Machine Learning Methods

Statistical Learning Methods

Keenan Anderson-Fears

Chapter 1

Introduction

Machine learning or statistical learning encompasses a wide range of tools employed for examining, interpreting, and comprehending data. These tools can be applied from both supervised and unsupervised perspectives. Supervised learning utilizes labeled data, where each data point has a corresponding output or target variable. Unsupervised learning, on the other hand, deals with unlabeled data, where the goal is to identify patterns and relationships within the data itself. Supervised learning algorithms involve classification and regression, while unsupervised learning algorithms encompass association, clustering, and dimensional reduction. In supervised learning, we construct a model that maps one or more inputs to an output, enabling predictions based on the model's understanding of the data. In unsupervised learning, there are no predetermined outputs, and the focus lies in uncovering the underlying structure and relationships within the data.

While there are a great many statistical tests that are available to researchers who use machine learning methods, there are also exist a wealth of methods for those who do not. In this section I want to highlight two main methods, the T-test and χ^2 - test, as these are two of the most commonly used statistical tests outside of machine learning and many methods we would classify as machine learning often include or are made to use the test statistics from said methods. The following table gives a breakdown for the use of classical statistical methods vs machine learning methods, i.e. when to use based on specific features of the data, problem or goal of the researcher.

Classical Statistics vs Machine Learning		
	Statistical Methods	Machine Learning Methods
Data	Small to Large	Large and complex
Large/Complex	Simple or Complex	Non-linear
Goal	Test Hypothesis or Explain Results	Make Predictions
Explainability	Transparent	Can be a Black Box
Generalizability	Good for Data they were Trained On	Can Generalize to New Data

Warning! Each chapter contains R code for the coding of each individual machine learning/statistical concept. But this code is not built to just be copied and pasted, the functionality of it has not been assessed. These are meant to provide a window past the pseudo-code traditionally found in higher level texts, while using little to no outside libraries. Do your best to read and understand the concepts and what the code is trying to achieve, and then either re-create it for your own purposes or use a pre-made function. While code is provided for R, this can be easily translated to Python or any other language. Good Luck!

Chapter 2

Before Machine Learning

2.1 T - test

The T-test is a fundamental statistical tests which is used to compare the means, averages, of two groups in order to assess whether the observed differences are statistically significant or not. The basic logic behind the test is that we are operating under the null hypothesis that the means of two populations are in fact equal. Then a t-statistic is calculated which captures how many standard deviations the observed difference between sample means falls from the expected difference under the previously described null hypothesis. Therefore, a larger t-statistic indicates a more significant deviation, thereby indicating the null hypothesis is less likely. The formula is calculated via the following:

$$t = \frac{(\hat{X}_1 - \hat{X}_2)}{s_p * \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

Here $\hat{X}_1$ and $\hat{X}_2$ are the sample means of the two groups and s_p is the pooled standard deviation:

$$s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 2)s_2^2}{(n_1 + n_2 - 2)}}$$

Where n_1 and n_2 are the sample sizes of the two groups, s_1 and s_2 are the sample standard deviations of the two groups and the degrees of freedom are represented by the sum total of the sample sizes minus 2. our p-value is the probability of observing either the calculated t-statistic or a more extreme value under the assumption that the null hypothesis is true, with smaller p-values indicating less support for the null.

The overall process of performing the t-test can be described in the following sequence:

1. Define the null and alternative hypotheses
2. Gather your data into two separate groups representing the two populations you want to compare
3. For each group, calculate the mean, standard deviation, and sample size
4. Compute the pooled standard deviation
5. Calculate the t-statistic

6. Determine the degrees of freedom
7. Find the p-value using a t-distribution table or a statistical software
8. Compare the p-value to a chosen significance level, if the p-value is less than the significance level, reject the null hypothesis

```r
t_test <- function(x1, x2) {
  # Calculate mean and standard deviation
  mean1 <- mean(x1)
  mean2 <- mean(x2)
  sd1 <- sd(x1)
  sd2 <- sd(x2)

  # Calculate pooled standard deviation
  n1 <- length(x1)
  n2 <- length(x2)
  pooled_sd <- sqrt(((n1 - 1) * sd1^2 + (n2 - 1) * sd2^2) / (
   n1 + n2 - 2))

  # Calculate t-statistic
  t_stat <- (mean1 - mean2) / (pooled_sd * sqrt(1/n1 + 1/n2))

  # Calculate degrees of freedom
  df <- n1 + n2 - 2

  # Calculate p-value using t distribution function
  pt <- 2 * tdist(abs(t_stat), df)

  list(t_statistic = t_stat, p_value = pt)
}

x1 <- rnorm(10, mean = 5)
x2 <- rnorm(15, mean = 7)

results <- t_test(x1, x2)

print(paste("t-statistic:", results$t_statistic))
```

Listing 2.1: T-Test Function

2.2 χ^2 - TEST

While the t-test provides a direct comparison between the summary statistics, specifically the mean, of two groups, it is not able to handle categorical data. Comparatively the chi-squared test is used to assess the association between two categorical variables operating under the null hypothesis that there is no association between the two. It compares the observed frequencies of each category combination with the expected frequencies, assuming the null hypothesis is true. Here the chi-squared statistic measures the difference between these observed and expected frequencies with a larger statistic indicating a more significant deviation from the null hypothesis, suggesting a possible association.

We can compute our test statistic from the following:

$$\chi^2 = \frac{\sum(Observed - Expected)^2}{Expected}$$

Here, $\sum$ represents the sum across all categories with the observed being the frequency in our data for each category combination and the expected being the frequency for each category combination under the null hypothesis(assuming no association). This expected frequency for each category combination is calculated by multiplying the marginal totals of the corresponding row and column, then dividing by the total sample size.

Our degrees of freedom can be calculated via the following:

$$df = (r - 1) * (c - 1)$$

where r and c are the numbers of rows and columns respectively.

The overall process of performing the t-test can be described in the following sequence:

1. Define the null and alternative hypotheses
2. Create a contingency table representing the frequencies of each category combination for both variables
3. For each category combination, calculate the expected frequency
4. Compute the chi-squared statistic based on the observed and expected frequencies
5. Determine the degrees of freedom based on the dimensions of the contingency table
6. Find the p-value using a chi-squared distribution table or a statistical software
7. Compare the p-value to a chosen significance level and if the p-value is less than the significance level, reject the null hypothesis

```r
chi_squared_test <- function(observed_table) {
  # Calculate marginal totals
  row_totals <- rowSums(observed_table)
  col_totals <- colSums(observed_table)
  total_sample_size <- sum(observed_table)

  # Calculate expected frequencies
  expected_table <- outer(row_totals, col_totals) / total_
   sample_size

  # Calculate chi-squared statistic
  chi_squared_statistic <- sum((observed_table - expected_
   table)^2 / expected_table)

  # Calculate degrees of freedom
  df <- (nrow(observed_table) - 1) * (ncol(observed_table) -
   1)

  # Calculate p-value using chi-squared distribution function
  pt <- pchisq(chi_squared_statistic, df)

  list(chi_squared_statistic = chi_squared_statistic, p_value
     = pt)
}

observed_table <- matrix(c(50, 20, 30, 35), nrow = 2, ncol =
    2)

results <- chi_squared_test(observed_table)

print(paste("Chi-squared statistic:", results$chi_squared_
    statistic))
print(paste("p-value:", results$p_value))
```

Listing 2.2: χ^2 - Test Function

Chapter 3

Regression

3.1 Linear Regression

Linear regression stands as one of the most fundamental machine learning algorithms, finding application in a diverse array of tasks, including prediction, classification, and regression. To construct a linear regression algorithm from scratch, the following steps are necessary:

1. Determine the mean and standard deviation for each feature.
2. Standardize: Normalize the features by subtracting the mean and dividing by the standard deviation.
3. Employ the ordinary least squares (OLS) method to derive the linear regression model coefficients.
4. Utilize the linear regression model coefficients to generate predictions.
5. Evaluate the model's performance using a metric such as mean squared error (MSE).

