (j in subset_data){
      ent_hijos[i] <- ent_hijos[i] - (j/sum(subset_data)* log2(j/sum(subset_data)))
    }
    ganancia = ganancia - ((sum(subset_data)/sum(valores)) * ent_hijos[i])
  }
  return(ganancia)
}
```

Listing 3.14: entropy Function

The entropy function calculates the impurity measure based on information entropy. It uses the following parameters:

- n: A vector of class values.
- data: The original dataset.
- classy: The name of the class or target variable in the dataset.
- name: The name of the feature being evaluated.

It begins by computing the entropy of the entire class variable classy using the formula for entropy:

$$E(S) = -\sum_{i=1}^{c} p_i \cdot \log_2(p_i)$$

Where c is the number of unique class values, and p_i is the proportion of instances belonging to class i. Next, it calculates the entropy for each subset of the data obtained by splitting based on the values of the evaluated feature. The gain is computed by subtracting the weighted average of subset entropies from the entropy of the entire class variable. The function returns the gain value.

3.5.6.2 The gini Function

```r
gini <- function(n, data, classy, name){
  gin_hijos <- rep(1, length(n))
  ginp <- 1
  valores <- table(data[,classy])
  for (i in 1:length(valores)){
    ginp <- ginp - (valores[i]/sum(valores) * (valores[i]/sum(valores)))
  }
  ganancia = ginp[[1]]
  for (i in 1:length(n)){
    subset_data <- table(data[data[[name]] %in% unlist(strsplit(n[i], " ")), ][,classy])
    for (j in subset_data){
      gin_hijos[i] <- gin_hijos[i] - ((j/sum(subset_data)) * (j/sum(subset_data)))
    }
    ganancia = ganancia - ((sum(subset_data)/sum(valores)) * gin_hijos[i])
  }
  return(ganancia)
}
```

Listing 3.15: gini Function

The `gini` function calculates the impurity measure based on the Gini impurity. It uses the same parameters as the `entropy` function. It starts by computing the Gini impurity of the entire class variable `classy` using the formula:

$$G(S) = 1 - \sum_{i=1}^{c} (p_i)^2$$

Similar to entropy, it then calculates the Gini impurity for each subset obtained by splitting based on feature values. The gain is computed by subtracting the weighted average of subset Gini impurities from the Gini impurity of the entire class variable. The function returns the gain value.

3.5.6.3 The **error** Function

```
error <- function(n, data, classy, name){
  err_hijos <- rep(1, length(n))
  errp <- 1
  valores <- table(data[,classy])
  for (i in 1:length(valores)){
    errp <- errp - (valores[i]/sum(valores) * (valores[i]/sum(valores)))
  }
  ganancia = errp[[1]]
  for (i in 1:length(n)){
    subset_data <- table(data[data[[name]] %in% unlist(strsplit(n[i], " ")), ][,classy])
    err_hijos[i] <- max(subset_data/sum(subset_data))
    ganancia = ganancia - ((sum(subset_data)/sum(valores)) * err_hijos[i])
  }
  return(ganancia)
}
```

Listing 3.16: error Function

The `error` function calculates the impurity measure based on the error rate. It uses the same parameters as the `entropy` function and the `gini` function. It calculates the error rate as the proportion of instances that do not belong to the most frequent class in the dataset. The gain is computed by subtracting the weighted average of error rates for subsets obtained by splitting based on feature values from the error rate of the entire class variable. The function returns the gain value.

3.5.7 The **all_combs** Function

```
all_combs <- function(vp, k){
  v = c()
  if (k > length(vp)){
    k = length(vp)
  }
  combinations<- data.frame()
  combinations <- comb(v, vp, k, combinations)
  combinations <- apply(combinations, 1, sort, simplify = TRUE)
  combinations <- data.frame(t(combinations))
  combinations <- combinations[!duplicated(combinations), ]
  rownames(combinations) <- NULL

  return(combinations)
}
```

Listing 3.17: all_combs Function

The `all_combs` function utilizes the `comb` function to generate all possible feature combinations for categorical attributes. Here's how it works:

1. **Parameters**:
   - `vp`: A vector of potential values for the feature.
   - `k`: The maximum number of values to include in each combination.
2. **Generate Combinations**: The `comb` function is called recursively with the following parameters:
   - `v`: The accumulator vector, initially empty.
   - `vp`: The vector of potential values for the feature.
   - `k`: The maximum number of values to include in each combination.
   - `combinations`: The data frame to store generated combinations.
3. **Handling Duplicate Combinations**: After generating combinations, the function removes duplicate rows from the `combinations` data frame. Duplicate rows may occur when multiple paths in the decision tree lead to the same combination of feature values in different orders. This can happen in some cases because A, BC, D and BC, A, D are effectively the same. The order of the elements does not really change the decision tree.
4. **Return**: The function returns the `combinations` data frame, which contains all unique feature combinations.

3.5.8 The `comb` Function

```
comb <- function(v, vp, k, combinations){
  if (k > 1){
    if (length(vp) == 0){
      v <- c(v,"---")
      comb(v,vp,k-1, combinations)
    }
    else if(length(vp) == 1){
      v <- c(v, paste(vp, collapse = " "))
      comb(v,c(),k-1, combinations)
    }
    else {
      vec <- c(v, paste(vp, collapse = " "), rep(c("---"), each = k-1))
      combinations <- rbind (combinations, vec)
      combi <- vector("list", length(vp))
      l <- ifelse(length(vp)%%2 == 0, length(vp)/2, length(vp)/2 - 0.5)

      for (i in 1:l) {
        temp <- combn(vp, i, paste, collapse = " ")
        if (length(vp)%%2 == 0 && nchar(temp[1]) == l*2 - 1){
          half <- split(temp, f = ifelse(seq_along(temp) <= length(temp)/2, "first", "second")
    )
```

```
          temp <- half$first
        }
        combi[[i]] <- temp
      }
      df <- unlist(combi)
      for (i in 1:length(df)){
        n <- strsplit(df[i], " ")[[1]]
        new_data <- setdiff(vp, n)
        combinations <- comb(c(v,paste(n, collapse = " ")),new_data, k-1, combinations)
      }
      return (combinations)
    }
  }
  else if (k == 1){
    ifelse(length(vp) > 0, v <- c(v, paste(vp, collapse = " ")) , v <- c(v,"---"))
    combinations <- rbind(combinations, v)
    return(combinations)
  }
}
```

Listing 3.18: comb Function

The `comb` function generates all possible combinations of values from a given set of categorical values for a feature. It creates these combinations as strings, where each string represents a possible combination of values. It aims to only generate combinations that have not been generated previously (this is done by considering the combinations as pairs made of one element and a combination, which can be subsequently divided into another pair and so on), but in some cases it does. When the code flow goes back to the `all_combs` function, these duplicated are eliminated. Here's a step-by-step explanation of how it works:

1. **Parameters**:
   - `v`: A vector that accumulates values to build combinations.
   - `vp`: A vector of potential values for the feature.
   - `k`: The maximum number of groups in which values can be combined. Each iteration it is one value lower
   - `combinations`: A data frame that stores generated combinations.
2. **Base Case**:
   - If `k` is greater than 1, the function recursively generates combinations until `k` becomes 1.
   - If `vp` is empty (no more potential values to add), it adds a placeholder "—" to `v`, effectively marking the absence of a value for this combination.
   - If `vp` contains exactly one value, it concatenates this value to the `v` vector to form a combination string.
   - If `vp` contains more than one value, it creates combinations by selecting one value and recursively generating combinations for the remaining values.
   - Combinations are represented as strings, with values separated by spaces.
3. **Combination Storage**: Each generated combination is stored in the `combinations` data frame as a row. After generating all combinations for a specific number of values (`k`), the `combinations` data frame is updated.

4. **Recursion**: The function uses recursion to explore all possible combinations, varying the number of values included in each combination (`k`). For each combination, it explores different ways of choosing values from the `vp` vector.

5. **Termination and Return**: When `k` reaches 1 (the base case), the function appends the final combinations (strings) to the `combinations` data frame. The `combinations` data frame is returned as the result.

In summary, the `comb` and `all_combs` functions efficiently generate all possible feature combinations from categorical values while ensuring that duplicate combinations are removed. These combinations are crucial for evaluating impurity measures during the decision tree construction process.

These functions collectively facilitate decision tree construction, feature combination selection, impurity measurement, and combination generation. All of them combined generate the best possible decision tree, if there is.

3.5.8.1 Printing the tree

In order to easily visualize the structure of the decision tree, the following function has been developed. It overwrites the default print function when the printed object is a `tree_struct` class object.

