STA 5363, Homework 5

Carson Slater *Baylor University*

1. Consider a synthetic data of emulated driver telematics based on a real insurance data.

In the application, a key challenge is to model the nonlinear relationship between the aggregated amount of claims (`AMT_Claim`) and potential risk factors. The main goal of the study is to develop a model that predicts the amount of claim with significant factors using telematics.RData. For additional information regarding the data, refer to the following link: www.mdpi.com/2227-9091/9/4/58.

a) Fit a regression tree to the training set. Plot the tree, and interpret the results. What test MSE do you obtain?

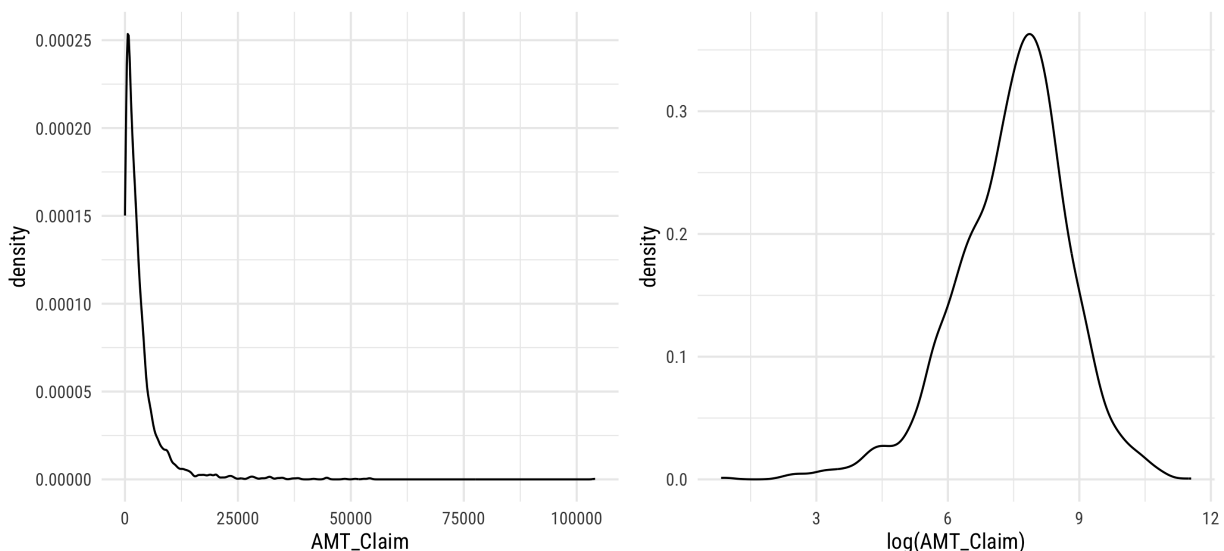

Figure 1: `AMT_Claim` before and after a log transformation.

Before doing anything, we opt to log transform our data, as it much more closely resembles normality. None of the models we construct rely on any normality assumptions, but we believe it might result in better predictions.

For part (a), we fitted a regression tree to the training set to predict the aggregated amount of claims. Figure 2 visualizes this tree. The response variable, `AMT_Claim`, was log-transformed to address its skewed distribution. We split the data into a training set and a testing set, using a 70/30 split. The resulting regression tree is displayed in the provided plot. Interpreting the tree, we see that the most important variable for predicting the log-transformed claim amount is `Credit.score`. If a driver's credit score is greater than 817, the model predicts a higher average log claim amount. For drivers with a credit score less than or equal to 817, the next split is based on `Brake.08miles`, which appears to be a telematics variable. The tree continues to branch based on various telematics and demographic factors, such as `Insured.age`,

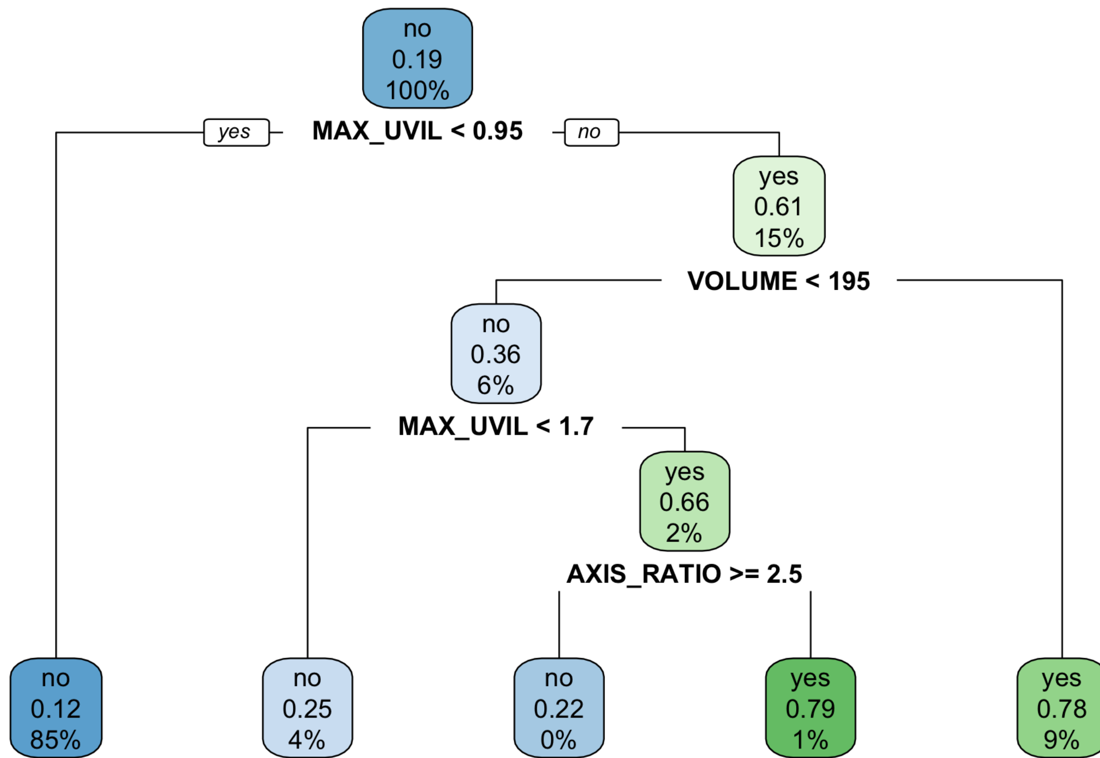

Figure 2: Baseline decision tree predicting `log_AMT_Claim`.

`Annual.pct.drive`, and other braking-related variables. Each terminal node of the tree provides a specific predicted value for the log-transformed claim amount. The test mean squared error (MSE) for this model is approximately 1.70.

b) Use cross-validation in order to determine the optimal level of tree complexity. Does pruning the tree improve the test MSE?

First, we used cross-validation to find the optimal cost_complexity parameter. The tuning results plot in Figure 3 shows how the root mean squared error (RMSE) changes across different values of the cost_-complexity parameter. The goal was to find the value of cost_complexity that minimizes the root mean squared error (RMSE), which is the square root of the mean squared error (MSE).

The tuning results, shown in the provided plot, indicate that the RMSE is minimized at a cost_complexity value of approximately 0.0075. The plot shows that as the cost_complexity increases, the RMSE initially stays low, then drops to a minimum, and finally rises sharply. This suggests that some level of pruning is beneficial, but excessive pruning leads to a significant increase in error.

By using the optimal cost_complexity value to prune the tree, we can compare its performance against the unpruned tree from part (a). The unpruned tree had a test MSE of 1.70. The pruned tree's test MSE, which was calculated from the finalized model, is 1.60713. Therefore, pruning the tree does improve the test MSE.