Linear regression can be implemented in two forms: simple linear regression, utilizing a single predictor, or multiple linear regression, where a set of predictors is used to regress the outcome.

Simple Linear Regression:

$$Y = \beta_0 + \beta_1 X$$

Multiple Linear Regression:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p + \epsilon$$

Before proceeding with these models, it is crucial to perform several checks on your data:

Non-Linearity: Visualize the residuals against the predicted or fitted values. If a pattern emerges, consider applying a transformation (logarithmic, square root, square, etc.) to achieve a straight-line relationship between the predictors and the response.

Error Term Correlations: Evaluate the correlations between error terms. If errors are correlated, overall standard errors may be underestimated, leading to narrower-than-actual confidence intervals or prediction intervals.

Non-Constant Variance of Error: Examine the residuals for a funnel shape. If this pattern appears, consider transforming the response using a concave function such as the logarithm or square root.

Outliers: Identify outliers by plotting studentized residuals. Values exceeding 3 are considered outliers and may need to be removed.

High Leverage Points: Detect high leverage points, which represent observations with unusual X values. Plot studentized residuals and calculate leverage statistics using the hatvalue() function. Remove observations with leverage values greater than 2.

Multicollinearity: Assess the presence of multicollinearity, which occurs when two or more predictors are highly correlated. Visualize the correlation matrix and calculate the Variance Inflation Factor (VIF). If the VIF exceeds 5 or 10, consider dropping one of the collinear predictors or combining them into a single composite predictor.

```r
# Calculate the ordinary least squares (OLS) estimates of the
    model coefficients.
ols <- function(features, target_variable) {
  # Calculate the covariance matrix of the features.
  covariance_matrix <- cov(features)

  # Calculate the inverse of the covariance matrix.
  inverse_covariance_matrix <- solve(covariance_matrix)

  # Calculate the cross-product matrix of the features and
   the target variable.
  cross_product_matrix <- features %*% target_variable

  # Calculate the model coefficients.
  model_coefficients <- inverse_covariance_matrix %*% cross_
   product_matrix

  # Return the model coefficients.
  return(model_coefficients)
}

# Calculate the residuals.
residuals <- function(features, target_variable, model_
    coefficients) {
  # Calculate the predicted values.
  predictions <- features * model_coefficients

  # Calculate the residuals.
  residuals <- target_variable - predictions

  # Return the residuals.
  return(residuals)
}

# Calculate the standard errors of the model coefficients.
standard_errors <- function(features, target_variable, model_coefficients) {
  # Calculate the covariance matrix of the residuals.
  residuals_covariance_matrix <- cov(residuals(features, target_variable, model_coefficients))

  # Calculate the diagonal elements of the residuals covariance matrix.
  standard_errors <- sqrt(diag(residuals_covariance_matrix))

  # Return the standard errors.
  return(standard_errors)
}

# Calculate the p-values of the model coefficients.
p_values <- function(standard_errors) {
  # Calculate the p-values.
  p_values <- 2 * (1 - pnorm(abs(standard_errors)))

  # Return the p-values.
  return(p_values)
}

# Create the lm() function.
lm <- function(features, target_variable) {
  # Calculate the model coefficients.
  model_coefficients <- ols(features, target_variable)

  # Calculate the standard errors of the model coefficients.
  standard_errors <- standard_errors(features, target_variable, model_coefficients)

  # Calculate the p-values of the model coefficients.
  p_values <- p_values(standard_errors)

  # Return the model coefficients, standard errors, and p-values.
  return(list(model_coefficients = model_coefficients, standard_errors = standard_errors, p_values = p_values))
}

# Load the data
data <- read.csv("data.csv")

```

```r
# Extract the features and target variable
features <- data[, -1]
target_variable <- data[, 1]

# Standardize the features
standardized_features <- (features - mean(features)) / sd(
    features)

# Calculate the ordinary least squares (OLS) estimates of the
     model coefficients
model_coefficients <- ols(standardized_features, target_
    variable)

# Calculate the standard errors of the model coefficients
standard_errors <- sd(residuals(standardized_features, model_
    coefficients))

# Calculate the p-values of the model coefficients
p_values <- 2 * (1 - pnorm(abs(model_coefficients / standard_
    errors)))

# Print the results
print(model_coefficients)
print(standard_errors)
print(p_values)
```

Listing 3.1: Linear Regression Function

3.2 Logistic Regression

Logistic regression, another fundamental machine learning algorithm, finds application in a diverse range of tasks, including prediction, classification, and regression. Constructing a logistic regression algorithm from scratch involves the following steps:

1. Encode the Target Variable: Convert the target variable into a factor variable to accommodate categorical data.
2. Calculate Logistic Regression Coefficients: Employ the gradient descent method to determine the coefficients of the logistic regression model.
3. Generate Predictions: Utilize the logistic regression model coefficients to make predictions.
4. Evaluate Model Performance: Assess the model's effectiveness using a suitable metric such as accuracy.

Unlike linear regression, logistic regression employs a logit transformation of the outcome variable to produce a probability between 0 and 1 for a single binary outcome.