```
print.tree_struct <- function(x, ...){
  for (i in 1:length(x)){
    console.log(paste("Height", i ,"has", length(x[[i]]),"sons, divided by", x[[i]][[1]][[5]],
    ":\n\n"))
    for (j in 1:length(x[[i]])){
      console.log(paste("Son", x[[i]][[j]][[2]] ,"(Whose father node is", x[[i]][[j]][[3]], ")
      filters by \"", x[[i]][[j]][[6]], "\". It contains:\n"))
      print(x[[i]][[j]][[1]])
      console.log("")
    }
    console.log("")
  }
}
```

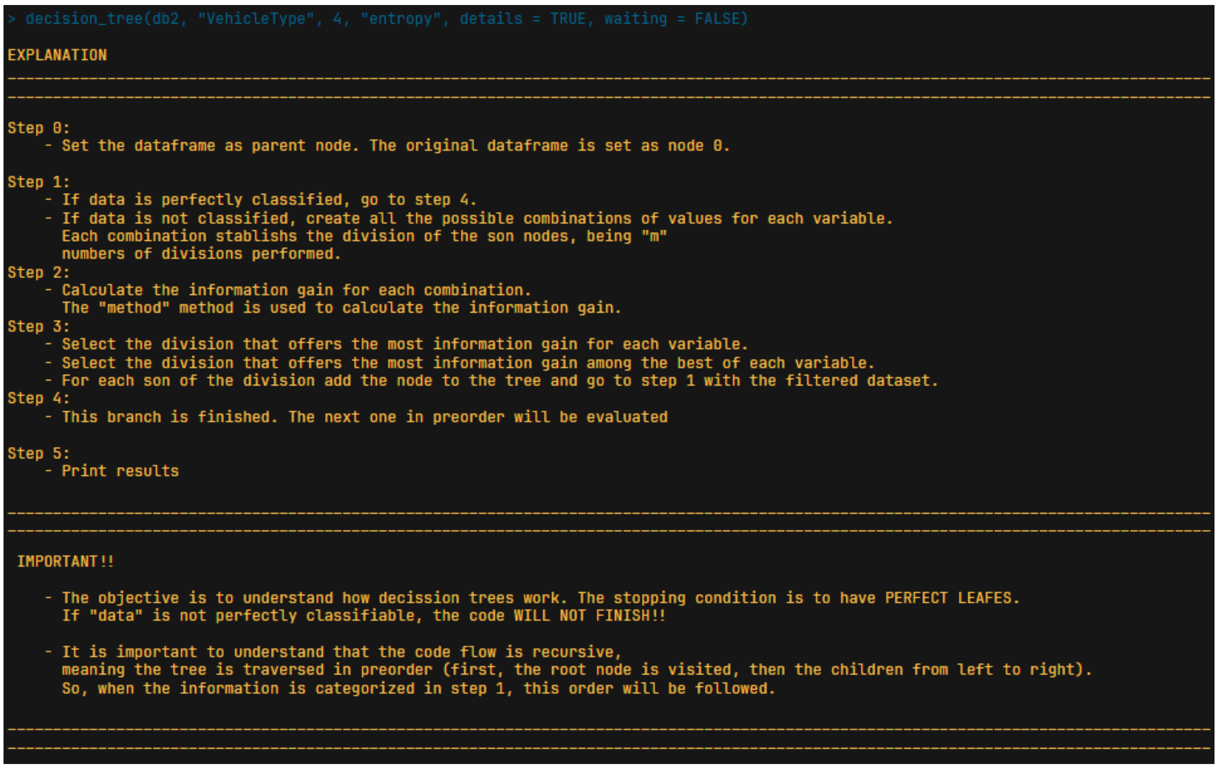
Listing 3.19: comb Function

3.5.9 Results

To gain a deeper insight into the inner workings of this code and comprehend its step-by-step processes, let's delve into its mechanics by exploring a simplified example. This practical illustration will illuminate the code's functionality and provide a clearer understanding of its operations. The 3.6 dataset will be used:

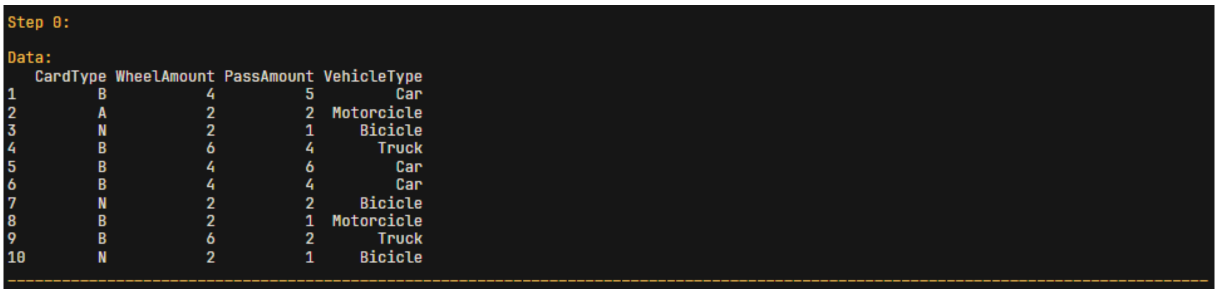
Table 3.6: db2

CardType	WheelAmount	PassAmount	VehicleType
B	4	5	Car
A	2	2	Motorcicle
N	2	1	Bicicle
B	6	4	Truck
B	4	6	Car
B	4	4	Car
N	2	2	Bicicle
B	2	1	Motorcicle
B	6	2	Truck
N	2	1	Bicicle



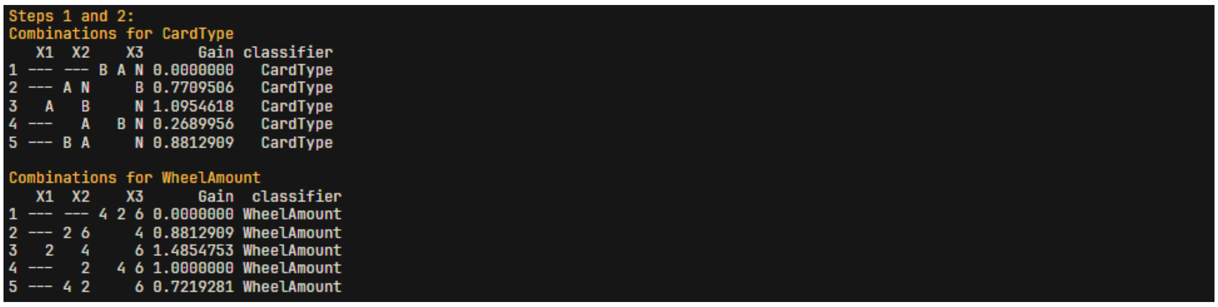
```
> decision_tree(db2, "VehicleType", 4, "entropy", details = TRUE, waiting = FALSE)

EXPLANATION
------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------

Step 0:
    - Set the dataframe as parent node. The original dataframe is set as node 0.

Step 1:
    - If data is perfectly classified, go to step 4.
    - If data is not classified, create all the possible combinations of values for each variable.
      Each combination stablishs the division of the son nodes, being "m"
      numbers of divisions performed.
Step 2:
    - Calculate the information gain for each combination.
      The "method" method is used to calculate the information gain.
Step 3:
    - Select the division that offers the most information gain for each variable.
    - Select the division that offers the most information gain among the best of each variable.
    - For each son of the division add the node to the tree and go to step 1 with the filtered dataset.
Step 4:
    - This branch is finished. The next one in preorder will be evaluated

Step 5:
    - Print results

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

 IMPORTANT!!

    - The objective is to understand how decission trees work. The stopping condition is to have PERFECT LEAFES.
      If "data" is not perfectly classifiable, the code WILL NOT FINISH!!

    - It is important to understand that the code flow is recursive,
      meaning the tree is traversed in preorder (first, the root node is visited, then the children from left to right).
      So, when the information is categorized in step 1, this order will be followed.