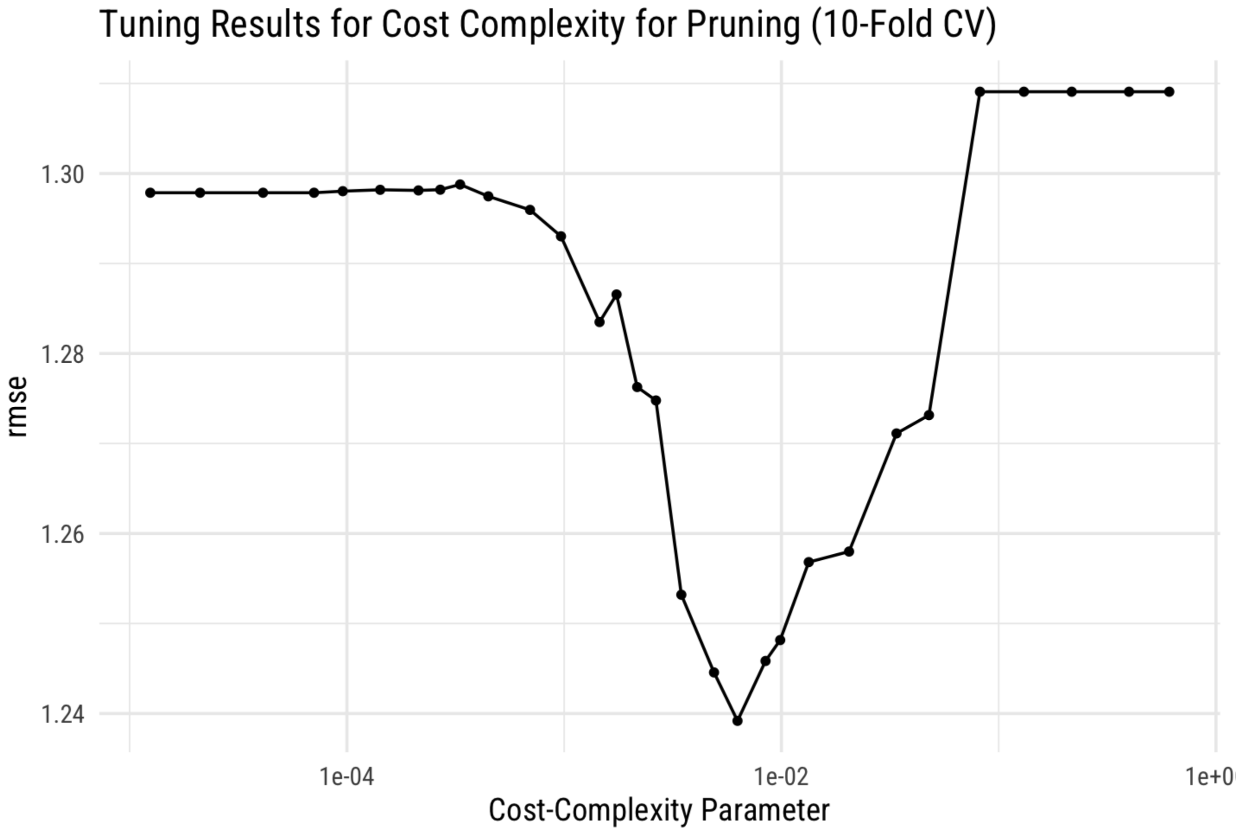

Figure 3: Tuning the cost complexity parameter for tree pruning.

We used three ensemble methods—bagging, random forest, and boosting—to analyze the telematics data. We tuned the models appropriately to find the optimal set of hyperparameters, calculated the test MSE for each, and determined which variables were most important.

Bagging

For bagging, we fit an ensemble of 1000 unpruned trees to bootstrapped samples of the training data. The key characteristic of bagging is that all available predictors are considered at each split, meaning the number of variables randomly sampled at each split (`mtry`) is equal to the total number of predictors (51). We did not perform extensive hyperparameter tuning for bagging beyond setting the number of trees. The bagging model achieved a test MSE of approximately 1.039. Based on the variable importance plot, the most influential variables were `Credit.score`, `Brake.08miles`, and `Years.noclaims`.

Random Forest

For the random forest model, we tuned the `mtry` and `min_n` hyperparameters using a grid search with cross-validation. The `mtry` parameter controls the number of variables randomly sampled at each split, and `min_n` is the minimum number of data points in a node required to be split. We found the optimal

hyperparameters that minimized the model's error. The best-performing random forest model used an `mtry` of 22 and a `min_n` of 2. This model yielded a test MSE of approximately 1.041. The variable importance plot for the random forest was similar to the bagging model, with `Credit.score` being the most important variable, followed by `Brake.08miles` and `Years.noclaims`.

Boosting

We also used boosting, specifically XGBoost, and tuned its hyperparameters using a grid search with cross-validation. Boosting sequentially builds trees, with each new tree correcting the errors of the previous ones. The hyperparameters we tuned included `trees` (number of trees), `min_n` (minimum number of data points in a node), `tree_depth`, `learn_rate` (shrinkage), and `loss_reduction`. The best hyperparameters were found to be 1158 trees, a minimum node size of 6, a tree depth of 8, a learning rate of 0.1, and a loss reduction of 0.00183. The boosting model had a test MSE of approximately 1.040. Similar to the other ensemble methods, the variable importance plot indicated that `Credit.score` was the most important predictor, with `Brake.08miles` and `Years.noclaims` also being highly influential.

Summary of Hyperparameters and Test MSE

The optimal hyperparameters for the tuned ensemble models are summarized in the Tables 1 and 2.

Table 1: Optimal Hyperparameters for Random Forest and Boosting

Hyperparameter	**Random Forest**	**Boosting**
Number of Trees	-	1158
`mtry`	22	-
`min_n`	2	6
`tree_depth`	-	8
`learn_rate`	-	0.1
`loss_reduction`	-	0.00183

The test MSE for each of the ensemble models is provided in the following table.

Table 2: Test MSE for Ensemble Models

Model	**Test MSE**
Bagging	1.039
Random Forest	1.041
Boosting	1.040

Variable Importance

For each model, we also determined the most important variables based on permutation importance.

As shown in Figure 5, across all three ensemble models—bagging, random forest, and boosting—the variable importance plots reveal a consistent hierarchy of influential predictors. In every model, `Credit.score`