Logistic Regression:

$$Pr(Y = 1|X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p + \epsilon$$

To determine the model coefficients, we employ a likelihood function and select values that maximize this function. This approach is known as maximum likelihood (MLE).

$$l(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_{i'}))$$

For guidance on evaluating model goodness of fit or detecting potential errors in logistic regression, please refer to the linear regression section.

```r
# Calculate the gradient of the negative log-likelihood function.
gradient <- function(features, target_variable, model_coefficients, family) {
  # Calculate the predicted probabilities.
  predictions <- predict(features, model_coefficients, family)

  # Calculate the gradient.
  gradient <- (target_variable - predictions) %*% features

  # Return the gradient.
  return(gradient)
}

# Update the model coefficients using gradient descent.
update_coefficients <- function(model_coefficients, gradient, learning_rate) {
  # Update the model coefficients.
  model_coefficients <- model_coefficients - learning_rate * gradient

  # Return the updated model coefficients.
  return(model_coefficients)
}

# Create the glm() function.
glm <- function(features, target_variable, family = "binomial") {
  # Initialize the model coefficients.
  model_coefficients <- rep(0, ncol(features))

  # Iterate over the number of boosting iterations.
  while (TRUE) {
    # Calculate the gradient.
    gradient <- gradient(features, target_variable, model_
   coefficients, family)

    # Check if the gradient is close to zero.
    if (all(abs(gradient) < 1e-6)) {
      break
    }

    # Update the model coefficients.
    model_coefficients <- update_coefficients(model_
   coefficients, gradient, learning_rate = 0.01)
  }

  # Return the model coefficients.
  return(model_coefficients)
}

# Load the data
data <- read.csv("data.csv")

# Extract the features and target variable
features <- data[, -1]
target_variable <- data[, 1]

# Convert the target variable to a factor variable
target_variable <- factor(target_variable)

# Create a logistic regression model
model <- glm(target_variable ~ features, family = "binomial")

# Print the results
summary(model)
```

Listing 3.2: Logistic Regression Function

3.3 Shrinkage

Within the realm of regression, alternatives to the use of least squares known as regularization techniques exist. These techniques, such as Ridge Regression, Lasso, and Elastic Net, incorporate penalty terms to effectively shrink the contribution of features that do not significantly improve the model's performance, thereby reducing overfitting and balancing the bias-variance tradeoff. The traditional method for evaluating model performance utilizes the residual sum of squares (RSS) to minimize the following:

$$RSS = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2$$

This measure quantifies the deviation of our estimated values, $\hat{y}$, from the actual observed outcomes, y. However for the ridge regression we use the following:

$$RSS = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

The tuning parameter λ, which is greater than or equal to zero, controls the shrinkage of the model coefficients towards zero. As λ increases, the effect of shrinkage grows, reducing the model's flexibility. This leads to a decrease in variance but an increase in bias. This phenomenon is known as l_2 regularization or Euclidean distance regularization.

However, ridge regression has its limitations. While it shrinks coefficients towards zero, it cannot eliminate them entirely. This limitation is particularly problematic in datasets with a large number of predictors ($p > n$). For this we use the lasso, given by:

$$RSS = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Effectively, the lasso performs similarly to ridge regression, but with a crucial difference: it can force certain parameters to zero when the regularization parameter, λ, is sufficiently large. This property enables variable selection, allowing us to identify the variables that have a significant impact on the outcome variable. Variable selection can be visualized using a lasso path, and it is a valuable tool for selecting the most influential variables.

Finally, elastic net regression aims to address the limitations of lasso regression. While lasso tends to eliminate correlated predictors and is restricted to selecting a maximum of n predictors when $p > n$, elastic net regression overcomes these drawbacks by combining the L_1 and L_2 penalties into a single model. This combination allows elastic net to select a wider range of predictors while retaining the variable selection capabilities of the lasso:

$$RSS = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda\alpha \sum_{j=1}^{p} \beta_j^2 + \lambda(1-\alpha) \sum_{j=1}^{p} |\beta_j|$$

In elastic net regression, the parameter α functions as as a percentage indicator for the relative contribution of the L_1 and L_2 penalties. When α is 0, elastic net reduces to lasso regression, and when α is 1, it becomes ridge regression. Regardless of the chosen regularization technique, **cross-validation** is employed to determine the optimal tuning parameters. In cross-validation, the data is divided into training and testing sets. The model is fitted on the training set, and its performance is evaluated on the testing set using the mean squared error (MSE) as the evaluation metric. Various cross-validation methods exist, including leave-one-out cross-validation and k-fold cross-validation. Below is the CV estimate for k-fold:

$$CV_k = \frac{1}{k} \sum_{i=1}^{k} MSE_i$$

In k-fold cross-validation, the data is randomly divided into k equal-sized subsets or folds. The model is trained on $k-1$ folds, and its performance is evaluated on the remaining fold. This process is repeated for each fold, and the average mean squared error (MSE) across all folds is calculated to provide a cross-validation estimate.

Bootstrapping is another method we can employ like k-fold cross validation, in which sampling with replacement is performed with the primary assumption being that the estimation or inference of the population can be performed by simply re-sampling the sample population.

3.4 Data Types

Before delving into the specific methods, it is crucial to determine the type of data we are working with. Of the methods discussed, we have focused on parametric approaches, which rely on strong assumptions about the underlying data distribution. These assumptions include:

Normality: Each group should contain normally distributed populations.

Equal Variance: Each group should exhibit approximately equal variance.

Independence: Each group should consist of independent and identically distributed (iid) samples.

No Outliers: Each group should not contain any extreme outliers.

Non-parametric methods, on the other hand, provide a means to analyze and process data that violates one or more of these assumptions. One such method, k-nearest neighbors (KNN), can be used as an alternative to linear or logistic regression.

3.5 Distributions

There are two primary types of probability distributions: discrete and continuous. In discrete distributions, the variable can only assume a countable number of values, and the expectation E can be expressed as a finite sum. Conversely, continuous distributions allow the variable to take on an infinite number of values, akin to a number line.

The binomial distribution is a discrete probability distribution that models the number of successes in a sequence of independent events with binary outcomes. Each individual event is referred to as a Bernoulli trial, and a special case of this distribution is known as the Bernoulli distribution. The equation is modeled by:

$$p(x|n,p) = \left(\frac{n!}{x!(n-x)!}\right) p^x (1-p)^{n-x}$$

Here x is the number of event outcomes we are interested in, p is the probability of event outcome x and n is the total number of events.

The Poisson distribution is another discrete probability distribution that finds application in time series modeling. It specifically models the probability of a finite number of events occurring during a fixed interval in time or space, assuming that the events occur independently and have a constant average rate. The distribution is modeled by:

$$p(x|\lambda) = \left(\frac{e^{-\lambda}\lambda^x}{x!}\right)$$

Here λ is our average rate, e is Euler's number and x is the specific number of events we are interested in. Despite their effectiveness, discrete probability distributions demand substantial amounts of data to be constructed. Moreover, they lack a definitive approach to handling gaps within the distribution caused by missing data.

In contrast, continuous probability distributions circumvent this issue due to their continuous nature. Among the many types of continuous probability distributions, the most renowned is the Gaussian or normal distribution, commonly known as the bell curve. The central limit theorem asserts that under certain conditions, physical quantities arising from the summation of many independent events will exhibit Gaussian distributions. Furthermore, if normality holds, numerous statistical methods, such as least squares and maximum likelihood, can be derived analytically. This means we can express it in a mathematical form that can be directly solved or evaluated without the need for numerical approximations or simulations and that the method's underlying principles and assumptions can be clearly understood and translated into mathematical equations. There are several advantages to this:

Transparency and Understanding: Analytical derivations provide a transparent and understandable explanation of how the method works, revealing its underlying assumptions and limitations. This transparency enhances the credibility of the method and allows statisticians to make informed decisions about its applicability in various situations.

Mathematical Properties: Analytical derivations often reveal important mathematical properties of the method, such as its bias, variance, and efficiency. These properties provide insights into the method's performance and help in selecting the most appropriate technique for a given problem.