------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------
```

Figure 3.32: Decision Tree Results_1



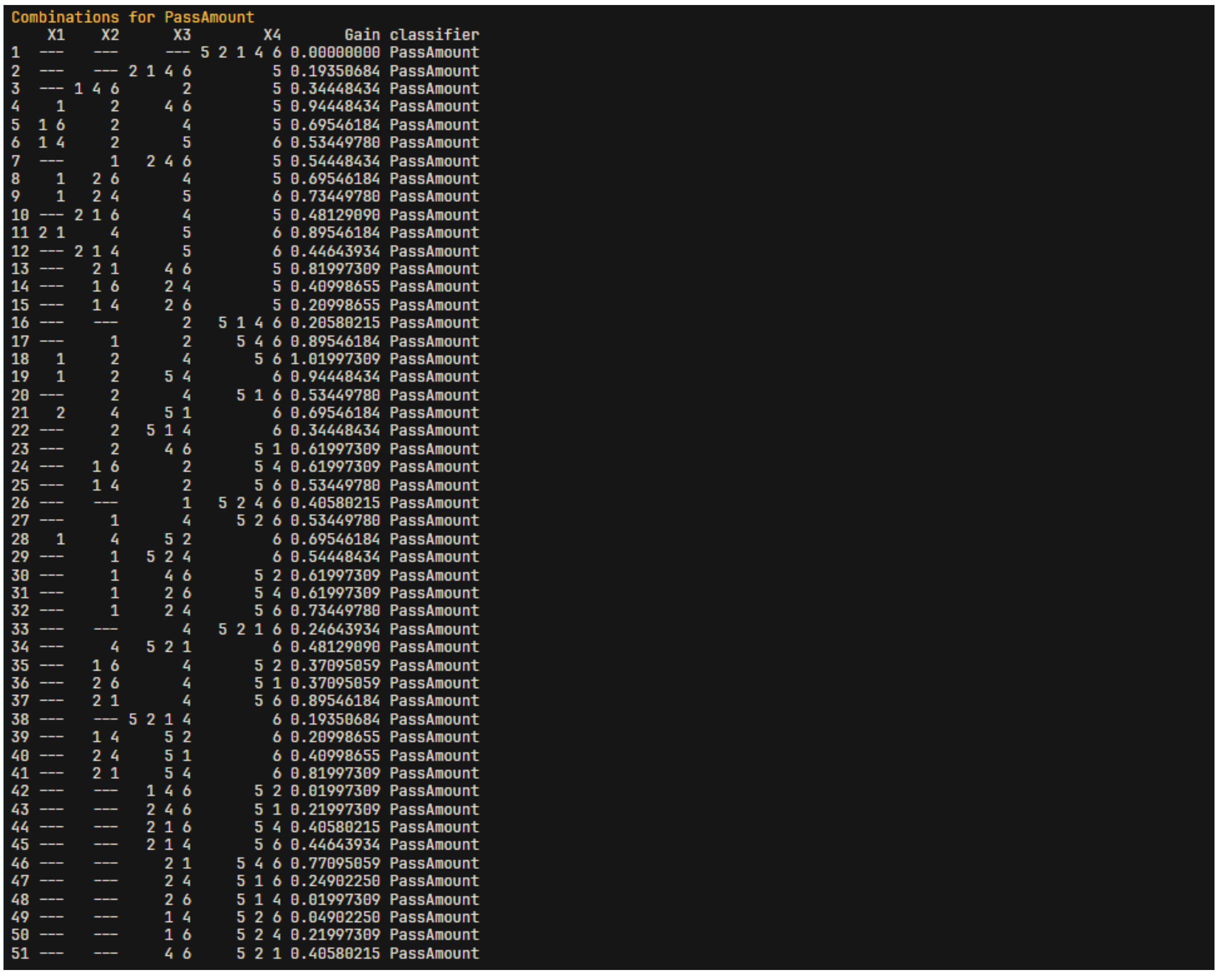
```
Step 0:

Data:
   CardType WheelAmount PassAmount VehicleType
1         B           4          5         Car
2         A           2          2  Motorcicle
3         N           2          1     Bicicle
4         B           6          4       Truck
5         B           4          6         Car
6         B           4          4         Car
7         N           2          2     Bicicle
8         B           2          1  Motorcicle
9         B           6          2       Truck
10        N           2          1     Bicicle
------------------------------------------------------------------------------------------------------------------------------------
```

Figure 3.33: Decision Tree Results_2



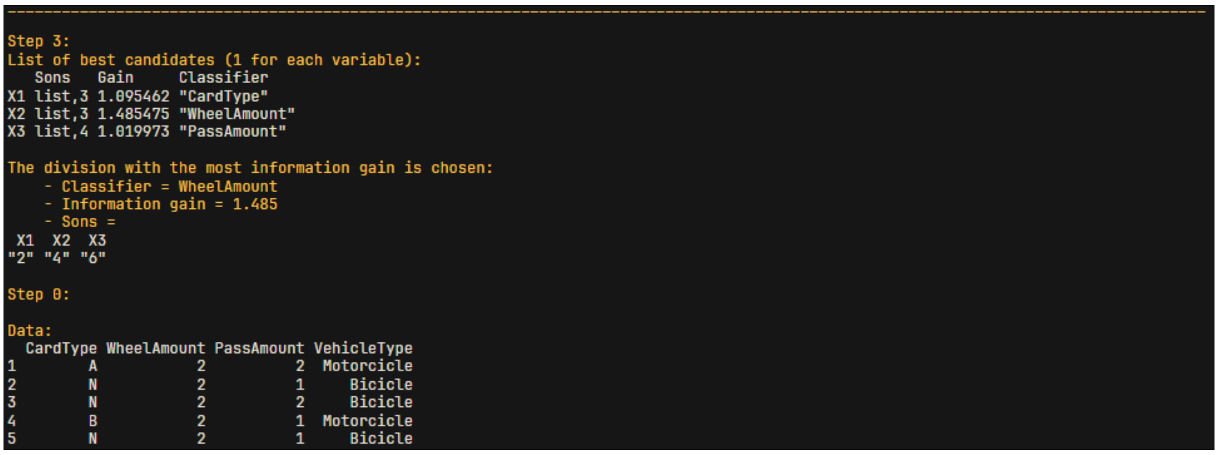
```
Steps 1 and 2:
Combinations for CardType
   X1  X2    X3      Gain classifier
1 --- --- B A N 0.0000000   CardType
2 --- A N     B 0.7709506   CardType
3   A   B     N 1.0954618   CardType
4 ---   A   B N 0.2689956   CardType
5 --- B A     N 0.8812909   CardType

