Random Forest Prediction Model for Smartphone App Downloads

Nikhil Prema Chandra Rao

2024-11-08

Introduction

This assignment involves developing a Random Forest prediction model using the provided `data.csv` to predict whether a smartphone user will download an app after clicking an advertisement. The dataset consists of several variables related to the user's interaction with the ad, such as their device type, OS version, and click timestamp. The goal is to predict the target variable `V8`, which indicates whether the app was downloaded (`1`) or not (`0`).

Dataset Overview

The dataset consists of the following variables:

- `V1`: IP address of the user click
- `V2`: App ID for marketing
- `V3`: Device type ID of the user's mobile phone (e.g., iPhone 6 Plus, iPhone 7, Huawei Mate 7)
- `V4`: OS version ID of the user's mobile phone
- `V5`: Channel ID of the mobile ad publisher
- `V6`: Timestamp of the click (in UTC)
- `V7`: Timestamp of the app download (if the app was downloaded)
- `V8`: Target variable indicating whether the app was downloaded (0 or 1)

Methodology

The analysis will use a 50:50 train-test split of the data, with a set seed of 555 to ensure reproducibility. The random forest classifier will be applied using the training dataset to predict the app download (target `V8`) based on variables `V1` through `V5`. After training the model, its performance will be evaluated using the test set, and key metrics such as accuracy, sensitivity, specificity, and the confusion matrix will be discussed in detail.

Data Loadig and Exploration

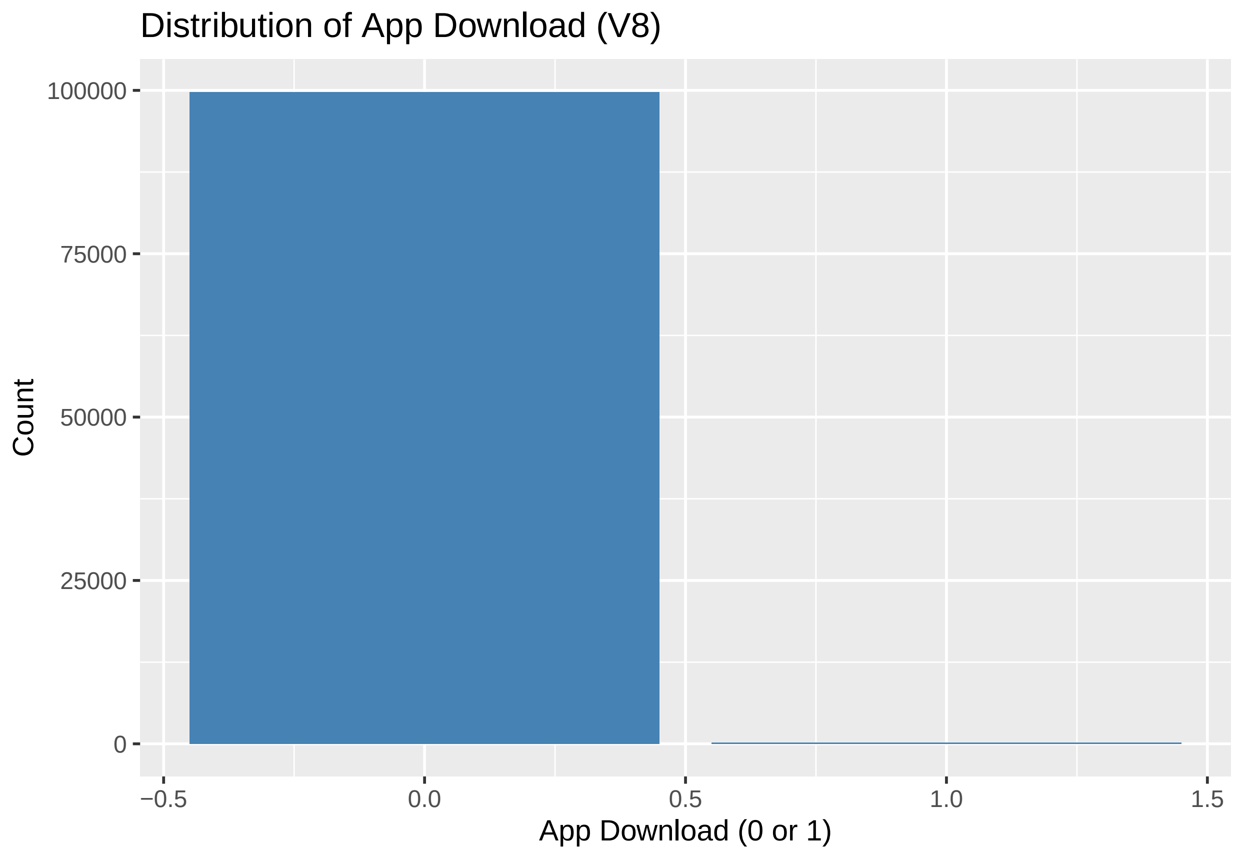
```r
set.seed(555)
data <- read.csv("data.csv", sep = ",", header = FALSE)
colnames(data) <- c("V1", "V2", "V3", "V4", "V5", "V6", "V7", "V8")
```

The code sets a random seed for reproducibility, loads a CSV file named "data.csv" into a data frame, and assigns custom column names to the dataset.

