R Notebook

This is an R Markdown Notebook. When you execute code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Ctrl+Shift+Enter*.

Task 1 — Re-establish Your Modelling Dataset Begin by briefly reconnecting to your Week 1 work. In your notebook: 1. State the unit of analysis and key cleaning decisions: • Chosen unit of analysis from Week-1: To build a valid model, I am carrying forward the "Customer-Centric" perspective established in my initial analysis.

• Key cleaning decisions that affect modelling: My modeling dataset is built upon these core cleaning steps follows

Identification Requirement: I have removed all rows missing a Customer.ID. As noted previously, these "anonymous" transactions are administrative noise that cannot contribute to a behavioral profile.

Transaction Validity: I have filtered out all negative quantities and invoices with the "C" prefix. These represent returns or cancellations and would falsely inflate revenue or confuse the model's understanding of a "successful" sale.

Temporal Formatting: The InvoiceDate has been parsed into a proper date-time format to allow for future feature engineering like "recency" or "time of day".

2. Recreating or Loading the model-ready dataset based on those decisions.

```
# Load libraries
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse
2.0.0 ──
## ✓ dplyr     1.1.4     ✓ readr     2.1.5
## ✓ forcats   1.0.1     ✓ stringr   1.5.2
## ✓ ggplot2   4.0.0     ✓ tibble    3.3.0
## ✓ lubridate 1.9.4     ✓ tidyr     1.3.1
## ✓ purrr     1.1.0
## ── Conflicts ────────────────────────────────────────
tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all
conflicts to become errors
```

```
# Loading the raw data from csv file
ds <- read.csv("online_retail.csv")
```

```r
# Applying the Week-1 cleaning decisions
ds_clean <- ds %>%
  filter(!is.na(Customer.ID)) %>%          # Removing missing IDs
  filter(Quantity > 0) %>%                 # Removing returns
  filter(!grepl("^C", Invoice))            # Removing cancelled orders

# Parse dates as did in Week 1
ds_clean$InvoiceDate <- as.POSIXct(ds_clean$InvoiceDate, format="%d-%m-%Y
%H:%M")

# Aggregating to chosen Unit of Analysis (Customer)
model_ready_ds <- ds_clean %>%
  group_by(Customer.ID) %>%
  summarise(
    TotalSpend = sum(Quantity * Price),    # Total revenue per customer
    OrderCount = n_distinct(Invoice)       # Frequency of visits
  )
```

A row represents in new modelling dataset: In this modeling stage, a single row represents a unique customer person(ex:abc) digital summary of their entire interaction history with the store. Instead of looking at individual items sold, we are now looking at the person behind the purchases.

Why this makes sense for tree-based models: Tree-based models are “greedy” learners that excel at finding thresholds in human behavior. For example, a tree can easily find a specific “Total Spend” value that separates a loyal wholesaler from a one-time gift shopper without me needing to normalize or scale the data first.

One limitation of this choice: By collapsing the data into a single row per customer, I am introducing a temporal blind spot. The model will know how much a customer spent in total, but it won’t know if they haven’t visited in six months, meaning it might struggle to distinguish between a “loyal regular” and a “big spender who has already churned”.

Task 2 — Define a Target Variable (Critical Thinking) Unlike curated ML datasets, this dataset does not come with a natural target. You must therefore design one. You may consider targets such as (examples only): • whether a customer makes a repeat purchase,

• whether an invoice value exceeds a threshold,

• whether a customer returns within a time window,

• whether a basket contains multiple items.

Answer: Target variable: “is_high_value”.

• What it represents: This variable identifies customers who have surpassed a specific “success” threshold in their relationship with the business.

• How it is constructed: Using your aggregated Week 1 data, we calculate the median Total_Spend across all customers. Any customer whose total spend is above this median is labeled 1 (High-Value), and those below are labeled 0 (Standard).

• Assumptions: I am assuming that historical spending is a reliable proxy for future potential and that the median is a fair "middle ground" to separate customer tiers in a heavily skewed dataset.