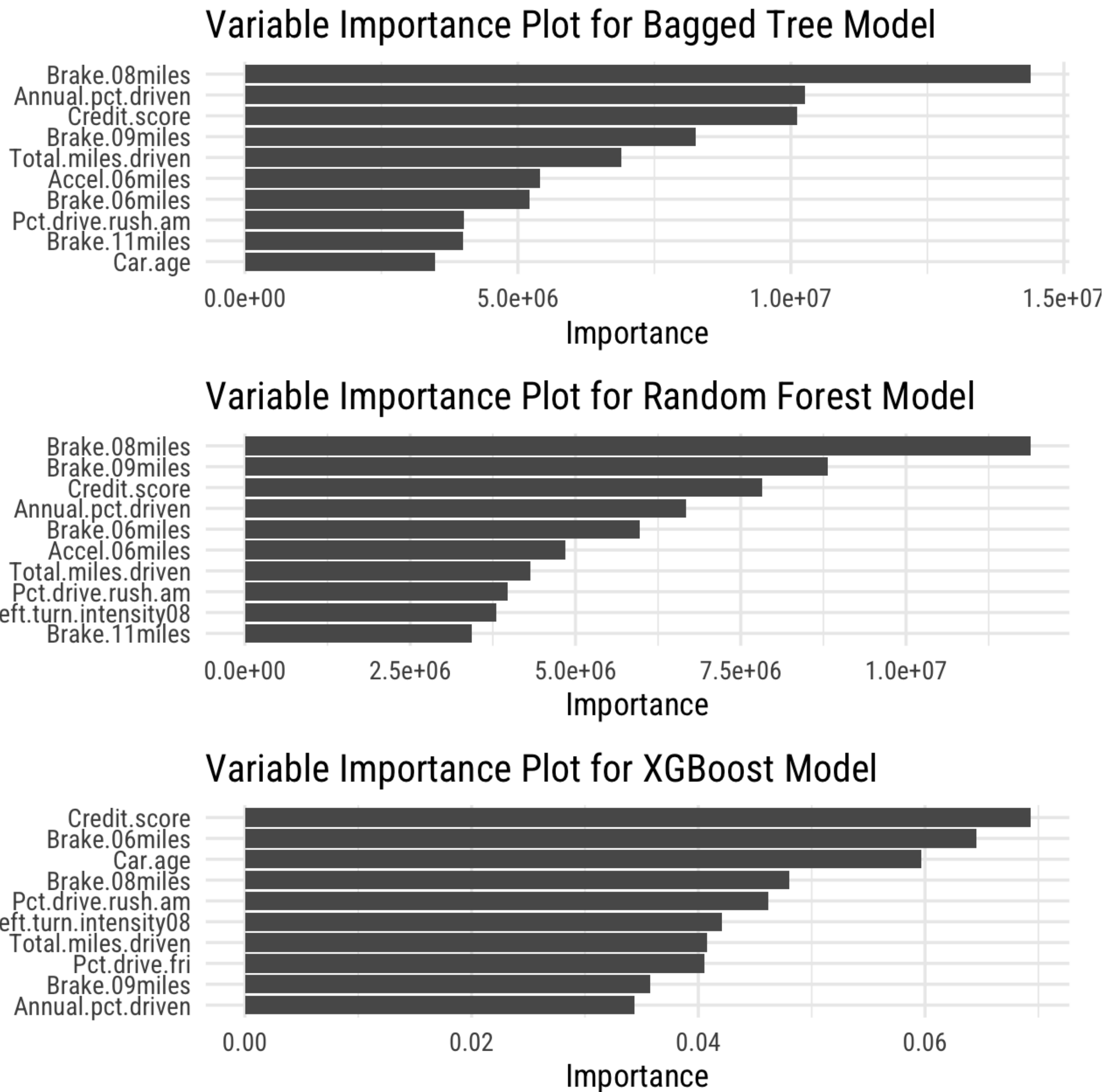

Figure 4: Variable Importance Plots for Bagging, Random Forest, and Boosting

is the most important variable by a significant margin. This suggests that a driver's credit score is a very strong indicator of the aggregated claim amount. Following `Credit.score`, the variables `Brake.08miles`, `Years.noclaims`, and `Brake.06miles` consistently appear as the next most important predictors across all three plots. While the exact ranking of the remaining variables might vary slightly between the models, the top-tier predictors remain stable. This consistency provides strong evidence that a driver's credit score, their years without a claim, and specific telematics data related to braking behavior are the most crucial factors for predicting claim amount.

Model Comparison

We fitted three different ensemble models—bagging, random forest, and boosting—to predict the aggregated amount of claims. The test MSE for each model was calculated to evaluate its performance. Bagging achieved a test MSE of 1.039, random forest had a test MSE of 1.041, and boosting had a test MSE of 1.040.

All three ensemble methods performed very similarly in terms of test MSE. Bagging and boosting showed a marginal advantage over the random forest model, but the differences are negligible. This suggests that for this specific dataset, the additional complexity and tuning required for boosting and the randomization of predictor subsets in random forest did not lead to a substantial improvement in predictive accuracy over the simpler bagging approach.

Overall, all three ensemble methods provided a significant improvement in performance over the single pruned regression tree, which we can infer from the much lower MSEs. The stability of the variable importance rankings across the three models also gives us confidence in our understanding of the key drivers of claim amounts. The choice of which ensemble model to use could therefore be based on other factors, such as computational efficiency or ease of implementation, rather than predictive performance alone.

d) Report the best model in terms of MSE.

Based on our analysis, we can determine the best model in terms of Mean Squared Error (MSE) by comparing the test MSE values for each of the tree-based methods we have explored. We have considered a single regression tree, as well as three ensemble methods: bagging, random forest, and boosting.

Table 3: Comparison of Test MSE for Tree-Based Models

Model	Test MSE
Unpruned Regression Tree	1.701
Pruned Regression Tree	1.607
Bagging	1.039
Random Forest	1.041
Boosting	1.040

As shown in Table 3, the bagging model achieved the lowest Test MSE of 1.039. This value is lower than all other models considered, including the unpruned regression tree (1.701), the pruned regression tree (1.607), random forest (1.041), and boosting (1.041). Therefore, the bagging is the best model for predicting the amount of claims in this study, as it demonstrates superior performance on unseen data.

e) Compare the best models in your midterm and (d). Is a tree-based method better?

Because the due date for this assignment is before the due date for the Midterm, I have been unable to complete the section of the Midterm that is required for me to answer this question. Because I am at JSM, I have not had time to complete the midterm question before this is due.

2. Consider Cloud-to-ground (CG) lightning flash detection with radar-based predictors (`Storm.RData`).

The primary objective of the study is to develop a radar-based algorithm or model for predicting the potential CG lightning flash. Use storm. train and storm.test for training and validation, respectively. The response variable is storm indicating CG lightning flash, and the other variables measured from operational C-band Doppler weather radar are predictors.

a) Fit a classification tree to the training set. Plot the tree, and interpret the results. What test MSE do you obtain? How many terminal nodes does the tree have?

A classification tree was fitted to the `storm.train` dataset to predict the occurrence of Cloud-to-Ground (CG) lightning (`storm`) based on radar measurements. The resulting tree is evaluated on the `storm.test` dataset.

Tree Plot

The classification tree generated from the training data is shown in Figure 5. Each node displays the predicted class (`no"  or`yes"), the probability of a "yes", and the percentage of observations from the training set that fall into that node.

Interpretation

The final tree consists of 5 terminal nodes (or leaves). The interpretation of the splits is as follows:

- The initial and most significant split is on the predictor `MAX_VIL`. If a storm's Maximum Vertically Integrated Liquid (`MAX_VIL`) is less than 9.5, it is classified as not producing a CG flash ("no"). This single rule correctly classifies a large portion (85%) of the observations.
- For storms with `MAX_VIL` greater than or equal to 9.5, the model considers the storm's `VOLUME`.
- If the `VOLUME` is less than 195, a further split is made on `MAX_UVIL`. A `MAX_UVIL` less than 1.7 leads to a "no" prediction, while a value greater than or equal to 1.7 leads to a "yes" prediction.
- If the `VOLUME` is greater than or equal to 195, the final split is determined by the `AXIS_RATIO`. An `AXIS_RATIO` of 2.5 or more leads to a "yes" prediction with a high probability (0.79). This suggests that for large, intense storms, those that are more elongated are more likely to produce CG lightning.

In summary, the model indicates that storms with high `MAX_VIL` and large `VOLUME` are the most likely to produce CG lightning, with `AXIS_RATIO` and `MAX_UVIL` acting as further refining criteria.

Test Performance and Tree Complexity

When the model is applied to the test set:

- The test Mean Squared Error (MSE), calculated as the Brier score from the predicted probabilities against the actual outcomes, is **0.1337**.