Combinations for WheelAmount
   X1  X2    X3      Gain  classifier
1 --- --- 4 2 6 0.0000000 WheelAmount
2 --- 2 6     4 0.8812909 WheelAmount
3   2   4     6 1.4854753 WheelAmount
4 ---   2   4 6 1.0000000 WheelAmount
5 --- 4 2     6 0.7219281 WheelAmount
```

Figure 3.34: Decision Tree Results_3



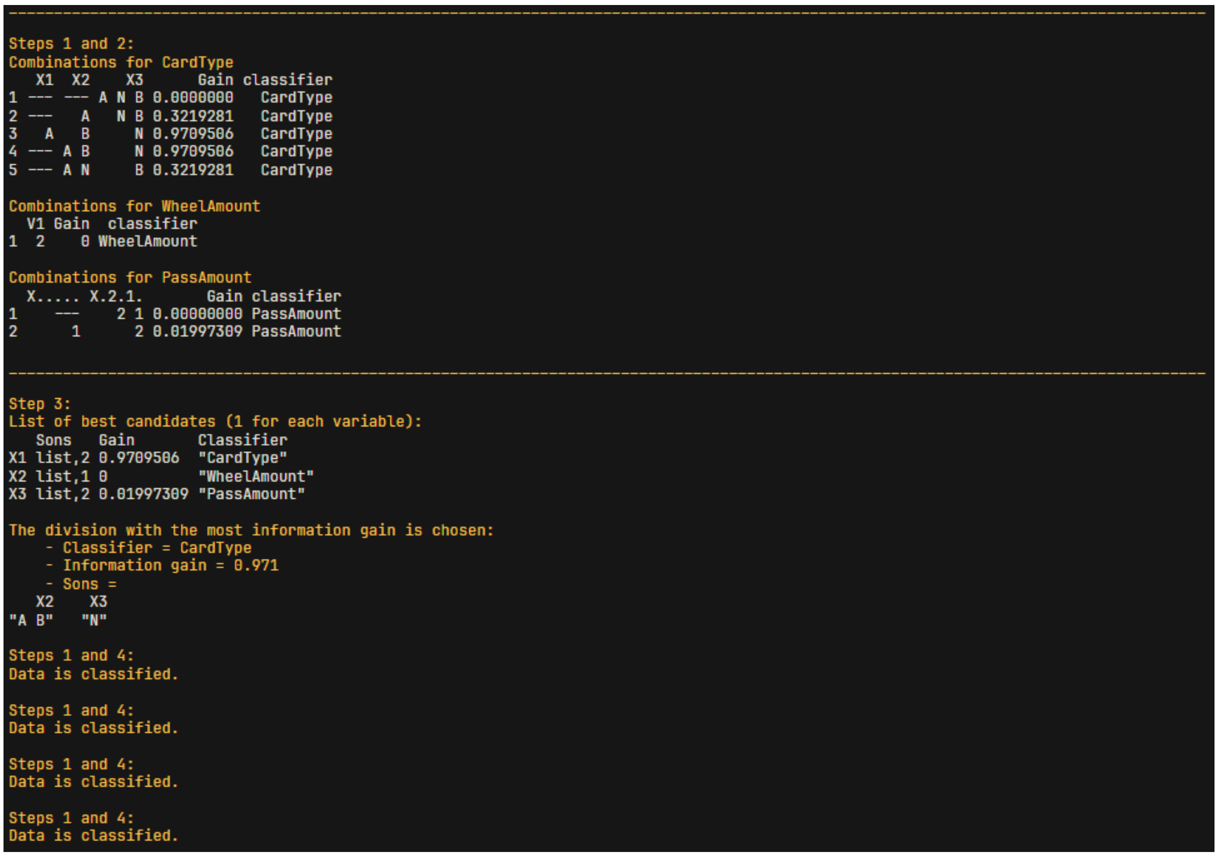
```
Combinations for PassAmount
    X1    X2      X3        X4       Gain classifier
1  ---   ---     --- 5 2 1 4 6 0.00000000 PassAmount
2  ---   --- 2 1 4 6         5 0.19350684 PassAmount
3  --- 1 4 6       2         5 0.34448434 PassAmount
4    1     2     4 6         5 0.94448434 PassAmount
5  1 6     2       4         5 0.69546184 PassAmount
6  1 4     2       5         6 0.53449780 PassAmount
7  ---     1   2 4 6         5 0.54448434 PassAmount
8    1   2 6       4         5 0.69546184 PassAmount
9    1   2 4       5         6 0.73449780 PassAmount
10 --- 2 1 6       4         5 0.48129090 PassAmount
11 2 1     4       5         6 0.89546184 PassAmount
12 --- 2 1 4       5         6 0.44643934 PassAmount
13 ---   2 1     4 6         5 0.81997309 PassAmount
14 ---   1 6     2 4         5 0.40998655 PassAmount
15 ---   1 4     2 6         5 0.20998655 PassAmount
16 ---   ---       2   5 1 4 6 0.20580215 PassAmount
17 ---     1       2     5 4 6 0.89546184 PassAmount
18   1     2       4       5 6 1.01997309 PassAmount
19   1     2     5 4         6 0.94448434 PassAmount
20 ---     2       4     5 1 6 0.53449780 PassAmount
21   2     4     5 1         6 0.69546184 PassAmount
22 ---     2   5 1 4         6 0.34448434 PassAmount
23 ---     2     4 6       5 1 0.61997309 PassAmount
24 ---   1 6       2       5 4 0.61997309 PassAmount
25 ---   1 4       2       5 6 0.53449780 PassAmount
26 ---   ---       1   5 2 4 6 0.40580215 PassAmount
27 ---     1       4     5 2 6 0.53449780 PassAmount
28   1     4     5 2         6 0.69546184 PassAmount
29 ---     1   5 2 4         6 0.54448434 PassAmount
30 ---     1     4 6       5 2 0.61997309 PassAmount
31 ---     1     2 6       5 4 0.61997309 PassAmount
32 ---     1     2 4       5 6 0.73449780 PassAmount
33 ---   ---       4   5 2 1 6 0.24643934 PassAmount
34 ---     4   5 2 1         6 0.48129090 PassAmount
35 ---   1 6       4       5 2 0.37095059 PassAmount
36 ---   2 6       4       5 1 0.37095059 PassAmount
37 ---   2 1       4       5 6 0.89546184 PassAmount
38 ---   --- 5 2 1 4         6 0.19350684 PassAmount
39 ---   1 4     5 2         6 0.20998655 PassAmount
40 ---   2 4     5 1         6 0.40998655 PassAmount
41 ---   2 1     5 4         6 0.81997309 PassAmount
42 ---   ---   1 4 6       5 2 0.01997309 PassAmount
43 ---   ---   2 4 6       5 1 0.21997309 PassAmount
44 ---   ---   2 1 6       5 4 0.40580215 PassAmount
45 ---   ---   2 1 4       5 6 0.44643934 PassAmount
46 ---   ---     2 1     5 4 6 0.77095059 PassAmount
47 ---   ---     2 4     5 1 6 0.24902250 PassAmount
48 ---   ---     2 6     5 1 4 0.01997309 PassAmount
49 ---   ---     1 4     5 2 6 0.04902250 PassAmount
50 ---   ---     1 6     5 2 4 0.21997309 PassAmount
51 ---   ---     4 6     5 2 1 0.40580215 PassAmount
```

Figure 3.35: Decision Tree Results_4



```
---------------------------------------------------------------------------------------------------------------------------------------

Step 3:
List of best candidates (1 for each variable):
   Sons   Gain     Classifier
X1 list,3 1.095462 "CardType"
X2 list,3 1.485475 "WheelAmount"
X3 list,4 1.019973 "PassAmount"

The division with the most information gain is chosen:
    - Classifier = WheelAmount
    - Information gain = 1.485
    - Sons =
 X1  X2  X3
"2" "4" "6"

Step 0:

Data:
  CardType WheelAmount PassAmount VehicleType
1        A           2          2  Motorcicle
2        N           2          1     Bicicle
3        N           2          2     Bicicle
4        B           2          1  Motorcicle
5        N           2          1     Bicicle
```

Figure 3.36: Decision Tree Results_5



```
---------------------------------------------------------------------------------------------------------------------------------------

Steps 1 and 2:
Combinations for CardType
   X1  X2    X3      Gain classifier
1 --- --- A N B 0.0000000   CardType
2 ---   A   N B 0.3219281   CardType
3   A   B     N 0.9709506   CardType
4 --- A B     N 0.9709506   CardType
5 --- A N     B 0.3219281   CardType

Combinations for WheelAmount
  V1 Gain  classifier
1  2    0 WheelAmount

Combinations for PassAmount
  X..... X.2.1.       Gain classifier
1    ---     2 1 0.00000000 PassAmount
2      1       2 0.01997309 PassAmount

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

Step 3:
List of best candidates (1 for each variable):
   Sons   Gain       Classifier
X1 list,2 0.9709506  "CardType"
X2 list,1 0          "WheelAmount"
X3 list,2 0.01997309 "PassAmount"

The division with the most information gain is chosen:
    - Classifier = CardType
    - Information gain = 0.971
    - Sons =
   X2    X3
"A B"   "N"

Steps 1 and 4:
Data is classified.

Steps 1 and 4:
Data is classified.

Steps 1 and 4:
Data is classified.

Steps 1 and 4:
Data is classified.
```

Figure 3.37: Decision Tree Results_6

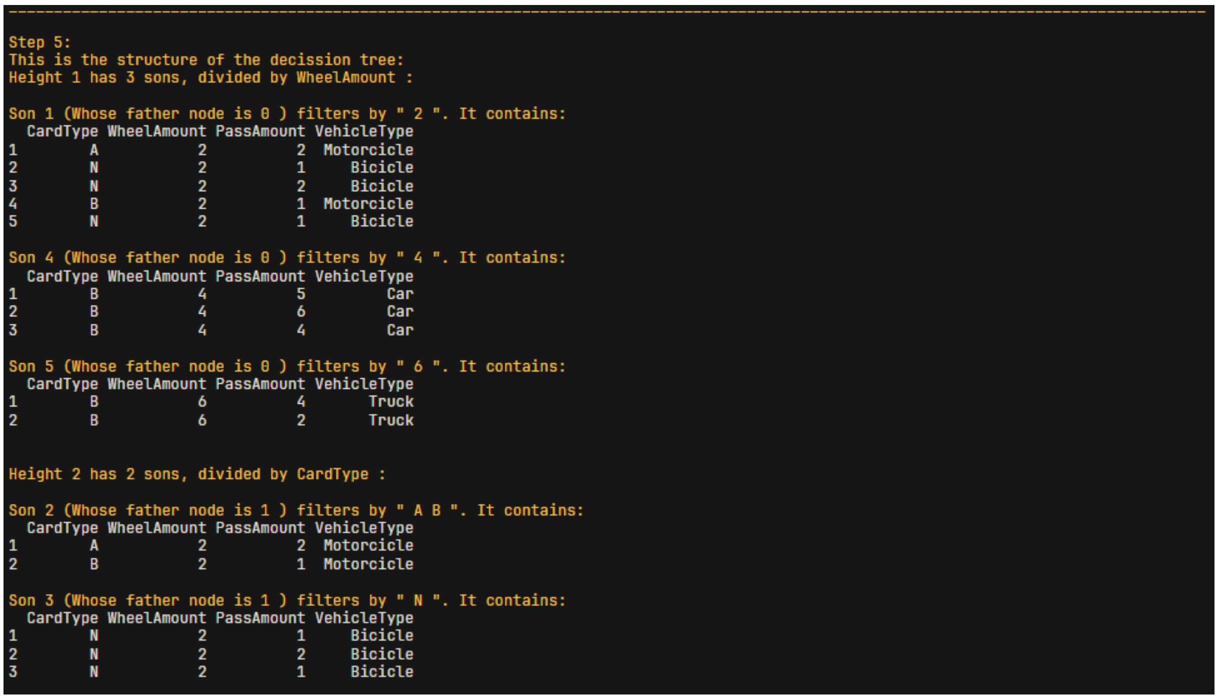