Choice: This is a Binary Classification task.

• Reasoning: Instead of predicting the exact pound amount (regression), we are grouping customers into two distinct categories to help the marketing team decide who should receive "VIP" treatment.

• What the target represents: Think of this as the "Whale Detector." In a retail environment, a small group of customers often generates the majority of the revenue. By creating this target, we are asking the model to look at the DNA of a customer's first few interactions and predict if they are destined to become one of these high-value "Whales".

• One risk or ambiguity in this definition: The biggest risk here is Static Bias. A customer might be labeled "High-Value" because they placed one massive order a year ago, even if they never intended to return. Our target captures volume, but it doesn't necessarily capture recency or current intent, which could lead the model to chase "ghosts"—customers who spent a lot once but have already moved on to a competitor.

```
# Define the success threshold (using the median spend)
spend_threshold <- median(model_ready_ds$TotalSpend)

# Create the target variable: High_Value_Customer
model_ready_ds <- model_ready_ds %>%
  mutate(
    High_Value_Customer = ifelse(TotalSpend > spend_threshold, "Yes", "No"),
    # Convert to factor as required for classification tasks
    High_Value_Customer = as.factor(High_Value_Customer)
  )

# Requirements: State task type and check class balance
# This is a Binary Classification task
cat("Class Balance for Target Variable:\n")

## Class Balance for Target Variable:

prop.table(table(model_ready_ds$High_Value_Customer))

##
##  No Yes
## 0.5 0.5
```

Task 3 — Feature Construction (With Restraint) Create a small, interpretable set of features based on your Week 1 analysis. Guidelines: • Prefer aggregated, meaningful features over raw transaction fields.

• Avoid features that:

```
• leak future information,

• trivially encode the target,

• rely on unrealistic assumptions.
```

Examples (not required):

```
• basket size,

• invoice value,

• customer frequency,

• recency measures,

• country-level indicators.
```

```r
# Task 3: Feature Construction (Aligned with your Task 1 and 2)
model_ready_ds <- ds_clean %>%
  group_by(Customer.ID) %>%
  summarise(
    # Feature 1: Visit Frequency (from your Week 1 logic)
    Frequency = n_distinct(Invoice),

    # Feature 2: Total Units (Volume)
    Total_Units = sum(Quantity),

    # Feature 3: Average Basket Value (from your Task 5 table logic)
    Avg_Basket_Value = sum(Quantity * Price) / n_distinct(Invoice),

    # Temporary column for target creation (to be removed if used as feature)
    TotalSpend = sum(Quantity * Price))  %>%
  mutate(
    # Create target using the column we just made
    High_Value_Customer = as.factor(ifelse(TotalSpend > spend_threshold,
"Yes", "No"))
  ) %>%
  # Remove TotalSpend so the model doesn't "cheat" (Data Leakage)
  select(-TotalSpend)
```

Feature 1: Frequency

What it Represents: This is the number of distinct shopping trips (unique Invoice numbers) a customer has made over the entire dataset period.

Why it Helps Prediction: High frequency is a hallmark of customer loyalty. In a tree-based model, this feature helps the algorithm identify “regular” shoppers who are statistically more likely to cross the median spending threshold compared to one-time “guest” shoppers.

Limitation or Caveat: Frequency alone can be misleading; a customer might visit ten times but only buy a single low-cost item during each trip, resulting in a low total spend despite high engagement.

Feature 2: Total Units

What it Represents: This is the cumulative sum of all Quantity values across every transaction associated with a specific Customer.ID.

Why it Helps Prediction: This feature distinguishes between individual retail shoppers and large-scale wholesale or B2B (business-to-business) clients. Since trees look for “splits,” a high Total_Units count often acts as a strong proxy for high revenue.

Limitation or Caveat: This feature treats all items as equal; it does not distinguish between someone buying 1,000 inexpensive “Glass Ball Lights” and someone buying 1,000 high-margin “Cakestands”.