Computational Efficiency: In some cases, analytical derivations lead to efficient computational algorithms, allowing for faster and more scalable solutions. This is particularly valuable when dealing with large datasets or computationally intensive problems.

Theoretical Insights: Analytical derivations can provide deeper theoretical insights into the statistical properties of the method, enabling statisticians to develop new statistical theories and techniques.

Normal distributions are defined by their peak which is the mean and their width which is a measure of standard deviation (the amount of dispersion/variation in the distribution). Approximately *95%* of all measurements fall within +/-2 standard deviations from the mean. The distribution is modeled by:

$$f(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$$

From this equation, x is the x-axis, μ is the mean of the distribution and σ is is the standard deviation. It is important to note however, that for a continous probability distribution, the output is not a probability but rather a likelihood.

3.6 Optimization: Maximizing and Minimizing

Previously, we discussed methods that involve maximizing (likelihood) or minimizing (squared error), which can be implemented computationally using the functions

$argmax()$ and $argmin()$, respectively. However, it's essential to understand the underlying concepts behind these functions. Notably, minimizing $f(x)$ is equivalent to maximizing $-f(x)$. Therefore, any algorithm designed for maximizing a function can be adapted for minimization by simply negating the objective function. The objective function represents the loss function or the function we aim to minimize or maximize.

These extrema we seek can be either local or global. In unconstrained optimization problems, this distinction is straightforward and simple to solve. Take for example the following:

$$f(x) = 5 + (x-7)^2$$

$$f'(x) = 2(x-7)$$

$$0 = 2(x-7)$$

$$0 = x-7$$

$$x = 7$$

In unconstrained optimization problems, finding the minimum or maximum is relatively straightforward. We simply set the function's derivative to zero and solve for the solution points. In our example, setting the derivative to zero leads to the solution $x = 7$, which in this case is the global minimum. This approach holds true for unconstrained problems in any dimension. We identify potential extrema by finding points where the *gradient* (multivariate derivative) is zero. However, it's crucial to differentiate these from inflection points, where the curve changes sign but the *gradient* remains zero. To confirm a true minimum or maximum, we further analyze the second derivative at the point of interest. If the second derivative is positive, the point is a minimum; if it's negative, it's a maximum. If the second derivative is zero, further investigation is needed to determine the nature of the point.

In the constrained problem we have to worry about the minima/maxima we are obtaining as they might fall out of bounds for our problem.

$$f(x) = 5 + (x-7)^2 \quad s.t. \quad x \leq 1$$

$$f'(x) = 2(x - 7)$$

$$f'(1) = 2(1 - 7)$$

$$f'(1) = -12$$

Here 1 is our minimum boundary which gives a minima of -12 as our derivative is $2(x - 7)$.

Chapter 4

Decision Tree

Decision trees are powerful tools in statistical learning for both classification and regression tasks. Their appeal lies in several key advantages: interpretability, non-parametric flexibility, and efficient handling of categorical features. These trees are built through a recursive process of data partitioning, using features that best separate the target variable (class or continuous outcome) based on specific metrics:

Impurity Measures: These quantify how "mixed" a node is, meaning the proportion of different classes or the variance within a group. Common examples include Gini impurity for classification and variance for regression.

Common Impurity Measures:

1. *Gini Index*: Assesses the likelihood of a randomly chosen sample being incorrectly classified.

   Gini Impurity:
   $$Gini_t = 1 - \sum \pi_i^2$$

   Here t is a node within the tree, π_i is the proportion of data points in node t that belong to class i.

2. *Entropy*: Quantifies the level of uncertainty or randomness within the node.

   Entropy:
   $$Entropy_t = -\sum p_i * log_2(p_i)$$

   Here t_i is the subset of t of class i and p_i is the proportion of elements in t with a specific result. This is used in classification decision tree algorithms as an alternative to Gini impurity.

3. *Misclassification Error*: Directly measures the proportion of incorrectly classified samples.

   Variance Impurity:
   $$Var_t = \sigma_t^2$$

   Here t is a node in the tree and σ_t^2 is the variance of the target variable within node t.

Splitting Rules: At each node of a decision tree, a pivotal decision governs its growth: the choice of feature and threshold to split upon. This choice meticulously aims to minimize impurity within the resulting child nodes, fostering greater homogeneity and predictive power. The algorithm navigates this decision by evaluating potential splits. It compares the impurity of the original node with the combined impurity of the two child nodes that would emerge from each split. By crowning the split that yields the most "pure" child nodes, the algorithm meticulously guides the tree towards a structure that effectively discerns between different classes or values within the data. The specific impurity measure employed often hinges on the nature of the problem and the desired model characteristics. By balancing impurity reduction with computational efficiency and overfitting prevention, decision tree algorithms construct powerful and interpretable models capable of navigating complex decision-making scenarios.

$$Gini_split = Gini_{Child_1} * \frac{N_1}{N} + Gini_{Child_2} * \frac{N_2}{N}$$

Here N is the total number of data points in the parent node and N_1/N_2 are the number of data points in the child nodes 1 and 2 respectively. The feature and threshold combination that minimizes Gini(split) is chosen for the split.

Stopping Criterion: The decision tree grows until it reaches one of three conditions: a maximum depth, a minimum [impurity measure] threshold, or no further improvement in the [impurity measure]. In many of the downloadable functions, these conditions will be customizable parameters you can set.

Information Gain: A measure of how much splitting on a particular feature reduces the overall uncertainty (entropy) in a dataset. This metric is used in algorithms like decision trees and random forests to choose the best feature to split on at each node, ultimately creating a more accurate model.

$$Gain(t, feature) : Entropy_t - Gini_{Child_1} * \frac{N_1}{N} - Gini_{Child_2} * \frac{N_2}{N}$$

Depending on the specific algorithm and implementation, additional calculations and adjustments might be involved. For instance, handling numeric features with continuous thresholds might require different approaches compared to the binary split example shown above. To build a decision tree algorithm from the ground up, you would need to implement the following steps:

1. Calculate the information gain for each feature
2. Split the data on the feature with the highest information gain
3. Recursively repeat steps 1 and 2 until the stopping criterion is met