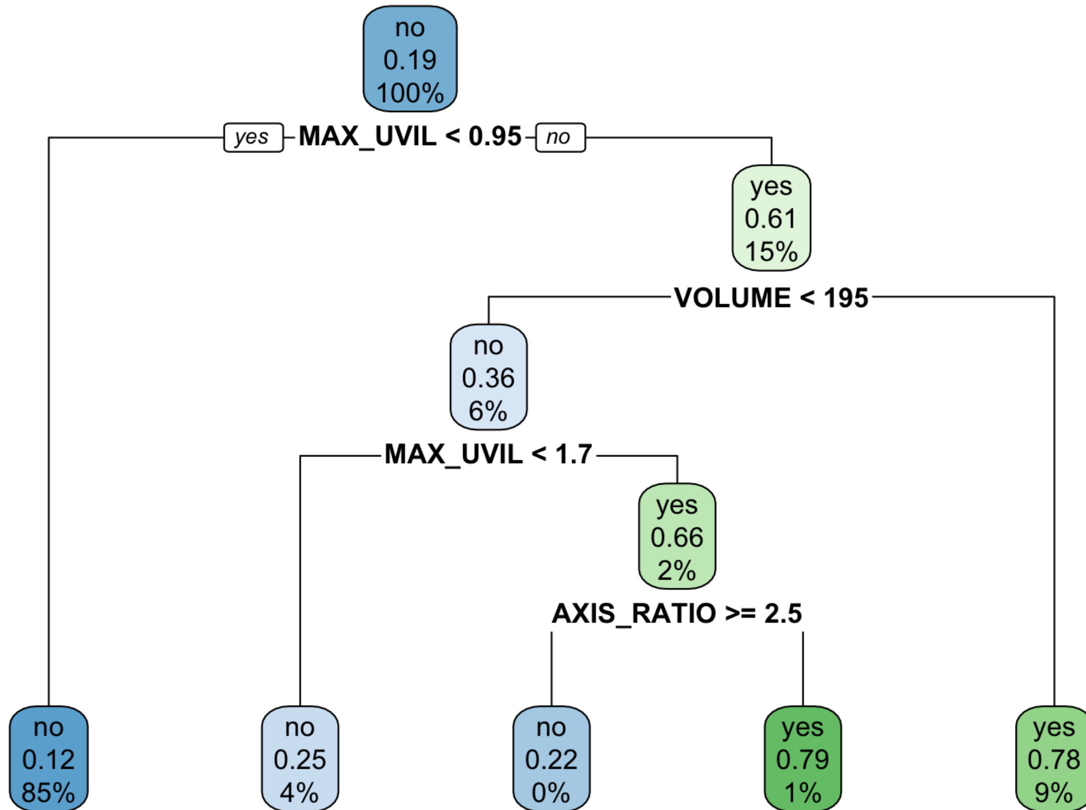

Figure 5: Classification decision tree on all of the training data.

- The test misclassification rate is **16.29%**, based on a confusion matrix of $\begin{pmatrix} 441 & 74 \\ 19 & 37 \end{pmatrix}$.
- The tree has **5** terminal nodes.

b) Use cross-validation in order to determine the optimal level of tree complexity. Does pruning the tree improve the test misclassification rate?

To determine the optimal level of tree complexity, 10-fold cross-validation was performed on the training data. The model's performance, measured by the ROC AUC metric, was evaluated across a range of cost-complexity (cp) values, as shown in Figure 6.

The cross-validation process identified an optimal cost-complexity parameter of cp = 0.00122. This value maximizes the cross-validated ROC AUC, balancing the model's bias and variance.

A final tree was then trained on the entire training set using this optimal cp value. This "pruned" model

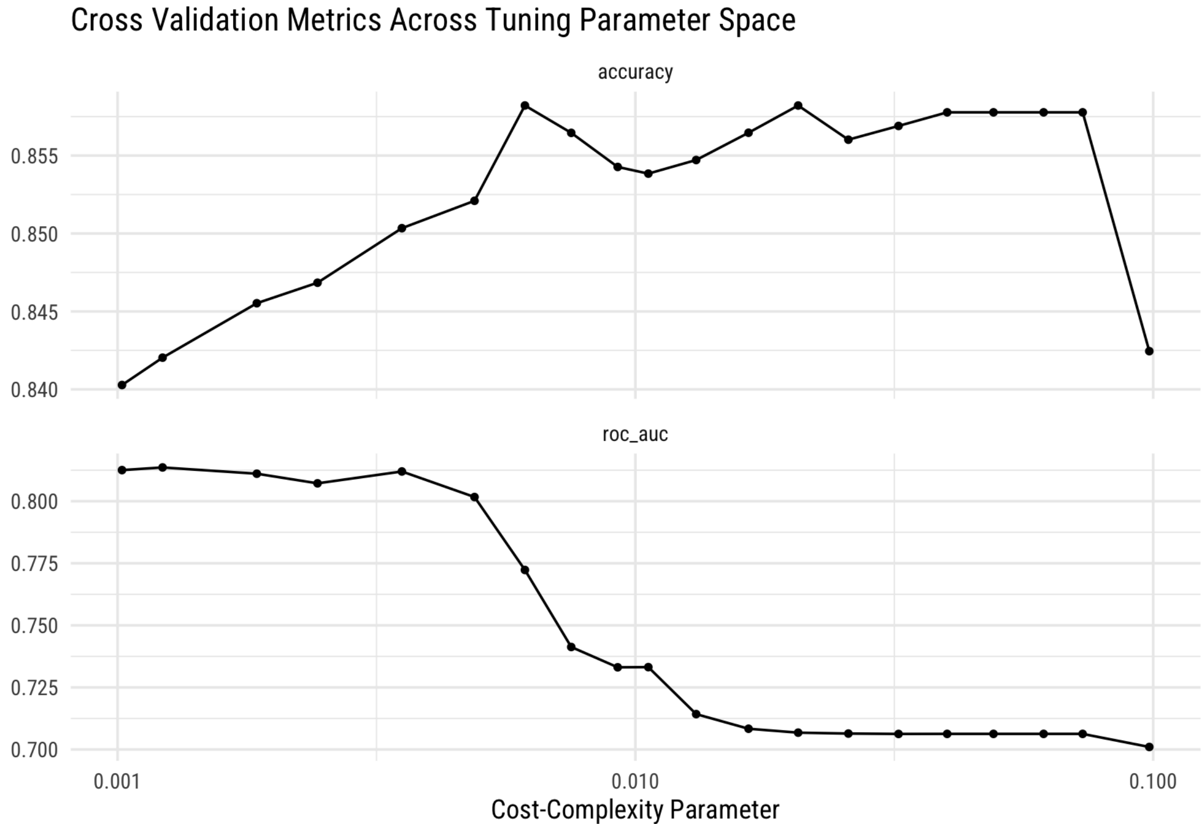

Figure 6: Model metrics across the cost complexity parameter space.

was evaluated on the test set, resulting in a test misclassification rate of 0.1716.

To answer the main question: Yes, pruning the tree improves the test misclassification rate.

The unpruned tree had a misclassification rate of 0.1751, while the optimally pruned tree achieved a rate of 0.1716. This represents a small but clear improvement of 0.0035. By slightly simplifying the model, pruning reduced overfitting to the training data and improved the model's ability to generalize to new, unseen data. The resulting pruned tree is shown in the "Final Pruned Classification Tree" plot.

c) Use the ensemble approaches (bagging, random forest, and boosting) in order to analyze this data. Tune them appropriately, calculate test misclassification rate and determine which variables are most important.

To improve upon the single classification tree, we analyzed the data using three powerful ensemble approaches: bagging, random forest, and boosting (with the XGBoost implementation). For each method, we performed hyperparameter tuning to optimize performance before evaluating the final model on the test set.