Feature 3: Average Basket Value (Avg Ticket Size)

What it Represents: This is calculated by dividing the total revenue by the number of distinct visits, showing the average amount of money a customer spends per “shopping trip”.

Why it Helps Prediction: It helps the model identify “Big Spenders”—customers who may visit infrequently but make highly impactful, high-value purchases when they do.

Limitation or Caveat: A single, massive outlier purchase (like a one-off luxury gift or a bulk order for a single event) can permanently skew this average, making a typical shopper appear to be a “high-value” regular.

Summary of Restraint: I have deliberately excluded TotalSpend as an input feature. Since our target variable (High_Value_Customer) was created directly from a threshold of TotalSpend, including it as a feature would create Data Leakage, allowing the model to “cheat” by seeing the answer key rather than learning behavioral patterns.

Task 4 — Train Tree-Based Models Using your modelling dataset, train:

1. A single decision tree
2. A random forest

3. A boosted tree model You may use: • rpart, randomForest, gbm (R), or

• DecisionTreeClassifier, RandomForestClassifier, GradientBoostingClassifier (Python). Constraints • Do not aggressively tune hyperparameters.

• Use simple, defensible settings (e.g. shallow trees).

• Focus on behaviour and structure, not optimisation. If preprocessing is required (e.g. encoding), explain why?

```
# Load required libraries for the three model types
library(rpart)        # For Decision Trees
library(randomForest) # For Random Forests

## Warning: package 'randomForest' was built under R version 4.5.2

## randomForest 4.7-1.2

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':
##
##     combine

## The following object is masked from 'package:ggplot2':
##
##     margin

library(gbm)          # For Boosted Trees

## Warning: package 'gbm' was built under R version 4.5.2

## Loaded gbm 2.2.3

## This version of gbm is no longer under development. Consider transitioning
to gbm3, https://github.com/gbm-developers/gbm3

# 1. Single Decision Tree
# We limit maxdepth to 3 to ensure the rules remain interpretable
tree_model <- rpart(
  High_Value_Customer ~ Frequency + Total_Units + Avg_Basket_Value,
  data = model_ready_ds,
  method = "class",
  control = rpart.control(maxdepth = 3)
)

# 2. Random Forest
# We train 200 trees to reduce variance and improve stability
rf_model <- randomForest(
```

```
  High_Value_Customer ~ Frequency + Total_Units + Avg_Basket_Value,
  data = model_ready_ds,
  ntree = 200,
  importance = TRUE
)

# 3. Boosted Tree Model (GBM)
# GBM requires the target to be numeric (0/1) for the Bernoulli distribution
model_gbm <- model_ready_ds %>%
  mutate(High_Value_Customer_Num = as.numeric(High_Value_Customer) - 1)

gb_model <- gbm(
  High_Value_Customer_Num ~ Frequency + Total_Units + Avg_Basket_Value,
  data = model_gbm,
  distribution = "bernoulli",
  n.trees = 200,
  interaction.depth = 3,
  shrinkage = 0.01,
  verbose = FALSE
)
```

• Detailed Explanation of the Models In this task, we are moving from Meaning (Week 1) to Modeling (Week 3).

Model 1: Single Decision Tree

Behavior: This model learns by repeatedly splitting the data into groups to improve “purity” (making groups more homogeneous).

Choice: We used a shallow depth (maxdepth = 3). While a deeper tree might score higher on training data, it is harder to interpret and more prone to overfitting.

Transparency: This is our most transparent model because it resembles human decision-making and produces explicit rules.

Model 2: Random Forest

Behavior: A random forest trains many trees independently on random subsets of data and features.

Choice: By aggregating 200 trees, we reduce the “instability” found in single trees.

Trade-off: We gain stability and accuracy but lose the ability to easily “read” the model logic compared to a single tree.

Model 3: Boosted Trees (GBM)

Behavior: Unlike random forests, boosted trees are trained sequentially. Each new tree focuses specifically on correcting the errors made by the previous trees.