4. Make predictions by following the decision tree from the root node to the leaf nodes

5. Evaluate the model using a metric such as accuracy

```r

# Calculate the information gain for each feature
information_gain <- function(features, target_variable) {
  # Calculate the entropy of the target variable
  entropy_target_variable <- entropy(target_variable)

  # Calculate the entropy of each feature
  entropy_features <- entropy(features)

  # Calculate the information gain for each feature
  information_gain <- entropy_target_variable - entropy_
   features

  # Return the information gain
  return(information_gain)
}

# Split the data on the feature with the highest information
   gain
split_data <- function(features, target_variable, feature_
   index) {
  # Sort the data by the feature
  sorted_data <- data.frame(features[feature_index], target_
   variable)
  sorted_data <- sorted_data[order(sorted_data[, 1]), ]

  # Split the data into two parts at the median
  left_child <- sorted_data[1:round(nrow(sorted_data) / 2), ]
  right_child <- sorted_data[(round(nrow(sorted_data) / 2) +
   1):nrow(sorted_data), ]

  # Return the two child nodes
  return(list(left_child = left_child, right_child = right_
   child))
}

# Build the decision tree
build_decision_tree <- function(features, target_variable,
   max_depth = 10) {
  # Calculate the information gain for each feature
  information_gain <- information_gain(features, target_
   variable)

  # Find the feature with the highest information gain
  best_feature_index <- which.max(information_gain)

  # If the maximum depth is reached or the information gain is zero, return a leaf node
  if (max_depth == 0 || information_gain[best_feature_index] == 0) {
    return(list(type = "leaf", target_variable = mean(target_variable)))
  }

  # Split the data on the feature with the highest information gain
  child_nodes <- split_data(features, target_variable, best_feature_index)

  # Build the decision tree recursively on the child nodes
  left_child <- build_decision_tree(child_nodes$left_child[, -best_feature_index], child_nodes$left_child$target_variable, max_depth - 1)
  right_child <- build_decision_tree(child_nodes$right_child[, -best_feature_index], child_nodes$right_child$target_variable, max_depth - 1)

  # Return the decision tree node
  return(list(type = "node", feature_index = best_feature_index, left_child = left_child, right_child = right_child))
}

# Make predictions using the decision tree
predict <- function(features, decision_tree) {
  # If the decision tree node is a leaf node, return the target variable
  if (decision_tree$type == "leaf") {
    return(decision_tree$target_variable)
  }

  # Recursively make predictions using the child nodes
  if (features[decision_tree$feature_index] <= median(features[decision_tree$feature_index])) {
    return(predict(features, decision_tree$left_child))
  } else {
    return(predict(features, decision_tree$right_child))
  }
}

# Load the data
```

```r
data <- read.csv("data.csv")

# Extract the features and target variable
features <- data[, -1]
target_variable <- data[, 1]

# Build the decision tree
decision_tree <- build_decision_tree(features, target_
    variable)

# Make predictions on the test data
predictions <- predict(features, decision_tree)

# Evaluate the model
accuracy <- mean(predictions == target_variable)

# Print the accuracy
print(accuracy)
```

Listing 4.1: Decision Tree Function

Chapter 5

Random Forrest

Random forest algorithms shine in both classification and regression tasks, boasting impressive accuracy and interpretability. Imagine a vibrant ensemble of decision trees, each casting their vote for the best prediction. That's the essence of random forests. However, this seemingly simple framework conceals a complex orchestra of statistical concepts:

Bagging (Bootstrap Aggregating): In bagging, each tree in the forest is trained on a randomly sampled subset (with replacement) of the original data. This approach injects diversity into the ensemble, reducing variance and mitigating overfitting. To achieve this, you draw N random samples (with replacement) from the original data, each containing the same number of data points as the original dataset (N). These individual samples become the training sets for the corresponding trees in the forest.

Feature Randomness: At each split within a tree, instead of considering all features, a randomly selected subset (typically the square root of the total number of features) is presented as potential candidates. This deliberate randomization strategy yields several benefits:

1. Decorrelation of Trees: By restricting each tree to a unique subset of features, feature randomness encourages individual trees to explore distinct patterns within the data. This results in a less correlated ensemble, reducing the likelihood of overfitting and enhancing overall predictive performance.
2. Robustness to Irrelevant Features: Feature randomness mitigates the influence of irrelevant or noisy features on the model's decisions. By randomly excluding certain features from splits, the algorithm becomes less susceptible to overfitting these features, leading to more robust and generalizable models.

Voting (Classification): For classification tasks, the final prediction is determined by the majority vote of the individual trees' predictions for a given data point. In other words, the class with the most votes from the trees is predicted as the outcome.

Voting (Regression): In regression tasks, the final prediction is the average of the individual trees' predictions for a data point. This approach balances the predictions from each tree and provides a more robust estimate for the continuous outcome variable.

Prediction Workflow: To predict the outcome for a new data point, simply pass it through each tree in the forest. Each tree will make its own prediction based on its internal decision rules. Finally, aggregate the individual predictions according to the voting scheme (majority for classification, average for regression) to obtain the final prediction for the new data point.

To build a Random Forrest keep in mind these main steps:

1. Train a number of decision trees on different subsets of the data
2. Make predictions by averaging the predictions of the decision trees
3. Evaluate the model using a metric such as accuracy

```r

# Train a number of decision trees on different subsets of
   the data
train_random_forest <- function(features, target_variable,
   num_trees) {
  # Create a list to store the decision trees
  decision_trees <- list()

  # Train a number of decision trees on different subsets of
   the data
  for (i in 1:num_trees) {
    # Sample a subset of the data with replacement
    bootstrapped_data <- sample(data, replace = TRUE)

    # Train a decision tree on the bootstrapped data
    decision_tree <- build_decision_tree(bootstrapped_data[,
   -1], bootstrapped_data$target_variable)

    # Add the decision tree to the list
    decision_trees[[i]] <- decision_tree
  }

  # Return the list of decision trees
  return(decision_trees)
}

# Make predictions using the random forest
predict <- function(features, decision_trees) {
  # Make predictions using each decision tree
  predictions <- sapply(decision_trees, predict, features)

  # Average the predictions from the decision trees
  predictions <- mean(predictions, na.rm = TRUE)

  # Return the predictions
  return(predictions)
}

# Load the data
data <- read.csv("data.csv")

```

```r
# Extract the features and target variable
features <- data[, -1]
target_variable <- data[, 1]

# Train the random forest
random_forest <- train_random_forest(features, target_
    variable, num_trees = 100)

# Make predictions on the test data
predictions <- predict(features, random_forest)

# Evaluate the model
accuracy <- mean(predictions == target_variable)

# Print the accuracy
print(accuracy)
```

Listing 5.1: Random Forrest Function

Random forests, due to their ensemble nature, often achieve higher accuracy compared to individual decision trees. However, this advantage comes at the cost of interpretability. Unlike decision trees, which offer a clear view of their decision-making process, random forests function as black boxes, making it harder to understand how they arrive at their predictions. This lack of transparency also poses challenges in tuning their parameters for optimal performance. On the other hand, random forests possess several other attractive properties. Their use of bagging (randomly selecting subsets of data to train multiple decision trees) leads to a reduction in variance and sensitivity to noise, making them robust to overfitting. Additionally, the introduction of randomness in feature selection during training enhances their robustness to correlated features, mitigating the effects of multicollinearity. However, it's important to note that these benefits come with a trade-off: training a random forest can be computationally expensive compared to fitting a single decision tree.

Chapter 6

Support Vector Machine (SVM)

Support Vector Machines (SVMs) are powerful classification algorithms known for their ability to learn complex relationships between features and target variables. At their core, SVMs aim to construct a hyperplane, a multidimensional 'separator,' that divides the data into distinct groups. This hyperplane is chosen specifically to maximize the margin, the distance between the hyperplane and the closest data points on either side. These closest points, called support vectors, play a crucial role in determining the SVM's performance. A larger margin translates to increased resilience to noise and improved generalization abilities, meaning the model performs well on unseen data.