```
------------------------------------------------------------------------------------------------------------------------------------

Step 5:
This is the structure of the decission tree:
Height 1 has 3 sons, divided by WheelAmount :

Son 1 (Whose father node is 0 ) filters by " 2 ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        A           2          2  Motorcicle
2        N           2          1     Bicicle
3        N           2          2     Bicicle
4        B           2          1  Motorcicle
5        N           2          1     Bicicle

Son 4 (Whose father node is 0 ) filters by " 4 ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        B           4          5         Car
2        B           4          6         Car
3        B           4          4         Car

Son 5 (Whose father node is 0 ) filters by " 6 ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        B           6          4       Truck
2        B           6          2       Truck


Height 2 has 2 sons, divided by CardType :

Son 2 (Whose father node is 1 ) filters by " A B ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        A           2          2  Motorcicle
2        B           2          1  Motorcicle

Son 3 (Whose father node is 1 ) filters by " N ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        N           2          1     Bicicle
2        N           2          2     Bicicle
3        N           2          1     Bicicle
```

Figure 3.38: Decision Tree Results_7

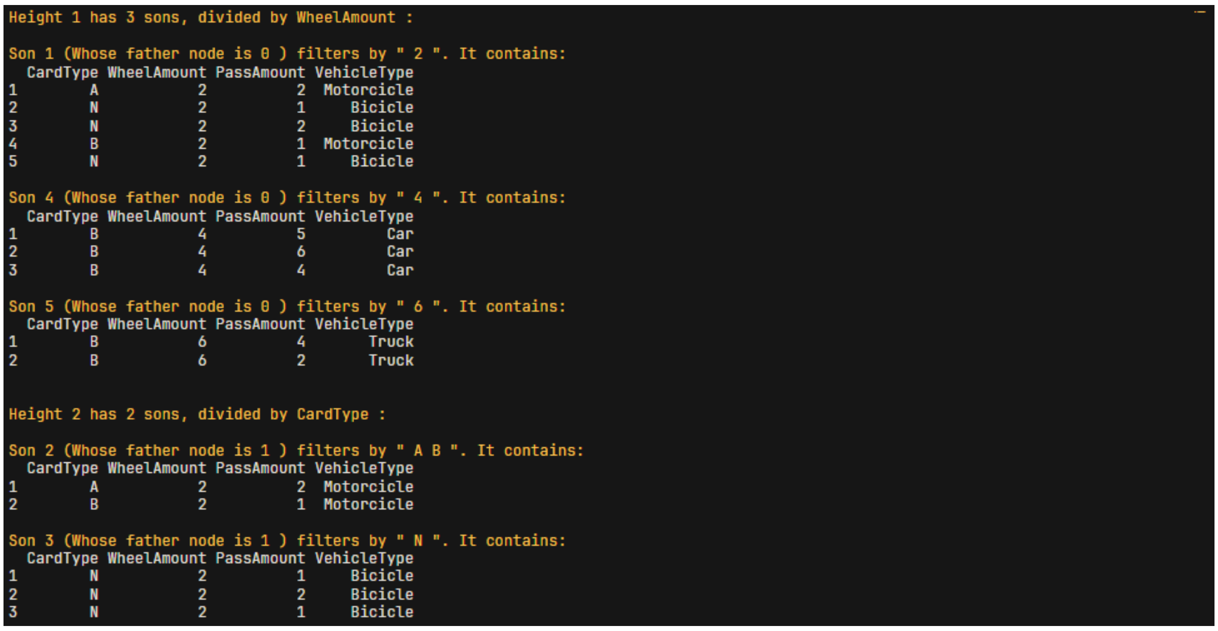

```
Height 1 has 3 sons, divided by WheelAmount :

Son 1 (Whose father node is 0 ) filters by " 2 ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        A           2          2  Motorcicle
2        N           2          1     Bicicle
3        N           2          2     Bicicle
4        B           2          1  Motorcicle
5        N           2          1     Bicicle

Son 4 (Whose father node is 0 ) filters by " 4 ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        B           4          5         Car
2        B           4          6         Car
3        B           4          4         Car

Son 5 (Whose father node is 0 ) filters by " 6 ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        B           6          4       Truck
2        B           6          2       Truck


Height 2 has 2 sons, divided by CardType :

Son 2 (Whose father node is 1 ) filters by " A B ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        A           2          2  Motorcicle
2        B           2          1  Motorcicle

Son 3 (Whose father node is 1 ) filters by " N ". It contains:
  CardType WheelAmount PassAmount VehicleType
1        N           2          1     Bicicle
2        N           2          2     Bicicle
3        N           2          1     Bicicle
```

Figure 3.39: Decision Tree Results_8

3.6 Complementary functions

The following is an explanation of the ‘console.log‘ function in R, which is used to format and display text in a manner resembling how it might be shown in a console or terminal. This is done so that the printed messages are correctly shown in any terminal, independant of its width.

```r
console.log <- function(txt, ...) {

  width <- console_width()
  #Split the input text into lines based on newline and carriage return characters
  tmp <- strsplit(txt, '[\r\n]')[[1]]

  #Vector to store processed lines
  lines <- NULL

  for (line in tmp) {
    #Leading whitespaces of the line
    white <- get_whitespace(line)

    #Remove trailing whitespace from the line
    line <- substring(line, nchar(white) + 1)
    if (is.na(line)) line <- ''

    # Expand tab characters in the leading whitespaces
    white <- gsub('\t', '  ', white)

    #Split the line into segments that fit within the console width
    parts <- strsplit(
      line,
      paste0("(?<=.{", max(width - nchar(white), 1), "})"),
      perl = TRUE
    )[[1]]

    #Add leading whitespace to each part
    parts <- paste0(white, parts)

    #Add processed parts to the list of lines
    lines <- c(lines, parts)
  }

  if (length(lines) < 1) lines <- ""

  for (line in lines) {
    message(line, ...)
  }
}

get_whitespace <- function(txt) {
  # Find whitespace at the beggining of the string
  fst_match <- gregexpr("^[ \t]*", txt)[[1]]

  # Extract whitespace
  white.length <- attr(fst_match, "match.length")
  substring(txt, 1, white.length)
}
```

The ‘console.log‘ function in R performs the following steps:

1. It calculates the width of the console or terminal.
2. The input text is split into lines based on newline and carriage return characters.
3. It iterates through each line of text.
4. Leading whitespaces at the beginning of each line are obtained.
5. Trailing whitespace is removed from each line, and if the line becomes empty, it is set to an empty string.
6. Tab characters in leading whitespaces are expanded to two spaces for consistent indentation.
7. Each line is split into segments that fit within the console width.
8. Leading whitespaces are added to each part.
9. Processed parts are added to the list of lines.
10. It ensures there is at least one line to print.
11. Finally, it prints all processed lines using the ‘message()‘ function.

This function is useful for enhancing the readability of console output, especially for long lines of text or when outputting text to a narrow console or terminal.

This last function has been developed for output clarity when printing messages through the terminal. It simply prints a line of the length of the terminals width.