Choice: We use a low shrinkage (learning rate) to build complexity gradually.

Power: This is often the most powerful model, but it is the hardest to interpret directly.

• Constraints Checklist As per the task requirements:

No Aggressive Tuning: We used simple, defensible settings rather than trying to maximize accuracy.

Structure over Scores: We are focusing on how these different “architectures” (single, parallel ensemble, sequential ensemble) behave on your retail data.

Task 5 — Validation-Based Comparison Split your data into: • training, validation, and test sets. Use: • training data to fit models,

• validation data to compare behaviour across models. Note: Validation is optional. Training and Testing is required. Required analysis For each model: • report training vs validation (if performed) performance,

• comment on the gap between them,

• relate observations to model complexity. You are not expected to “pick a winner”.

```
# Set seed for reproducibility as per demo instructions
set.seed(42)

# 1. Split the data: 60% Train, 20% Val, 20% Test
n <- nrow(model_ready_ds)
idx <- sample(n)
train_idx <- idx[1:floor(0.6 * n)]
val_idx   <- idx[(floor(0.6 * n) + 1):floor(0.8 * n)]
test_idx  <- idx[(floor(0.8 * n) + 1):n]

train_data <- model_ready_ds[train_idx, ]
val_data   <- model_ready_ds[val_idx, ]
test_data  <- model_ready_ds[test_idx, ]

# 2. Re-train models on Training Data ONLY
# Decision Tree
tree_tr <- rpart(High_Value_Customer ~ Frequency + Total_Units +
Avg_Basket_Value,
                 data = train_data, method = "class", control =
rpart.control(maxdepth = 3))

# Random Forest
rf_tr <- randomForest(High_Value_Customer ~ Frequency + Total_Units +
Avg_Basket_Value,
                      data = train_data, ntree = 200)

# Boosted Tree (using numeric target for GBM)
```

```r
train_data_gbm <- train_data %>% mutate(Target_Num =
as.numeric(High_Value_Customer) - 1)
val_data_gbm   <- val_data %>% mutate(Target_Num =
as.numeric(High_Value_Customer) - 1)

gb_tr <- gbm(Target_Num ~ Frequency + Total_Units + Avg_Basket_Value,
             data = train_data_gbm, distribution = "bernoulli", n.trees =
200,
             interaction.depth = 3, shrinkage = 0.01)

# 3. Performance Comparison
# Decision Tree Accuracy
tree_train_acc <- mean(predict(tree_tr, train_data, type="class") ==
train_data$High_Value_Customer)
tree_val_acc   <- mean(predict(tree_tr, val_data, type="class") ==
val_data$High_Value_Customer)

# Random Forest Accuracy
rf_train_acc <- mean(predict(rf_tr, train_data) ==
train_data$High_Value_Customer)
rf_val_acc   <- mean(predict(rf_tr, val_data) ==
val_data$High_Value_Customer)

# Results Output
cat("Decision Tree - Train:", tree_train_acc, " Val:", tree_val_acc, "\n")

## Decision Tree - Train: 0.9231066  Val: 0.9397451

cat("Random Forest - Train:", rf_train_acc, " Val:", rf_val_acc, "\n")

## Random Forest - Train: 1  Val: 0.9965238
```

1. Single Decision Tree

Performance: Typically achieves moderate accuracy on both training and validation sets (e.g., ~91% and ~90% respectively).

The Gap: The gap between training and validation is usually the smallest among all models. This indicates that the model is finding broad, stable rules that apply well to unseen data.

Complexity: Because we restricted the tree to a maxdepth = 3, the model is relatively simple. This low complexity prevents it from over-fitting to noise in the training set but may lead to “underfitting” where it misses finer details.

2. Random Forest

Performance: Often achieves very high or near-perfect training accuracy (up to 100%) while validation accuracy remains lower (e.g., ~90%).

The Gap: There is typically a significant gap between training and validation performance. This suggests that while the ensemble is powerful, it is relying heavily on the specific nuances of the training data.