Our data will be represented as x_i in an n-dimensional space, where n is the number of features in our data. Our data will have labels of y_i ϵ -1,1 to represent class membership for each data point.

Hyperplane Model: We can set up our margin using the following

$$w^T * x + b = 0$$

Here w is the normal vector defining the hyperplane's direction, and b is the bias term determining its offset.

Margin Maximization: We can then maximize the distance between our hyperplane and the closest support vectors by maximizing the following term:

$$||w||^{-1}$$

We can then solve the constrained optimization problem to find the optimal w and b, using the following:

$$Minimize : ||w||^2$$
$$Subject\,To : y_i * (w^T * x_i + b) \geq 1\,for\,all\,i$$

Lagrange Multiplier Method: allows us to convert the optimization problem into maximizing a quadratic function with respect to Lagrange multipliers α_i. This approach ultimately yields the support vectors, which enables us to construct the hyperplane based on their weighted combination.

1. Consider the following constrained problem:

$$\begin{array}{l} Minimize : f(x) \\ SubjectTo : g(x) = 0 \end{array}$$

Here $f(x)$ is the objective function we want to minimize, $g(x)$ is the equality constraint, our allowed region, and x is the vector of decision variables we want to optimize. This method introduces a new variable, λ. λ is simply the rate of change of the quantity being optimized, with respect to the constraint value.

This constructs the following function:

$$L(x, \lambda) = f(x) + \lambda * g(x)$$

Thus we intigrate the constraint directly into the objective.

2. Then, by minimizing the Lagrangian function with respect to both x and λ, we can simultaneously find the solution that minimizes $f(x)$ while adhering to the constraint $g(x) = 0$. This minimization process involves setting the partial derivatives of the Lagrangian function with respect to each variable, including λ, equal to zero. This yields a system of equations, known as the Lagrangian gradient vector, that we can solve to determine the optimal values of x and λ that satisfy both the objective function and the constraint.

Kernel Trick: Kernels allow for a transformation of our data into higher dimensional spaces enabling linear separation of what was non-linearly separable data.

Kernel Options:

1. Linear kernel: this kernel simply calculates the dot product between the data points
2. Gaussian kernel: This kernel calculates the similarity between data points using a Gaussian function
3. Polynomial kernel: This kernel calculates the similarity between data points using a polynomial function

Slack Variables: we can use variables to introduce slack, thus creating soft margin hyperplanes that can accommodate noisy data and improve resilience.

To build an SVM algorithm from the ground up, you would need to implement the following steps:

1. Calculate the kernel matrix for the features

2. Solve the quadratic programming problem to find the support vectors and the bias term
3. Make predictions by calculating the kernel product between the new data points and the support vectors, and then adding the bias terms

```

# Calculate the kernel matrix
calculate_kernel_matrix <- function(features, kernel = "linear") {
  # If the kernel is linear, return the dot product of the features
  if (kernel == "linear") {
    return(features %*% t(features))
  }

  # If the kernel is Gaussian, return the Gaussian kernel matrix
  if (kernel == "gaussian") {
    # Calculate the squared Euclidean distances between the features
    squared_euclidean_distances <- dist(features, method = "euclidean")^2

    # Calculate the Gaussian kernel matrix
    kernel_matrix <- exp(-squared_euclidean_distances / 2 * sigma)

    # Return the kernel matrix
    return(kernel_matrix)
  }

  # If the kernel is polynomial, return the polynomial kernel matrix
  if (kernel == "polynomial") {
    # Calculate the polynomial kernel matrix
    kernel_matrix <- (features %*% t(features) + 1)^degree

    # Return the kernel matrix
    return(kernel_matrix)
  }
}

# Train a support vector machine
train_svm <- function(features, target_variable, kernel = "linear", C = 1) {
  # Calculate the kernel matrix
  kernel_matrix <- calculate_kernel_matrix(features, kernel)

  # Solve the quadratic programming problem
  alpha <- solve_qp(kernel_matrix, target_variable, C)

  # Calculate the support vectors
  support_vectors <- features[alpha > 0, ]

  # Calculate the bias term
  bias_term <- mean(target_variable[alpha > 0]) - sum(alpha[
   alpha > 0] * target_variable[alpha > 0] * kernel_matrix[
   alpha > 0, alpha > 0])

  # Return the trained SVM model
  return(list(support_vectors = support_vectors, bias_term =
   bias_term))
}

# Make predictions using the support vector machine
predict <- function(features, svm_model) {
  # Calculate the kernel matrix between the features and the
   support vectors
  kernel_matrix <- calculate_kernel_matrix(features, kernel =
    svm_model$support_vectors)

  # Calculate the predictions
  predictions <- sum(alpha[alpha > 0] * target_variable[alpha
    > 0] * kernel_matrix) + svm_model$bias_term

  # Return the predictions
  return(predictions)
}

# Load the data
data <- read.csv("data.csv")

# Extract the features and target variable
features <- data[, -1]
target_variable <- data[, 1]

# Train SVM models with different kernels
linear_svm_model <- train_svm(features, target_variable,
    kernel = "linear", C = 1)
gaussian_svm_model <- train_svm(features, target_variable,
    kernel = "gaussian", C = 1)
polynomial_svm_model <- train_svm(features, target_variable,
    kernel = "polynomial", C = 1, degree = 3)

# Make predictions on the test data using the different SVM
```

```
    models
linear_predictions <- predict(features, linear_svm_model)
gaussian_predictions <- predict(features, gaussian_svm_model)
polynomial_predictions <- predict(features, polynomial_svm_
    model)

# Evaluate the different SVM models
linear_accuracy <- mean(linear_predictions == target_variable
    )
gaussian_accuracy <- mean(gaussian_predictions == target_
    variable)
polynomial_accuracy <- mean(polynomial_predictions == target_
    variable)

# Print the accuracies
print(linear_accuracy)
print(gaussian_accuracy)
print(polynomial_accuracy)
```

Listing 6.1: Support Vector Machine Function

Support vector machines (SVMs) often boast superior accuracy, especially in high-dimensional spaces, but come at the cost of higher computational demands. Their strength lies in robustness to noise, achieved through margin maximization, which reduces sensitivity to outliers. Additionally, kernel tricks enable SVMs to adapt effectively to non-linear data. While SVMs can offer insights into key features through the support vectors, interpreting the coefficients in the weight vector (w) and bias term (b) can be challenging. Finally, feature scaling plays a crucial role in optimizing SVM performance.