```
hline <- function() {
  console.log(strrep('_', cli::console_width()))
}
```

This functions are used in every function implemented in the package.

3.7 Showcase dataframes

In addition to the functions that we’ve previously showcased, many of the dataframes used in these functions, as well as several others, have been thoughtfully included in the package. This deliberate inclusion serves the dual purpose of showcasing both the strengths and weaknesses of the implemented algorithms. Here’s how this comprehensive approach contributes to the package’s overall utility:

1. **Transparency**: By including the dataframes used in the functions, the package promotes transparency in the way it handles data. Users can access and examine the raw data, gaining insights into the input data’s quality and characteristics. This transparency allows users to make informed decisions and understand how algorithms operate.

2. **Educational Value**: These included dataframes serve as valuable educational resources. They provide tangible examples of the types of data the package is designed to handle, making it easier for users to grasp the practical application of the algorithms. Novice users can learn from these examples and become more proficient in data analysis.

3. **Testing and Validation**: The presence of these datasets simplifies testing and validation efforts. Developers and users can use these datasets to verify the correctness of the implemented algorithms. This is especially important in ensuring that the package functions as expected and produces reliable results.

4. **Diversity of Data**: Including a variety of datasets with different characteristics (e.g., size, complexity, structure) allows users to assess the algorithms' versatility. They can determine which algorithms excel under specific conditions and which may have limitations.

5. **User Engagement**: Encouraging users to work with these dataframes promotes user engagement and community participation. Users can provide feedback, suggest improvements, or share their experiences with different datasets, contributing to the package's continuous improvement.

By integrating these dataframes into the package, it becomes a more holistic and informative tool for users. It not only provides solutions but also empowers users to explore, experiment, and make informed decisions. This approach fosters a deeper understanding of the package's capabilities and limitations, ultimately enhancing its utility and user satisfaction.

3.8 How to create and upload an R package

Creating an R package using RStudio is a structured process that involves several steps, from setting up your development environment to documenting your package. In this guide, we'll walk through the entire process, emphasizing how RStudio and the `roxygen2` package can be used effectively. We will also explain how this package has been uploaded to CRAN.

3.8.1 RStudio

RStudio is a powerful integrated development environment (IDE) specifically designed for R programming. It provides numerous features and tools that make R package development more efficient and organized. Here's why RStudio has been the preferred choice for the development of this package:

- **Interactive Environment:** RStudio offers an interactive and user-friendly environment that includes a console, script editor, and various panels for viewing objects, plots, and more. This interactive interface simplifies the process of writing, testing, and debugging R code.

- **Project Management:** It allows the creation R projects, which are directories that contain all the files and settings related to your work. When developing an R package, creating a project is especially beneficial because it keeps the package files organized, making it easier to manage.

- **Code Debugging:** The IDE has built-in tools for debugging R code. You can set breakpoints, step through code, and examine variables, making it easier to identify and fix issues.

- **Version Control Integration:** RStudio integrates with version control systems like Git, allowing you to track changes, collaborate with others, and maintain version history of your package.

- **Package Development Tools:** RStudio has tools like devtools and roxygen2 (which we'll cover shortly) integrated into its environment, making package development more streamlined.

3.8.2 Roxygen2

Roxygen2 is a package in the R programming language that is used for documenting your R code. Documentation is an essential part of software development, as it helps developers and users understand how functions and packages work, what inputs they expect, and what outputs they produce. Good documentation also facilitates collaboration and makes code easier to maintain.

1. **Comments with Roxygen Tags**: To use Roxygen2, specially formatted comments need to be include, called "Roxygen tags," within the R script files. These comments begin with a #' (hash followed by a single quote), followed by a Roxygen tag and its associated documentation.

   Here's an example of a simple Roxygen2 comment:

```r
#' This is a simple Roxygen2 comment

```

2. **Documenting Functions**: Roxygen2 is often used to document functions within R packages. To document a function, Roxygen tags are included before the function definition, like so:

```r
#' Title: My Function
#' Description: This function does something useful.
#' @param x A numeric vector.
#' @return The result of the operation.
#' @examples
#' my_function(1:5)
my_function <- function(x) {
  # function code here
}

```

   In this example, the Roxygen tags provide information about the function's title, description, input parameters, return value, and examples of usage.

3. **Generating Documentation**: Once tags have been added Roxygen to the code, the roxygen2 package can be used to automatically generate documentation files. To do this, the roxygen2::roxygenize() function is run in the R project directory. This function processes the code files, extracts the documentation, and generates documentation files in the appropriate format.

4. **Building and Installing the Package**: When developing an R package, tools like devtools or R CMD build can be used to build and install the package. Roxygen2-generated documentation will be included in the package.

Here are some of the commonly used Roxygen2 tags:

- #' @title: Provides the title of the function or topic.
- #' @description: Offers a brief description of the function or topic.
- #' @param: Documents input parameters and their descriptions.
- #' @return: Documents the return value of a function.

- `#' @examples`: Provides examples of how to use the function.
- `#' @seealso`: Links to related functions or topics.
- `#' @export`: Specifies which functions should be exported from the package.

Roxygen2 offers several advantages for R developers. Firstly, it promotes consistency in documentation by enforcing a standardized format, enhancing code comprehension and maintainability. Secondly, it automates the documentation generation process, reducing the risk of outdated or incomplete documentation. Thirdly, its seamless integration with RStudio provides a user-friendly experience, including features like code completion for Roxygen tags. For package developers, Roxygen2 is particularly valuable as it simplifies the creation of package documentation that adheres to CRAN standards. Lastly, it supports documentation in multiple formats, ensuring accessibility to a broad audience and enhancing the overall usability and clarity of R code and packages.

3.8.3 Devtools

`devtools` is an R package developed by Hadley Wickham, which simplifies various aspects of package development, including building, documenting, testing, and sharing packages. Here's why devtools is an essential tool for R package development:

- **Package Creation:** `devtools` provides functions to create the basic structure of an R package, including the necessary files and directories. You can start with a template by running `devtools::create("my_package")`. The key directories include:
  - `"R"`: This directory contains the R code that defines your functions. Each function should be in its own file with a ".R" extension.
  - `"man"`: This directory contains documentation files for your package functions. Each function should have its own file with a ".Rd" extension.
  - `"data"`: This directory contains any data files that your package uses. Data files should be in a format that can be read by R, like CSV or RDS.
  - `"tests"`: This directory contains test files for your package functions. Tests should be written in the "testthat" package format, which allows you to test your functions automatically.
  - `"inst"`: This directory contains installation files for your package, like a README or LICENSE file.
- **Package Building:** Building an R package involves compiling your code and generating the necessary package metadata. `devtools::build()` automates this process, creating a distributable package (.tar.gz file) in the ./pkg directory.
- **Package Installation:** During development, Frequently the package will need to be installed to test it. `devtools::install()` simplifies the installation process, ensuring that we are always working with the latest version of your package.
- **Code Testing with `testthat`:** `devtools` integrates with the `testthat` package, allowing us to write and run tests for your functions. Test files can be placed in the tests/testthat directory and use RStudio's testing tools for seamless testing.

- **Documentation with `roxygen2`:** `devtools` also works seamlessly with `roxygen2` for documentation generation. This simplifies the process of documenting your functions using roxygen2-style comments, as we discussed in the previous section.

By leveraging RStudio's IDE, `roxygen2` package and `devtools` package, we have a robust environment that simplifies the entire R package development process. This combination of tools streamlines coding, documentation, testing, and packaging, making it easier to create high-quality R package that are well-documented, tested, and ready for sharing or publication. All this tools have been used in the creation of the package.

The package we developed includes an informative README file in the package's root directory. This file provides an overview of the package, installation instructions, usage examples, and links to further documentation.

Detailed documentation for each function in the man/ directory has been created. These .Rd files are generated from the roxygen2-style comments of the implemented functions.

3.8.4 Upload to CRAN

Once the package has been developed and thoroughly tested, it's time to share it with others. Sharing the package makes it accessible to a broader audience and potentially allows for collaboration and feedback. Here we will explore various ways to share our R package effectively.

3.8.4.1 Package versioning

Before sharing the package, it's crucial to assign a version number to it. Versioning helps users understand changes, updates, and compatibility between different versions of the package. Semantic versioning (e.g., MAJOR.MINOR.PATCH) is commonly used in R package development:

- **Major:** Increment when incompatible changes to the package API are made.
- **Minor:** Increment when new features or functionality are added in a backward-compatible manner.
- **Patch:** Increment when backward-compatible bug fixes or minor improvements are made.

The `DESCRIPTION` file must be edited to include the version information.