Hyperparameter Tuning

We tuned each ensemble model using 10-fold cross-validation on the training data. To efficiently search the parameter space, we employed a space-filling design (`grid_space_filling`) to generate 20 candidate hyperparameter combinations for each model. The combination that yielded the highest mean ROC AUC across the cross-validation folds was selected as the optimal set of parameters for the final model.

Bagging Results

For the bagging model, we tuned the tree depth, the minimum number of observations in a node, and the class cost. The optimal hyperparameters found are detailed in Table 4. The final model, trained with these parameters, achieved a test misclassification rate of **0.1419**. The confusion matrix for the test set is shown in Table 5.

Table 4: Optimal Hyperparameters for Bagging

Parameter	Value
Tree Depth	10
Min Node Size (min_n)	7
Class Cost	1.0

Table 5: Bagging Confusion Matrix (Test Data)

		Actual	
		no	**yes**
Predicted	**no**	441	62
	yes	19	49

Random Forest Results

For the random forest model, we tuned the number of predictors to randomly sample at each split (`mtry`) and the minimum node size (`min_n`). The optimal values are shown in Table 6. This model resulted in a test misclassification rate of **0.1471**. Its confusion matrix is presented in Table 7.

Table 6: Optimal Hyperparameters for Random Forest

Parameter	Value
`mtry`	9
Min Node Size (min_n)	11

Boosting (XGBoost) Results

For the boosting model, we tuned the tree depth, minimum node size, and the learning rate. The optimal hyperparameters are listed in Table 8. The final boosted model yielded a test misclassification rate of

Table 7: Random Forest Confusion Matrix (Test Data)

		Actual	
		no	**yes**
Predicted	**no**	441	65
	yes	19	46

0.1524, and its confusion matrix is shown in Table 9.

Table 8: Optimal Hyperparameters for XGBoost

Parameter	Value
Tree Depth	2
Min Node Size (min_n)	6
Learning Rate	0.0234

Table 9: XGBoost Confusion Matrix (Test Data)

		Actual	
		no	**yes**
Predicted	**no**	437	64
	yes	23	47

Variable Importance

To determine which variables were most influential in the predictions, we generated variable importance plots for the random forest and XGBoost models, shown in Figure 7. Across both models, a consistent set of predictors emerged as the most important. The variables `MAX_VII`, `MAX_UVIL`, and VOLUME were consistently ranked at or near the top, indicating their strong predictive power for CG lightning flashes. Other variables like AREA, `MAX_Z`, and `MAX_VIL` also showed significant contributions.

d) Report the best model in terms of misclassification rate.

Table 10: Comparison of Model Performance for Storm Prediction

Model	Test Misclassification Rate
Unpruned Classification Tree	0.1751
Pruned Classification Tree	0.1716
Bagging	0.1419
Random Forest	0.1471
Boosting (XGBoost)	0.1524

To determine the best model in terms of misclassification rate, the performance of the single classification trees (both unpruned and pruned) and the three ensemble methods were compared. The unpruned classification tree had a test misclassification rate of 0.1751, which improved to 0.1716 after pruning. All three ensemble methods outperformed the single trees. The bagging model achieved a test misclassification rate

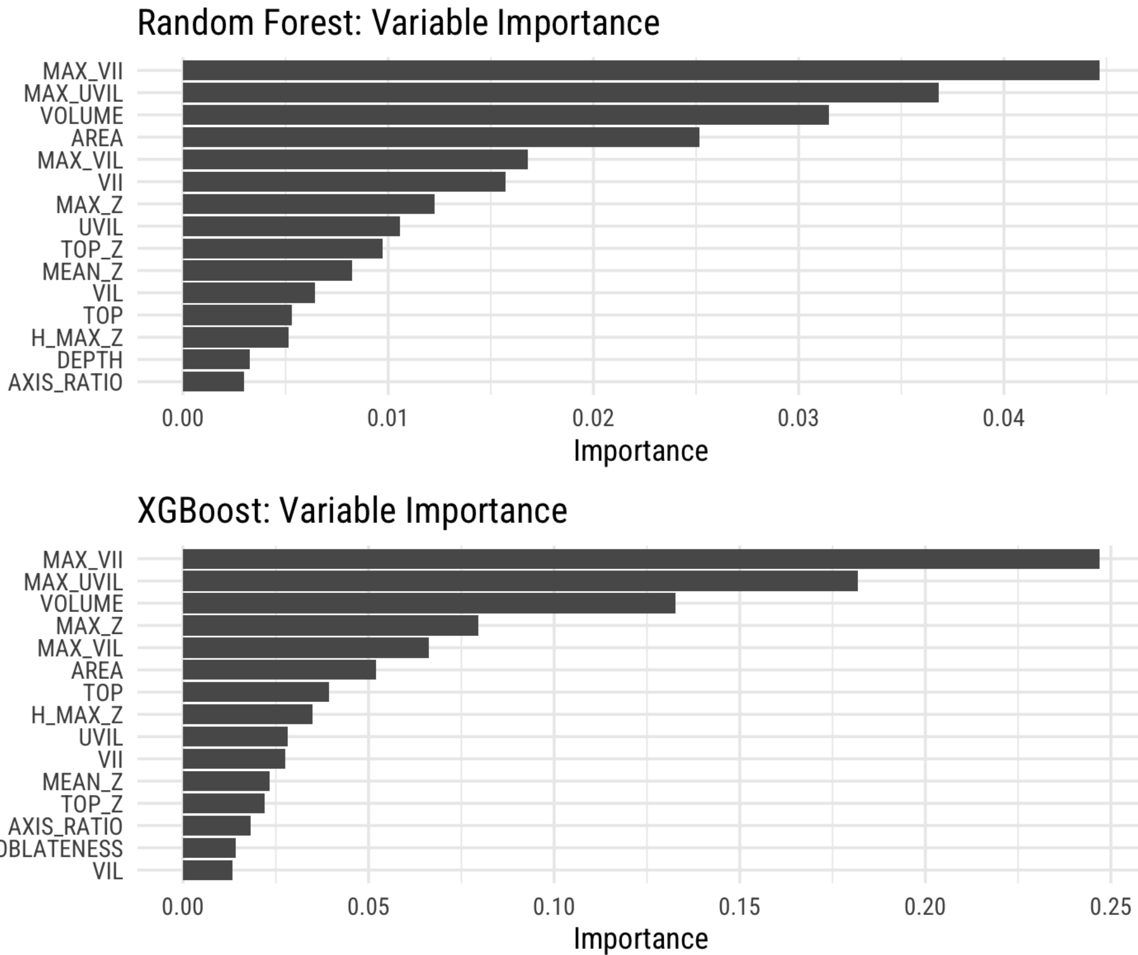

Figure 7: Variable importance plots for two of the three models. For some reason, the computational package we used to fit the bagging model was not supported by the variable importance plot package we were using.

of 0.1419 , the random forest model had a rate of 0.1471 , and the boosting model yielded a rate of 0.1524. Based on these results, the bagging model is the best model, as it achieved the lowest test misclassification rate of 0.1419. Table 10 shows all of the metrics.

e) Compare the best models in your midterm and d). Is a tree-based method better?

When comparing the best models from our midterm and our homework analysis for the storm prediction dataset, the bagging model from the homework demonstrates superior performance based on the accuracy metric. The best model from the midterm in terms of raw accuracy was the `Baseline_Logistic` model, which achieved a test accuracy of 0.8424, corresponding to a misclassification rate of 0.1576. In contrast, the bagging model from the homework assignment, which was the top performer among the tree-based methods in that analysis, achieved a test misclassification rate of 0.1419. This translates to an accuracy of 0.8581, which is higher than the midterm's `Baseline_Logistic` model. While the

bagging model is better on this metric, the midterm analysis noted a trade-off, ultimately selecting the XGB_ROSE model because preprocessing techniques improved sensitivity and ROC AUC, which are considered more important evaluation criteria due to the severe class imbalance of the dataset.