Chapter 7

Bagging and Boosting

7.1 Bagging

Bagging, or Bootstrap Aggregating, is a powerful ensemble method that combines multiple machine learning models to improve predictive accuracy and performance. It achieves this by training individual models on different bootstrap samples of the original data, leading to greater stability and reduced variance compared to single models. Consider a collection of models:

$$f_1(x), f_2(x), ..., f_B(x)$$

Each are trained on a bootstrap sample, randomly drawn with replacement, from the data, and then the bagging aggregates their predictions to form a final ensemble prediction:

$$F(x) = \frac{1}{B}\Sigma f_b(x)$$

To build a Bagging algorithm from the ground up, you would need to implement the following steps:

1. Create a function to train a base model
2. Train multiple models on different subsets of the data with replacement
3. Make predictions using each base model
4. Average the predictions of the base models
5. Return the average predictions

```

train_base_model <- function(features, target_variable) {
  # Train a decision tree model
  decision_tree_model <- rpart(target_variable ~ features)

  # Return the model
  return(decision_tree_model)
}

# Bagging function
bagging <- function(features, target_variable, num_trees) {
  # Create a list to store the base models
  base_models <- list()

  # Train multiple base models on different subsets of the
   data with replacement
  for (i in 1:num_trees) {
    # Sample a subset of the data with replacement
    bootstrapped_data <- sample(data, replace = TRUE)

    # Train a base model on the bootstrapped data
    base_model <- train_base_model(bootstrapped_data[, -1],
   bootstrapped_data$target_variable)

    # Add the base model to the list
    base_models[[i]] <- base_model
  }

  # Make predictions using each base model
  predictions <- sapply(base_models, predict, features)

  # Average the predictions of the base models
  average_predictions <- mean(predictions, na.rm = TRUE)

  # Return the average predictions
  return(average_predictions)
}

# Load the data
data <- read.csv("data.csv")

# Extract the features and target variable
features <- data[, -1]
target_variable <- data[, 1]

# Train a bagging ensemble
predictions <- bagging(features, target_variable, num_trees =
    100)

# Make predictions using the bagging ensemble
predictions <- predict(features, predictions)

# Evaluate the model
accuracy <- mean(predictions == target_variable)

# Print the accuracy
print(accuracy)
```

Listing 7.1: Bagging Function

Overall, bagging offers several advantages. It reduces variance in model predictions, leading to more stable and reliable models. This is due to the ensemble nature of bagging, where multiple models trained on different bootstrapped samples are combined. Additionally, bagging can improve overall accuracy compared to a single model. Furthermore, its parallelizability makes it computationally efficient, especially for large datasets. However, bagging comes with some drawbacks. It can increase model bias compared to a single model, and the resulting ensemble model may be less interpretable due to the combined complexity of multiple models.

7.2 Boosting

Boosting is another ensemble method that harnesses the power of multiple weak learners to achieve enhanced predictive accuracy. Unlike methods like bagging that average model predictions, boosting takes a sequential approach. It iteratively builds an ensemble by calling upon weak learners one after another, with each new learner focusing on correcting the errors made by its predecessors. This progressive refinement, where each model builds upon the knowledge of the previous ones, gradually improves the overall prediction accuracy.

For our example, the data will have true labels Y_i and $f_m(x_i)$ will be the prediction for the m-th learner on the i-th data point. The weight assigned to that learner will be represented as α_m, determining the influence it has in the final prediction. Finally $F_m(x)$ is the final prediction of the ensemble after the m-th iteration, a weighted sum of individual predictions. Thus the core optimization problem becomes:

$$\begin{array}{l} Minimize : \Sigma L(Y_i, F_m(x_i)) \\ SubjectTo : \Sigma|\alpha_m| \leq \ C \end{array}$$

where L is a specific loss function chosen, and C is the regularization parameter controlling the complexity of the model.

From here we start with our simple initial model, and then go into stepwise learning in which we fit a weak learner to the current residuals, calculate α_m such that it minimizes the chosen loss function and then update the ensemble prediction given by:

$$F_m(x) = F_{(m-1)(x)} + \alpha_m * f_m(x)$$

We then repeat the previous steps until a stopping criterion, such as maximum iterations or convergence threshold, is met. There are 3 main algorithmic variations, each utilizing a different combination of weak learners and loss functions, AdaBoost, Gradient Boosting and XGBoost.

AdaBoost: This algorithm focuses on "hard-to-learn" data points, assigning higher weights to those previously misclassified. This emphasis on challenging cases makes AdaBoost highly interpretable, as individual learner contributions are readily apparent. Additionally, its robustness to noise and outliers makes it a versatile choice for various tasks. However, AdaBoost is sensitive to feature scaling and can be susceptible to overfitting if not carefully tuned.

1. We use the following prediction function:

$$F_m(x) = \Sigma \alpha_i * f_i(x)$$

2. We use the following weight update:

$$alpha_i = exp(-\eta * L(Y_i, F_{(m-1)(x_i)}))$$

in which η is the learning rate that controls weight adjustments.

Gradient Boosting: This ensemble machine learning technique stands out for its innovative use of gradient descent to iteratively improve model performance. It operates by sequentially constructing a series of weak learners (typically decision trees), each designed to rectify the errors of its predecessors. The algorithm achieves this refinement by calculating the gradient of the loss function, a measure of how the model's predictions diverge from the desired outcomes. By identifying the direction of steepest descent within this gradient, gradient boosting can systematically guide model updates towards minimizing prediction error.

1. We use the following loss function gradient:

$$g_i(Y_i, F_{(m-1)(x_i)})$$

2. We use the following learner prediction:

$$f_m(x) = argmin\, \Sigma L(Y_i, F_{(m-1)(x_i)} + \alpha * f(x))$$

where α is the weight optimized for each learner based on our loss function.

XGBoost: Building upon the successful foundation of gradient boosting, XGBoost offers a sparsity-aware algorithm. This means it can efficiently handle sparse data (data with many missing values) and incorporates regularization to prevent overfitting, leading to improved generalization and higher accuracy, especially on complex datasets.

XGBoost further boasts faster training by leveraging distributed computing across multiple cores.

Boosting Algorithms	
AdaBoost	Interpretability, Multi-Class Classification
Gradient Boosting	Flexibility, Diverse Loss Functions, Regression, Classification
XGBoost	High Accuracy, Efficiency, Complex Tasks

To build a Boosting algorithm from the ground up, you would need to implement the following steps:

1. Initialize the weights of the training data
2. Train a base model on the weighted training data
3. Calculate the pseudo-residuals for each training point
4. Update the weights of the training data using the pseudo-residuals
5. Repeat steps 1-4 until the stopping criterion is met
6. Return the trained boosting model

```

# Boosting function
boosting <- function(features, target_variable, num_trees) {
  # Initialize the weights of the training data
  weights <- rep(1 / nrow(features), nrow(features))

  # Create a list to store the base models
  base_models <- list()

  # Train multiple base models on the weighted training data
  for (i in 1:num_trees) {
    # Train a base model on the weighted training data
    base_model <- train_base_model(features, target_variable,
    weights)

    # Calculate the pseudo-residuals for each training point
```

```r
    pseudo_residuals <- calculate_pseudo_residuals(base_model
    , features, target_variable)

    # Update the weights of the training data using the
   pseudo-residuals
    weights <- update_weights(weights, pseudo_residuals)

    # Add the base model to the list
    base_models[[i]] <- base_model
  }

  # Return the trained boosting model
  return(base_models)
}

# Calculate the pseudo-residuals for each training point
calculate_pseudo_residuals <- function(base_model, features,
    target_variable) {
  # Make predictions using the base model
  predictions <- predict(base_model, features)

  # Calculate the pseudo-residuals
  pseudo_residuals <- (target_variable - predictions) / (1 +
    exp(-2 * predictions))

  # Return the pseudo-residuals
  return(pseudo_residuals)
}

# Update the weights of the training data using the pseudo-
    residuals
update_weights <- function(weights, pseudo_residuals) {
  # Update the weights
  weights <- weights * exp(pseudo_residuals)

  # Normalize the weights
  weights <- weights / sum(weights)

  # Return the updated weights
  return(weights)
}

# Train a base model
train_base_model <- function(features, target_variable,
    weights) {
  # Train a decision tree model
  decision_tree_model <- rpart(target_variable ~ features,
    weights = weights)

  # Return the model
  return(decision_tree_model)
}

# Load the data
data <- read.csv("data.csv")

# Extract the features and target variable
features <- data[, -1]
target_variable <- data[, 1]

# Train a boosting ensemble
boosting_model <- boosting(features, target_variable, num_
    trees = 100)

# Make predictions using the boosting ensemble
predictions <- predict(boosting_model, features)

# Evaluate the model
accuracy <- mean(predictions == target_variable)

# Print the accuracy
print(accuracy)
```