```
Package: mypackage
Version: 1.0.0
```

3.8.4.2 GitHub

GitHub is a widely used platform for open-source development. It allows the creation of a repository for the package, allows us to share the code, collaborate with others, and track issues and enhancements. To share the package on GitHub we have created a repository (https://github.com/ComiSeng/LearnSL) . Here we have uploaded all the code and documentation of the package. Engaging with the package's user community can be highly beneficial. In order to encourage users to provide feedback, report issues, and contribute to the package's development, an issue tracker has been created to track and respond to user feedback and bug reports promptly (https://github.com/ComiSeng/LearnSL/issues). Users can now install the developed R package (LearnSL) directly from GitHub using the `remotes` package:

```
remotes::install_github("https://github.com/ComiSeng/LearnSL.git")
```

3.8.4.3 CRAN

Once all the previous preparation has been made and the functions and implementations are complete. There are only a few missing steps to officially upload the package to CRAN. We must give the package a license these are the main ones:

- **MIT License:** This is a free software license that allows users to use, copy, modify, and distribute the package's code, even for commercial purposes, as long as the copyright notice and license are included in all copies.
- **GPL License:** This is another free software license that establishes that anyone who distributes the package must do so under the same terms as the original package. This means that if someone modifies the package and distributes it, they must also make the source code of their version available under the same license.
- **Apache License:** This is a free software license that allows users to use, copy, modify, and distribute the package's code with certain restrictions. For example, if someone modifies the package, they must clearly indicate the modifications made and provide a copy of both the original license and the new license.
- **Creative Commons License:** This is a license commonly used for creative content such as music, photography, and video, but can also be used for the source code of an R package. Creative Commons licenses allow users to use and distribute the package's code, as long as the terms and conditions of the chosen license are respected.
- **BSD License:** This is a free software license that allows users to use, copy, modify, and distribute the package's code, even for commercial purposes, as long as the copyright notice and license are included in all copies. Unlike the MIT License, the BSD License includes a clause that limits the author's liability in case of damages resulting from the use of the package.
- **LGPL License:** This is a free software license similar to the GPL License, but with more flexible restrictions. For example, the LGPL License allows the package's code to be used in proprietary projects, as long as credits and license information are provided. It also allows users to modify and distribute the package as they wish, as long as the changes made to the original package are compatible with the LGPL License.
- **MPL License:** This is a free software license that allows users to use, copy, modify, and distribute the package's code, even for commercial purposes, as long as the copyright notice and license are included in all copies. Additionally, the MPL License allows the package's code to be used in proprietary projects, as long as any changes made to the original package are published under the same MPL License.
- **CC0 License:** This is a public domain license that allows users to use, copy, modify, and distribute the package's code without any restrictions. With the CC0 License, the package's author waives all copyright and intellectual property rights to the package, making the code public property.

After carefully reading the terms and conditions of each license, we have decided to use the MIT License because the objective of this package, besides the obvious, is to reach as many people as possible.

To finish the preparation we have to create the file `cran-comments.md` in the package's root directory. This file contains information for CRAN maintainers, including any special notes or considerations regarding the package. There is no special note for our package.

Once your package is well-prepared, the following steps have been followed to submit it to CRAN:

1. **Check CRAN Policies:** The CRAN's policies and guidelines have been reviewed to ensure the package complies with their requirements. Special attention must be payed to aspects like package naming conventions, file structure, and documentation.

2. **Package Check:** We must run `devtools::check()` to perform a comprehensive check of the package. Address any warnings, errors, or notes provided by the check process. The package has to pass without any issue as it does.

3. **`release()` function** This function is used to oficially upload the package to CRAN. It ensures that many checks have been done. It checks the following:

   - `spell_check() function:` Checks misspelled words.
   - `check_win_devel() function:` This function works by bundling source package, and then uploading to https://win-builder.r-project.org/. Once building is complete a link to the built package is received in the email address listed in the maintainer field.
   - `check_rhub()` It runs build() on the package and then submits it to the R-hub builder at https://builder.r-hub.io. R-hub sends an email with the results to the package maintainer.
   - `Asks if NEWS.md has been updated:` This file specifies the features that have been added, deleted or modified in the package.
   - `Asks if DESCRIPTION has been updated:`
   - `Checks the email address`

After submitting your package, it goes through a review process by CRAN maintainers. This process can take up to 10 days. CRAN maintainers will evaluate the package for compliance with their guidelines and perform checks to ensure its functionality and stability.

During the review process, CRAN maintainers may request changes or clarifications. Be responsive to their feedback and address any issues promptly. This collaborative process helps ensure that your package meets the high standards set by CRAN.

For the developed package (LearnSL) CRAN maintainers requested minor changes in the code to follow R standars in a better way. Once all the requests where completed, we tried uploading the package one more time, this time it was accepted.

Now that the the package has passed CRAN's review process, it is be available to the R community through the CRAN repository (https://cran.r-project.org/web/packages/LearnSL/index.html). Users can install this package using the standard install.packages() function:

```
install.packages("LearnSL")
```

Chapter 4

Conclusions and future improvements

4.1 Conclusions

The primary goal of this project was to develop an R package that empowers users to grasp the intricacies of supervised classification techniques and algorithms through practical demonstrations and hands-on executions. Our intention was to facilitate the learning process by offering clear explanations and clarifications.

To accomplish this, we meticulously implemented the necessary codebase within the package, ensuring that users have easy access to the algorithms and associated functionalities. We also took the initiative to include illustrative example datasets, simplifying the learning curve for users and allowing them to put these algorithms into practice immediately.

The main distinctive feature of our package is the comprehensive insight it offers into the inner workings of the implemented algorithms. Users can delve deep into the algorithms, exploring their mechanisms step by step. Additionally, some algorithms within the package are equipped with visualization tools, enhancing the user's ability to comprehend and interpret their results effectively.

Once our package reached a state of completeness and thorough testing, we took the significant step of officially publishing it on both CRAN and GitHub. This dual availability ensures that any user, regardless of their preferred platform, can access and utilize our package freely. This open access not only benefits individual learners but also contributes to the broader R community's growth and knowledge-sharing.

By providing this valuable resource, we aim to empower individuals to build upon their R programming skills, fostering the creation of more intricate and innovative ideas within the R ecosystem. Our commitment is to facilitate learning and encourage the exploration of supervised classification techniques, ultimately strengthening the collective capabilities of R users.

Throughout the course of this project, we've acquired advanced knowledge and insights into the profound significance of machine learning, both as a broad concept and as individual algorithms. We've come to appreciate the intrinsic value of each algorithm and the pivotal role they play in solving a wide array problems. In particular, our focus has been on supervised classification algorithms, which are exceptionally valuable in domains where the logical interpretation of outcomes holds paramount importance—effectively, this applies to the vast majority of industries.

The development of this R package represents a significant milestone in our journey to democratize knowledge and foster a deeper understanding of these crucial machine learning techniques. Our package is designed as a fundamental resource for learners and practitioners alike who seek to navigate the complex landscape of supervised classification.

What sets our package apart is its commitment to providing not just a theoretical overview but also practical implementation. Users can gain hands-on experience, honing their skills through the execution of these algorithms. We firmly believe that learning by doing is the most effective way to internalize complex concepts.

Furthermore, our package illuminates the inner workings of these algorithms, granting users the ability to dissect and scrutinize each step of the classification process. This transparency empowers learners to not only apply these techniques but also comprehend why and how they produce specific results.

In today's data-driven world, the ability to harness the predictive power of machine learning is a coveted skill. With this package, we hope to lower the barriers to entry, making these advanced techniques accessible to a broader audience. Whether you're an aspiring data scientist, a business analyst, or simply someone with a passion for understanding data. This package serves as an introduction to the field. While it equips users with a strong foundation, the realm of machine learning is vast and ever-evolving. We encourage users to continue their journey beyond this introduction, exploring specialized topics and staying updated with the latest developments in the field.

In conclusion, our project represents a commitment to knowledge-sharing and empowerment. We are excited to contribute to the growth of the data science community and to witness the innovative solutions that will emerge from those who embark on this learning journey.

4.2 Future improvements

While the current state of the package effectively accomplishes its intended goals, there lies a realm of potential enhancements and additions that can further enrich its functionality in the medium and long term. Our commitment to providing a comprehensive coverage of machine learning prompts us to consider several avenues for expansion.

One promising avenue involves the incorporation of Bayesian algorithms. Algorithms like Naive Bayes or Bayesian Network Classifiers can significantly enhance the package's breadth of offerings. These algorithms are renowned for their effectiveness in handling probabilistic models, making them invaluable tools in various machine learning applications.

Another avenue of improvement includes the inclusion of Support Vector Machine (SVM) algorithms. SVMs are well-regarded for their robustness in handling both classification and regression tasks. By integrating SVMs into our package, we can empower users with an additional set of tools for tackling diverse machine learning challenges.

In addition to algorithmic expansions, there are opportunities to refine existing components. For instance, we can introduce enhancements that allow decision trees to gracefully handle continuous variables. This refinement can lead to more accurate and versatile decision tree models, particularly when dealing with datasets that encompass both discrete and continuous attributes.

Moreover, we envision the inclusion of more sophisticated algorithms that have been discussed throughout the document. Complex neural networks, Random Forests, and Gradient Boosting Trees are powerful machine learning techniques that warrant a place in our package's arsenal. These algorithms excel in modeling intricate relationships within data, and by incorporating them, we empower users to tackle advanced machine learning tasks with confidence.

While our current package serves as a solid foundation for learning and applying supervised classification techniques, these proposed enhancements and inclusions aim to make it even more versatile and potent. We are committed to continually evolving and refining the package to meet the evolving needs of the data science and machine learning community.

In summary, the journey of improving and expanding our package is an ongoing endeavor. By embracing these suggestions and implementing them in the medium and long term, we can elevate the package to new heights, ensuring that it remains a valuable resource for learners, practitioners, and enthusiasts in the field of machine learning.

Bibliography

[1] E. Rahm, H. H. Do, *et al.*, “Data cleaning: Problems and current approaches”, *IEEE Data Eng. Bull.*, vol. 23, no. 4, pp. 3–13, 2000.

[2] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang, “Data cleaning: Overview and emerging challenges”, in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 2201–2206.

[3] J. Li, K. Cheng, S. Wang, *et al.*, “Feature selection: A data perspective”, *ACM computing surveys (CSUR)*, vol. 50, no. 6, pp. 1–45, 2017.

[4] A. Kusiak, “Feature transformation methods in data mining”, *IEEE Transactions on Electronics packaging manufacturing*, vol. 24, no. 3, pp. 214–221, 2001.

[5] A. Ahmad and L. Dey, “A k-mean clustering algorithm for mixed numeric and categorical data”, *Data & Knowledge Engineering*, vol. 63, no. 2, pp. 503–527, 2007.

[6] L. Van Der Maaten, E. Postma, J. Van den Herik, *et al.*, “Dimensionality reduction: A comparative”, *J Mach Learn Res*, vol. 10, no. 66-71, 2009.

[7] H. Hjalmarsson, M. Gevers, S. Gunnarsson, and O. Lequin, “Iterative feedback tuning: Theory and applications”, *IEEE control systems magazine*, vol. 18, no. 4, pp. 26–41, 1998.

[8] D. H. Wolpert *et al.*, “On overfitting avoidance as bias”, Technical Report SFI TR 92-03-5001. Santa Fe, NM: The Santa Fe Institute, Tech. Rep., 1993.

[9] D. Berrar *et al.*, *Cross-validation.* 2019.

[10] Z. Reitermanova *et al.*, “Data splitting”, in *WDS*, Matfyzpress Prague, vol. 10, 2010, pp. 31–36.

[11] S. Kotsiantis, D. Kanellopoulos, P. Pintelas, *et al.*, “Handling imbalanced datasets: A review”, *GESTS international transactions on computer science and engineering*, vol. 30, no. 1, pp. 25–36, 2006.

[12] M. W. Hauser, “Principles of oversampling a/d conversion”, *Journal of the Audio Engineering Society*, vol. 39, no. 1/2, pp. 3–26, 1991.

[13] R. Mohammed, J. Rawashdeh, and M. Abdullah, “Machine learning with oversampling and undersampling techniques: Overview study and experimental results”, in *2020 11th international conference on information and communication systems (ICICS)*, IEEE, 2020, pp. 243–248.

[14] R. Bardenet, M. Brendel, B. Kégl, and M. Sebag, “Collaborative hyperparameter tuning”, in *International conference on machine learning*, PMLR, 2013, pp. 199–207.

[15] M. Belkin, D. Hsu, S. Ma, and S. Mandal, “Reconciling modern machine-learning practice and the classical bias–variance trade-off”, *Proceedings of the National Academy of Sciences*, vol. 116, no. 32, pp. 15 849–15 854, 2019.

[16] C. Kingsford and S. L. Salzberg, “What are decision trees?”, *Nature biotechnology*, vol. 26, no. 9, pp. 1011–1013, 2008.

[17] Y. Xia, C. Liu, Y. Li, and N. Liu, “A boosted decision tree approach using bayesian hyper-parameter optimization for credit scoring”, *Expert systems with applications*, vol. 78, pp. 225–241, 2017.

[18] J. Mingers, “An empirical comparison of pruning methods for decision tree induction”, *Machine learning*, vol. 4, pp. 227–243, 1989.

[19] M. Mehta, J. Rissanen, R. Agrawal, *et al.*, “Mdl-based decision tree pruning.”, in *KDD*, vol. 21, 1995, pp. 216–221.

[20] W. N. H. W. Mohamed, M. N. M. Salleh, and A. H. Omar, “A comparative study of reduced error pruning method in decision tree algorithms”, in *2012 IEEE International conference on control system, computing and engineering*, IEEE, 2012, pp. 392–397.

[21] L. Breiman, “Random forests”, *Machine learning*, vol. 45, pp. 5–32, 2001.

[22] L. Breiman, “Bagging predictors”, *Machine learning*, vol. 24, pp. 123–140, 1996.

[23] R. E. Schapire, Y. Singer, and A. Singhal, “Boosting and rocchio applied to text filtering”, in *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, 1998, pp. 215–223.

[24] Y. Freund and R. E. Schapire, “A desicion-theoretic generalization of on-line learning and an application to boosting”, in *European conference on computational learning theory*, Springer, 1995, pp. 23–37.

[25] J. H. Friedman, “Greedy function approximation: A gradient boosting machine”, *Annals of statistics*, pp. 1189–1232, 2001.

[26] M. Chester, *Neural networks: a tutorial.* Prentice-Hall, Inc., 1993.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors”, *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[28] Q. Wang, Y. Ma, K. Zhao, and Y. Tian, “A comprehensive survey of loss functions in machine learning”, *Annals of Data Science*, pp. 1–26, 2020.

[29] S. Ruder, “An overview of gradient descent optimization algorithms”, *arXiv preprint arXiv:1609.04747*, 2016.

[30] S. Sharma, S. Sharma, and A. Athaiya, “Activation functions in neural networks”, *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[31] K. O’Shea and R. Nash, “An introduction to convolutional neural networks”, *arXiv preprint arXiv:1511.08458*, 2015.

[32] Y. LeCun, Y. Bengio, *et al.*, “Convolutional networks for images, speech, and time series”, *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[33] L. R. Medsker and L. Jain, “Recurrent neural networks”, *Design and Applications*, vol. 5, no. 64-67, p. 2, 2001.

[34] S. Hochreiter and J. Schmidhuber, “Long short-term memory”, *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[35] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, “Generative adversarial nets”, *Advances in neural information processing systems*, vol. 27, 2014.

[36] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, and L. Bottou, “Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion.”, *Journal of machine learning research*, vol. 11, no. 12, 2010.

[37] D. P. Kingma and M. Welling, “Auto-encoding variational bayes”, *arXiv preprint arXiv:1312.6114*, 2013.

[38] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need”, *Advances in neural information processing systems*, vol. 30, 2017.

[39] T. Cover and P. Hart, “Nearest neighbor pattern classification”, *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[40] F. Rosenblatt, *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[41] S. B. Kotsiantis, “Decision trees: A recent overview”, *Artificial Intelligence Review*, vol. 39, pp. 261–283, 2013.

[42] L. Rokach and O. Maimon, “Decision trees”, *Data mining and knowledge discovery handbook*, pp. 165–192, 2005.

[43] J. R. Quinlan, “Induction of decision trees”, *Machine learning*, vol. 1, pp. 81–106, 1986.

[44] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[45] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[46] S. Singh and P. Gupta, “Comparative study id3, cart and c4. 5 decision tree algorithm: A survey”, *International Journal of Advanced Information Science and Technology (IJAIST)*, vol. 27, no. 27, pp. 97–103, 2014.

[47] H. Buhrman and R. De Wolf, “Complexity measures and decision tree complexity: A survey”, *Theoretical Computer Science*, vol. 288, no. 1, pp. 21–43, 2002.

[48] M. J. Moshkov, “Time complexity of decision trees”, in *Transactions on Rough Sets III*, Springer, 2005, pp. 244–459.

[49] M. Bramer, “Avoiding overfitting of decision trees”, *Principles of data mining*, pp. 119–134, 2007.

[50] F. Esposito, D. Malerba, G. Semeraro, and J. Kay, “A comparative analysis of methods for pruning decision trees”, *IEEE transactions on pattern analysis and machine intelligence*, vol. 19, no. 5, pp. 476–491, 1997.

[51] H. Kim and W.-Y. Loh, “Classification trees with unbiased multiway splits”, *Journal of the American Statistical Association*, vol. 96, no. 454, pp. 589–604, 2001.

[52] R. G. Mantovani, T. Horváth, R. Cerri, S. B. Junior, J. Vanschoren, and A. C. P. d. L. F. de Carvalho, “An empirical study on hyperparameter tuning of decision trees”, *arXiv preprint arXiv:1812.02207*, 2018.