Code Appendix

```
knitr::opts_chunk$set(dev = "cairo_pdf",
                      fig.width = 5,
                      fig.height = 3.25,
                      fig.align = 'center',
                      echo = FALSE,
                      message = FALSE,
                      warning = TRUE,
                      error = FALSE,
                      cache = FALSE)
library("tidymodels")
library("patchwork")
library("glue")
library("scales")
library("extrafont")
library("tinytex")
library("patchwork")
library("gridExtra")
library("tidyr")
library("latex2exp")
library("GGally")
library("leaps")
library("broom")
library("pls")
library("knitr")
library("rprojroot")
library("rpart")
library("rpart.plot")
theme_set(theme_minimal(base_family = "Roboto Condensed"))

conflicted::conflicts_prefer(
  readr::col_factor(),
  purrr::discard(),
  rstan::extract(),
  dplyr::lag(),
  rstan::traceplot(),
  viridis::viridis_pal(),
  readr::parse_date
)
load(here::here("Midterm", "telematics.RData"))
tm <- dat_1
rm(dat_1)
tm <- tm |>
  mutate(log_AMT_Claim = log(AMT_Claim))
```

```
# 1a ---------------------------------------------------------------------
# Partition the data into training and testing sets
set.seed(123) # for reproducibility
train_indices <- sample(1:nrow(tm), size = 0.7 * nrow(tm))
train_set <- tm[train_indices, ]
test_set <- tm[-train_indices, ]

# Fit the regression tree model
tree_model <- rpart(log_AMT_Claim ~ ., data = train_set |>
  select(-AMT_Claim), method = "anova")

# Plot the tree
rpart.plot(tree_model)

# Make predictions on the test set
predictions <- predict(tree_model, newdata = test_set |>
  select(-AMT_Claim))

# Calculate the Test MSE
test_mse <- mean((predictions - test_set$log_AMT_Claim)2)

# Print the test MSE
# print(paste("Test MSE:", test_mse))


# 1b ---------------------------------------------------------------------
# For reproducibility
set.seed(123)

unpruned_mse <- test_mse

# Load the data
# Assuming 'tm' is already loaded from the telematics.RData file
# The following code assumes a full 'tm' data frame is available with all 52

# Create training and testing splits
tm_split <- initial_split(tm, prop = 0.7, strata = log_AMT_Claim)
tm_train <- training(tm_split)
tm_test <- testing(tm_split)

# Define 10-fold cross-validation
tm_folds <- vfold_cv(tm_train, v = 10, strata = log_AMT_Claim)

# Define the regression tree model specification, making complexity parameter
tree_spec <- decision_tree(cost_complexity = tune()) |>
  set_engine("rpart") |>
  set_mode("regression")
```

```
# Create a workflow to combine the model and formula
tree_workflow <- workflow() |>
  add_model(tree_spec) |>
  add_formula(log_AMT_Claim ~ .)

cost_param <- cost_complexity(range = c(-5, -0.1), trans = log10_trans())

# Create a space-filling grid with 20 points
tree_grid <- grid_space_filling(cost_param, size = 30)

# Alternatively, use dials to create the grid
# tree_grid <- grid_regular(cost_complexity(), levels = 5)

# Tune the model using cross-validation
tree_results <- tune_grid(
  tree_workflow,
  resamples = tm_folds,
  grid = tree_grid,
  metrics = metric_set(rmse)
)

# Analyze the results
# Display the best performing models based on RMSE (the square root of MSE)
tree_results |>
  show_best(metric = "rmse")

# Plot the results to visualize the relationship between cost_complexity and p
autoplot(tree_results) + ggtitle("Tuning Results for Cost Complexity for Pruni
# Select the optimal cost_complexity parameter that minimizes RMSE
best_cp <- tree_results |>
  select_best(metric = "rmse")

# Finalize the workflow with the best parameter
final_tree_workflow <- finalize_workflow(tree_workflow, best_cp)

# Fit the final, pruned model to the full training data
final_tree_fit <- fit(final_tree_workflow, data = tm_train)

# Make predictions on the test data
final_test_pred <- predict(final_tree_fit, new_data = tm_test)

# Calculate the test MSE for the pruned tree
test_mse_pruned <- mean((final_test_pred$.pred - tm_test$log_AMT_Claim)2)

#
# cat("Optimal cost_complexity value:", best_cp$cost_complexity, "\n")
cat("Test MSE for the pruned tree:", test_mse_pruned, "\n")
```

```r
cat("Test MSE for the unpruned tree:", unpruned_mse, "\n")


# 1c -------------------------------------------------------
library("doParallel") # For parallel processing during tuning

# --- Data Preparation (assuming 'tm' dataframe is already loaded) ---
# Ensure categorical variables are factors
tm <- tm |>
  mutate(
    Insured.sex = as.factor(Insured.sex),
    Marital = as.factor(Marital),
    Car.use = as.factor(Car.use),
    Region = as.factor(Region)
  )

# For reproducibility
set.seed(123)

# Create training and testing splits
tm_split <- initial_split(tm, prop = 0.7, strata = log_AMT_Claim)
tm_train <- training(tm_split)
tm_test <- testing(tm_split)

# Define 10-fold cross-validation for tuning
tm_folds <- vfold_cv(tm_train, v = 10, strata = log_AMT_Claim)

# Set up parallel processing for faster tuning (optional but recommended)
num_cores <- parallel::detectCores(logical = FALSE) # Use physical cores
registerDoParallel(cores = num_cores)

# Define the bagged tree model specification
# Bagging is essentially a random forest with mtry set to the number of all pr
# and min_n is typically set to 1 for maximum tree growth.
bag_spec <- rand_forest(mtry = ncol(select(tm_train, -log_AMT_Claim)),
                        trees = 1000, # A common number of trees for bagging
                        min_n = 1) |>
  set_engine("ranger", importance = "permutation") |> # ranger for speed, impo
  set_mode("regression")

# Create a workflow for bagging
bag_workflow <- workflow() |>
  add_model(bag_spec) |>
  add_formula(log_AMT_Claim ~ .)

# Fit the bagged model to the training data (no tuning needed for basic baggi
bag_fit <- fit(bag_workflow, data = tm_train)
```

```r
# Make predictions on the test data
bag_predictions <- predict(bag_fit, new_data = tm_test)

# Calculate Test MSE for Bagging
bag_test_mse <- mean((bag_predictions$.pred - tm_test$log_AMT_Claim)2)


# Determine variable importance for Bagging
bag_var_imp <- extract_fit_parsnip(bag_fit)$fit |>
  vip::vip(num_features = 10) # Display top 10 important variables
bag_var_imp <- bag_var_imp + ggtitle("Variable Importance Plot for Bagged Tre
# cat("\n")

# --- 2. Random Forest ---

# Define the Random Forest model specification with tuneable hyperparameters
rf_spec <- rand_forest(mtry = tune(),
                       trees = 1000, # Fixed a high number of trees
                       min_n = tune()) |>
  set_engine("ranger", importance = "permutation") |>
  set_mode("regression")

# Create a workflow for Random Forest
rf_workflow <- workflow() |>
  add_model(rf_spec) |>
  add_formula(log_AMT_Claim ~ .)