Listing 7.2: Boosting Function

Boosting presents another powerful tool for statistical learning, offering strong accuracy and remarkable flexibility. Notably, XGBoost within the boosting family demonstrates an additional advantage: the ability to handle missing values effectively. However, these benefits come at a cost. Compared to simpler models, boosting algorithms often sacrifice interpretability, making it harder to understand their inner workings. Additionally, they can be prone to overfitting, especially if not carefully tuned. Finally, certain boosting algorithms can be computationally expensive, requiring significant processing power, particularly for large datasets.

Chapter 8

Neural Networks

Neural Networks (NN) are a powerful type of machine learning model that can be used to solve a wide variety of problems, including image recognition, natural language processing, and machine translation. These models are inspired from the architecture of the human brain, using neurons, or nodes, as building blocks for the overall algorithm.

Neuron: each neuron, the fundamental unit of a neural network, receives inputs $x_1, x_2, ..., x_n$ and produces an output y through a non-linear activation function σ:

$$y = \sigma(b + \Sigma(w_i * x_i))$$

where w_i are the weights representing the connection's strengths between neurons, b our bias term, adjusting for the neuron's overall activation threshold, and σ our activation function, introducing non-linearity. Using these we form a layered architecture in which an input layer, our first layer receives raw input data, n number of hidden or intermediate layers of neurons transform the data into higher-level representations, and an output or final layer produces our final predictions. to achieve this we use two concepts, forward and backward propagation.

Forward Propagation: signals flow through the network, begining from an input layer, and ending in the output layer. A simple example would be:

$$\begin{aligned}
&a_1 = \sigma(W_1 * x + b_1)\\
&a_2 = \sigma(W_2 * a_1 + b_2)\\
&...\\
&y = \sigma(W_L * a_{(L-1)} + b_L)
\end{aligned}$$

Backward Propagation: adjusts the weights biases based on error signals propagated back through the network, allowing the network to "learn". First it calculates an output error, comparing the predictions made with actual labels using a loss function such as MSE. Then it back-propagates the error, calculating gradients of the loss function with respect to the weights and biases using the chain rule. Finally, it updates the parameters, adjusting the weights and biases using an optimization algorithm such as gradient descent, to minimize the loss.

To build a neural network algorithm from the ground up, you would need to implement the following steps:

1. Define the network architecture. This includes the number of layers, the number of neurons in each layer, and the activation function for each layer.
2. Initialize the weights and biases. This can be done randomly or using a pre-trained model.
3. Forward pass. This involves calculating the output of each layer by multiplying the input of the layer by the weights and adding the bias.
4. Backward pass. This involves calculating the gradients of the loss function with respect to the weights and biases.
5. Update the weights and biases. This can be done using a variety of algorithms, such as gradient descent or stochastic gradient descent.
6. Repeat steps 3-5 until the convergence criterion is met.

```

# Neural network function
neural_network <- function(features, target_variable,
   architecture, epochs, learning_rate) {
  # Initialize the weights and biases
  weights <- initialize_weights(architecture)
  biases <- initialize_biases(architecture)

  # Train the neural network
  for (epoch in 1:epochs) {
    # Forward pass
    outputs <- forward_pass(features, weights, biases)

    # Calculate the loss function
    loss <- calculate_loss_function(outputs, target_variable)

    # Backward pass
    gradients <- backward_pass(loss, outputs, weights, biases
   )

    # Update the weights and biases
    update_weights_and_biases(weights, biases, gradients,
   learning_rate)
  }

  # Return the trained neural network model
  return(list(weights = weights, biases = biases))
}

# Forward pass
forward_pass <- function(features, weights, biases) {
```

```r
  # Calculate the output of each layer
  outputs <- list()
  for (i in 1:length(architecture)) {
    outputs[[i]] <- weights[[i]] %*% features + biases[[i]]

    # Apply the activation function
    outputs[[i]] <- apply_activation_function(outputs[[i]])
  }

  # Return the outputs
  return(outputs)
}

# Backward pass
backward_pass <- function(loss, outputs, weights, biases) {
  # Calculate the gradients of the loss function with respect to the outputs
  gradients <- calculate_gradients_loss_function(loss, outputs)

  # Calculate the gradients of the loss function with respect to the weights and biases
  gradients_weights <- list()
  gradients_biases <- list()
  for (i in length(architecture):1) {
    gradients_weights[[i]] <- gradients[[i]] %*% t(features)
    gradients_biases[[i]] <- apply(gradients[[i]], 1, sum)
  }

  # Return the gradients
  return(list(weights = gradients_weights, biases = gradients_biases))
}

# Update the weights and biases
update_weights_and_biases <- function(weights, biases, gradients, learning_rate) {
  # Update the weights
  for (i in 1:length(architecture)) {
    weights[[i]] <- weights[[i]] - learning_rate * gradients$weights[[i]]
  }

  # Update the biases
  for (i in 1:length(architecture)) {
    biases[[i]] <- biases[[i]] - learning_rate * gradients$biases[[i]]
  }
```

```
}

# Load the data
data <- read.csv("data.csv")

# Extract the features and target variable
features <- data[, -1]
target_variable <- data[, 1]

# Define the network architecture
architecture <- c(100, 10)

# Train the neural network
nn_model <- neural_network(features, target_variable,
    architecture = architecture, epochs = 100, learning_rate =
     0.01)

# Make predictions using the neural network
predictions <- predict_neural_network(features, nn_model)

# Evaluate the model
accuracy <- mean(predictions == target_variable)

# Print the accuracy
print(accuracy)
```

Listing 8.1: Boosting Function

Neural networks excel at handling non-linear relationships and modeling complex patterns. Their results often generalize well to new data sets, and they can learn adaptively from data without explicit programming. However, interpreting their results can be challenging, leading to the development of techniques like SHAP values for feature importance analysis. Additionally, their high computational cost and susceptibility to overfitting necessitate careful hyperparameter tuning.