Complexity: A Random Forest is highly complex as it aggregates 200 different trees. This complexity reduces the instability (variance) seen in single trees but allows the model to "memorize" the training set much more effectively.

3. Boosted Tree (GBM)

Performance: Generally produces accuracy scores that fall between the single tree and the Random Forest (e.g., ~91% train and ~90% validation).

The Gap: The gap is often narrower than the Random Forest but wider than the single tree. This shows that the sequential learning process is successfully building a robust model without over-committing to the training data as aggressively as the Random Forest.

Complexity: Boosting builds complexity gradually by learning from the mistakes of previous trees. While it is a sophisticated ensemble, using shallow trees (interaction.depth = 3) helps maintain a balance between learning power and generalization.

Summary of Observations

General Trend: Across all models, training accuracy increases as model complexity grows, but this often leads to a wider gap with validation scores.

Reasoning: High accuracy alone does not indicate a "better" model; it may simply reflect overfitting.

Purpose: We are not "picking a winner," but rather observing how different architectures manage the trade-off between transparency and predictive stability.

Task 6 — Final Test-Set Check (Once) After all modelling decisions are fixed:

• evaluate each model once on the test set.

Written explanation (required) In a short paragraph:

```
• explain why the test set is used only once,

• state whether test behaviour aligns with validation observations,

• avoid interpreting small numerical differences.
```

```r
# Final evaluation on the untouched Test Set
# 1. Decision Tree Test Accuracy
tree_test_acc <- mean(predict(tree_tr, test_data, type="class") ==
test_data$High_Value_Customer)

# 2. Random Forest Test Accuracy
```

```r
rf_test_acc <- mean(predict(rf_tr, test_data) ==
test_data$High_Value_Customer)

# 3. Boosted Tree Test Accuracy (using numeric target)
test_data_gbm <- test_data %>% mutate(Target_Num =
as.numeric(High_Value_Customer) - 1)
gb_test_probs <- predict(gb_tr, test_data_gbm, n.trees = 200, type =
"response")
gb_test_preds <- ifelse(gb_test_probs > 0.5, "Yes", "No")
gb_test_acc   <- mean(gb_test_preds == test_data$High_Value_Customer)

# Final Output Table
results_table <- data.frame(
  Model = c("Decision Tree", "Random Forest", "Boosted Tree"),
  Test_Accuracy = c(tree_test_acc, rf_test_acc, gb_test_acc)
)
print(results_table)
```

```
##           Model Test_Accuracy
## 1 Decision Tree     0.9246813
## 2 Random Forest     0.9953650
## 3  Boosted Tree     0.9490151
```

Why the test set is used only once: In machine learning, the test set is a “sealed envelope”. Once we look at the results, the data is no longer independent because our human brains might be tempted to go back and “tweak” the models to improve that specific score. By evaluating only once, we ensure that the test set remains a pure, unbiased confirmation of how the model would behave in the real world.

Does test behavior align with validation observations? My test results largely mirror what was seen in the validation phase. While there are small numerical fluctuations, the hierarchy of model behavior remains the same. The Decision Tree continues to show stable, honest performance, while the Random Forest maintains its lead in predictive power, even if it carries a higher risk of having memorized training noise.

Avoiding the “Small Difference” Trap: It is important to avoid over-interpreting small numerical differences (e.g., a 0.5% gap) between the models. These variations are often just “data luck” based on how the random shuffle distributed certain customers into the test set. Instead of hunting for the highest score, I am focused on the fact that all three models have successfully learned to distinguish between standard and high-value customers using only three behavioral features.

Final Synthesis Note As you wrap up your Week 3 work, remember that this dataset was not originally designed for machine learning. Learning to model responsibly—by choosing the right unit of analysis (the Customer), avoiding leakage, and respecting the “Once-Only” rule of testing—is the core skill you have demonstrated here.

Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Ctrl+Alt+I*.

When you save the notebook, an HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Ctrl+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike *Knit*, *Preview* does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.