# Create a tuning grid using grid_space_filling
# mtry: number of predictors to sample at each split
# min_n: minimum number of data points in a node required for the node to be s
rf_grid <- grid_space_filling(
  mtry(range = c(2, ncol(select(tm_train, -log_AMT_Claim)))), # mtry cannot b
  min_n(),
  size = 20 # Number of combinations to try
)

# Tune the Random Forest model using cross-validation
rf_results <- tune_grid(
  rf_workflow,
  resamples = tm_folds,
  grid = rf_grid,
  metrics = metric_set(rmse),
  control = control_grid(save_pred = TRUE) # Save predictions for potential a
)

# Show best Random Forest models
# cat("Best Random Forest Models:\n")
```

```r
rf_results |>
  show_best(metric = "rmse") |>
  print()

# Select the optimal Random Forest parameters
best_rf_params <- rf_results |>
  select_best(metric = "rmse")

# cat("Optimal Random Forest Parameters:\n")
# print(best_rf_params)

# Finalize the workflow with the best parameters
final_rf_workflow <- finalize_workflow(rf_workflow, best_rf_params)

# Fit the final Random Forest model to the full training data
final_rf_fit <- fit(final_rf_workflow, data = tm_train)

# Make predictions on the test data
rf_predictions <- predict(final_rf_fit, new_data = tm_test)

# Calculate Test MSE for Random Forest
rf_test_mse <- mean((rf_predictions$.pred - tm_test$log_AMT_Claim)2)

# Determine variable importance for Random Forest
rf_var_imp <- pull_workflow_fit(final_rf_fit)$fit |>
  vip::vip(num_features = 10)
rf_var_imp <- rf_var_imp + ggtitle("Variable Importance Plot for Random Fores

# --- 3. Boosting (Gradient Boosting Machine - GBM) ---

# Define the Boosted Tree model specification with tuneable hyperparameters
boost_spec <- boost_tree(
  trees = tune(),         # Number of boosting iterations
  min_n = tune(),         # Minimum number of data points in a node
  tree_depth = tune(),    # Maximum depth of the trees
  learn_rate = tune(),    # Shrinkage parameter
  loss_reduction = tune() # Minimum loss reduction to make a split
) |>
  set_engine("xgboost", importance = "permutation") |> # Using xgboost for bo
  set_mode("regression")

# Create a workflow for Boosting
boost_workflow <- workflow() |>
  add_model(boost_spec) |>
  add_formula(log_AMT_Claim ~ .)

# Create a tuning grid using grid_space_filling
```

```
boost_grid <- grid_space_filling(
  trees(),
  min_n(),
  tree_depth(),
  learn_rate(),
  loss_reduction(),
  size = 20 # Number of combinations to try
)

# Tune the Boosting model using cross-validation
boost_results <- tune_grid(
  boost_workflow,
  resamples = tm_folds,
  grid = boost_grid,
  metrics = metric_set(rmse),
  control = control_grid(save_pred = TRUE)
)

# Show best Boosting models
# cat("Best Boosting Models:\n")
boost_results |>
  show_best(metric = "rmse") |>
  print()

# Select the optimal Boosting parameters
best_boost_params <- boost_results |>
  select_best(metric = "rmse")

# cat("Optimal Boosting Parameters:\n")
print(best_boost_params)

# Finalize the workflow with the best parameters
final_boost_workflow <- finalize_workflow(boost_workflow, best_boost_params)

# Fit the final Boosting model to the full training data
final_boost_fit <- fit(final_boost_workflow, data = tm_train)

# Make predictions on the test data
boost_predictions <- predict(final_boost_fit, new_data = tm_test)

# Calculate Test MSE for Boosting
boost_test_mse <- mean((boost_predictions$.pred - tm_test$log_AMT_Claim)2)

# Determine variable importance for Boosting
# Note: xgboost's importance is slightly different, often based on gain or fre
boost_var_imp <- pull_workflow_fit(final_boost_fit)$fit |>
  vip::vip(num_features = 10)
```

```
boost_var_imp <- boost_var_imp + ggtitle("Variable Importance Plot for XGBoos

# --- Summary of Test MSEs ---
cat("\n--- Summary of Ensemble Model Test MSEs ---\n")
cat("Bagging Test MSE:", bag_test_mse, "\n")
cat("Random Forest Test MSE:", rf_test_mse, "\n")
cat("Boosting Test MSE:", boost_test_mse, "\n")

bag_var_imp / rf_var_imp / boost_var_imp

# Stop parallel processing
stopImplicitCluster()


# 2 ---------------------------------------------------
load(here::here("Midterm", "Storm.RData"))
load(here::here("Midterm", "telematics.RData"))

tm <- dat_1
train_data <- storm.train |>
  mutate(storm = factor(storm))
test_data <- storm.test |>
  mutate(storm = factor(storm))

rm(dat_1)
rm(storm.train)
rm(storm.test)


# 2a ---------------------------------------------------
set.seed(123) # for reproducibility
tree_model <- rpart(storm ~ ., data = train_data, method = "class")

# 2. Display the tree summary and find the number of terminal nodes.
# The summary provides details about the splits.
# printcp(tree_model) # Displays the complexity parameter table
# summary(tree_model) # Provides a detailed summary of the splits

# To find the number of terminal nodes (leaves) in an rpart tree:
num_terminal_nodes <- sum(tree_model$frame$var == "<leaf>")
cat("\nNumber of terminal nodes:", num_terminal_nodes, "\n")


# 3. Plot the tree using rpart.plot for better visualization.
# This function creates a much more readable plot than the base plot method.
# It shows the predicted class, the predicted probability, and the percentage
# of observations in each node.
```

```
rpart.plot(tree_model, main = "Classification Tree for Storm Prediction")


tree_pred_class <- predict(tree_model, newdata = test_data, type = "class")

# Create a confusion matrix to see the prediction performance.
confusion_matrix <- table(Predicted = tree_pred_class, Actual = test_data$sto
cat("\nConfusion Matrix on Test Data:\n")
print(confusion_matrix)

# Calculate the misclassification rate from the confusion matrix.
# It's (False Positives + False Negatives) / Total Observations.
misclassification_rate <- 1 - sum(diag(confusion_matrix)) / sum(confusion_mat
cat("\nTest Misclassification Rate:", round(misclassification_rate, 4), "\n")

# If you literally need to calculate MSE, you can compute it on the predicted
# probabilities against a numeric 0/1 version of the response.
# This is less common for classification evaluation but is shown here for comp
y_true_numeric <- as.numeric(test_data$storm) - 1 # Converts "no" to 0, "yes"
tree_pred_prob <- predict(tree_model, newdata = test_data, type = "prob")[, "y
test_mse <- mean((tree_pred_prob - y_true_numeric)2)
cat("Brier Score (based on probabilities):", round(test_mse, 4), "\n")



# 2b -------------------------------------------------------
set.seed(123) # For reproducibility

# --- Part 2b: Cross-Validation to Determine Optimal Complexity ---

# 1. Define model specification for a tunable decision tree
tree_spec <- decision_tree(
  cost_complexity = tune()
) |>
  set_engine("rpart") |>
  set_mode("classification")

# 2. Create 10-fold cross-validation resamples
cv_folds <- vfold_cv(train_data, v = 10, strata = storm)

# 3. Define a grid of cost_complexity (cp) values to test
cp_grid <- grid_space_filling(cost_complexity(range = c(-3, -1)), size = 20)

# 4. Bundle model and formula into a workflow
tree_wf <- workflow() |>
  add_model(tree_spec) |>
  add_formula(storm ~ .)
```

```
# 5. Tune hyperparameters using the CV folds and grid
tree_tune_res <- tune_grid(
  tree_wf,
  resamples = cv_folds,
  grid = cp_grid,
  metrics = metric_set(accuracy, roc_auc)
)

# 6. Visualize and select the best hyperparameter
autoplot(tree_tune_res) +
  ggtitle("Cross Validation Metrics Across Tuning Parameter Space")
best_cp <- select_best(tree_tune_res, metric = "roc_auc")
cat("\nBest Cost Complexity (cp) found via CV:", best_cp$cost_complexity, "\n

# 7. Finalize the workflow with the best cp value
final_wf <- finalize_workflow(tree_wf, best_cp)

# 8. Fit the final pruned model and evaluate on test data
final_model <- fit(final_wf, data = train_data)
pruned_pred_class <- predict(final_model, new_data = test_data, type = "class"
pruned_confusion_matrix <- table(Predicted = pruned_pred_class$.pred_class, Ac
pruned_misclassification_rate <- 1 - sum(diag(pruned_confusion_matrix)) / sum

cat("\nConfusion Matrix for Pruned Tree on Test Data:\n")
print(pruned_confusion_matrix)
cat("\nTest Misclassification Rate (Pruned Tree):", round(pruned_misclassific

# 9. Compare with the unpruned tree's performance
unpruned_model <- rpart(storm ~ ., data = train_data, method = "class", cp = 0
unpruned_pred <- predict(unpruned_model, newdata = test_data, type = "class")
unpruned_cm <- table(unpruned_pred, test_data$storm)
unpruned_misclassification_rate <- 1 - sum(diag(unpruned_cm)) / sum(unpruned_
cat("Test Misclassification Rate (Unpruned Tree):", round(unpruned_misclassif

# Conclusion
improvement <- unpruned_misclassification_rate - pruned_misclassification_rate
cat(sprintf(
  "\nDoes pruning improve the test misclassification rate? %s.",
  ifelse(improvement > 0, "Yes", "No")
))
cat(sprintf(
  "\nThe misclassification rate improved by %.4f (from %.4f to %.4f).\n",
  improvement, unpruned_misclassification_rate, pruned_misclassification_rate
))

# Plot the final pruned tree
final_tree_fit <- extract_fit_engine(final_model)
```

```r
rpart.plot(final_tree_fit, main = "Final Pruned Classification Tree")

library("baguette")
library("vip")

set.seed(123)

# --- Common Setup ---
cv_folds <- vfold_cv(train_data, v = 10, strata = storm)

base_wf <- workflow() |>
  add_formula(storm ~ .)

metric <- metric_set(accuracy, roc_auc)

# --- 1. Bagging ---
bag_spec <- bag_tree(
  tree_depth = tune(),
  min_n = tune(),
  class_cost = tune()
) |>
  set_engine("rpart", times = 50) |>
  set_mode("classification")

bag_grid <- grid_space_filling(
  tree_depth(range = c(5, 15)),
  min_n(range = c(2, 20)),
  class_cost(range = c(1, 2.5)),
  size = 20
)

bag_tune_res <- tune_grid(
  object = base_wf |> add_model(bag_spec),
  resamples = cv_folds,
  grid = bag_grid,
  metrics = metric,
  control = control_grid(save_pred = TRUE)
)

best_bag_params <- select_best(bag_tune_res, metric = "accuracy")
final_bag_wf <- finalize_workflow(base_wf |> add_model(bag_spec), best_bag_pa

final_bag_fit <- fit(final_bag_wf, data = train_data)
bag_preds <- predict(final_bag_fit, new_data = test_data, type = "class")
bag_conf_matrix <- table(Predicted = bag_preds$.pred_class, Actual = test_data
bag_misclassification_rate <- 1 - sum(diag(bag_conf_matrix)) / sum(bag_conf_m
```

```r
cat("\nBagging Model Results:\n")
print(bag_conf_matrix)
cat("Best Bagging Hyperparameters:\n"); print(best_bag_params)
cat("Bagging Test Misclassification Rate:", round(bag_misclassification_rate,

bag_vip_plot <- final_bag_fit |>
  extract_fit_engine() |>
  vip(geom = "col", num_features = 10) +
  labs(title = "Bagging: Variable Importance")

# --- 2. Random Forest ---
cat("\n--- Starting Random Forest Model ---\n")

rf_spec <- rand_forest(
  mtry = tune(),
  min_n = tune(),
  trees = 1000
) |>
  set_engine("ranger", importance = "permutation") |>
  set_mode("classification")

rf_grid <- grid_space_filling(
  mtry(range = c(1, ncol(train_data) - 1)),
  min_n(range = c(2, 20)),
  size = 20
)

rf_tune_res <- tune_grid(
  object = base_wf |> add_model(rf_spec),
  resamples = cv_folds,
  grid = rf_grid,
  metrics = metric,
  control = control_grid(save_pred = TRUE)
)

best_rf_params <- select_best(rf_tune_res, metric = "accuracy")
final_rf_wf <- finalize_workflow(base_wf |> add_model(rf_spec), best_rf_param

final_rf_fit <- fit(final_rf_wf, data = train_data)
rf_preds <- predict(final_rf_fit, new_data = test_data, type = "class")
rf_conf_matrix <- table(Predicted = rf_preds$.pred_class, Actual = test_data$
rf_misclassification_rate <- 1 - sum(diag(rf_conf_matrix)) / sum(rf_conf_matr

cat("\nRandom Forest Model Results:\n")
print(rf_conf_matrix)
cat("Best Random Forest Hyperparameters:\n"); print(best_rf_params)
cat("Random Forest Test Misclassification Rate:", round(rf_misclassification_
```

```
rf_vip_plot <- final_rf_fit |>
  extract_fit_engine() |>
  vip(geom = "col", num_features = 15) +
  labs(title = "Random Forest: Variable Importance")

# --- 3. XGBoost ---
cat("\n--- Starting Boosting (XGBoost) Model ---\n")

xgb_spec <- boost_tree(
  trees = 1000,
  tree_depth = tune(),
  min_n = tune(),
  learn_rate = tune()
) |>
  set_engine("xgboost") |>
  set_mode("classification")

xgb_grid <- grid_space_filling(
  tree_depth(range = c(2, 10)),
  min_n(range = c(2, 25)),
  learn_rate(range = c(-2.5, -1.0)),
  size = 20
)

xgb_tune_res <- tune_grid(
  object = base_wf |> add_model(xgb_spec),
  resamples = cv_folds,
  grid = xgb_grid,
  metrics = metric,
  control = control_grid(save_pred = TRUE)
)

best_xgb_params <- select_best(xgb_tune_res, metric = "accuracy")
final_xgb_wf <- finalize_workflow(base_wf |> add_model(xgb_spec), best_xgb_pa

final_xgb_fit <- fit(final_xgb_wf, data = train_data)
xgb_preds <- predict(final_xgb_fit, new_data = test_data, type = "class")
xgb_conf_matrix <- table(Predicted = xgb_preds$.pred_class, Actual = test_data
xgb_misclassification_rate <- 1 - sum(diag(xgb_conf_matrix)) / sum(xgb_conf_m

cat("\nBoosting (XGBoost) Model Results:\n")
print(xgb_conf_matrix)
cat("Best XGBoost Hyperparameters:\n"); print(best_xgb_params)
cat("XGBoost Test Misclassification Rate:", round(xgb_misclassification_rate,

xgb_vip_plot <- final_xgb_fit |>
  extract_fit_engine() |>
```

```r
  vip(geom = "col", num_features = 15) +
  labs(title = "XGBoost: Variable Importance")

# --- Combine All Variable Importance Plots ---
combined_vip_plot <- rf_vip_plot / xgb_vip_plot +
  plot_annotation(title = "Variable Importance Across Models",
                  theme = theme(plot.title = element_text(hjust = 0.5, size =

print(combined_vip_plot)
```