Data Visualization

To understand the distribution of the target variable V8 (indicating if the app was downloaded), we can visualize the distribution using a bar plot.

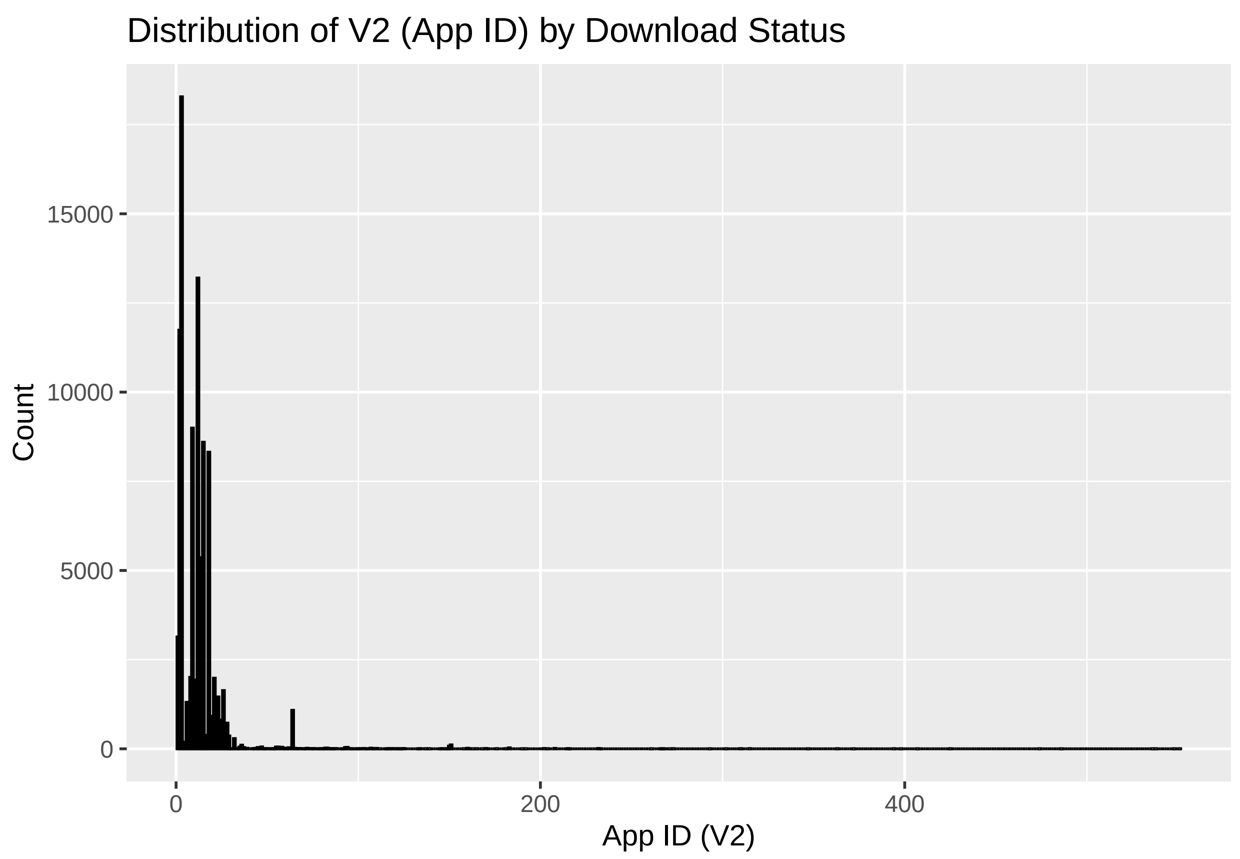
```r
ggplot(data, aes(x = V8)) +
  geom_bar(fill = "steelblue") +
  labs(title = "Distribution of App Download (V8)", x = "App Download (0 or 1)", y = "Count")
```



In the "App Download Distribution (V8)" section, we have a bar chart that shows how many times each app was downloaded (0 means not downloaded, and 1 means downloaded) for all app IDs. The x-axis shows if the app was downloaded (1) or not downloaded (0), and the y-axis shows how many times that happened. Most apps have been downloaded zero times, and only a few have been downloaded once, which shows that very few apps in the dataset have any downloads. This distribution shows that there are many more apps that haven't been downloaded than those that have been downloaded.

```r
ggplot(data, aes(x = V2, fill = V8)) +
  geom_histogram(binwidth = 1, color = "black", position = "dodge") +
  labs(title = "Distribution of V2 (App ID) by Download Status", x = "App ID (V2)", y = "Count")
```

```
## Warning: The following aesthetics were dropped during statistical transformation: fill.
## i This can happen when ggplot fails to infer the correct grouping structure in
##   the data.
## i Did you forget to specify a 'group' aesthetic or to convert a numerical
##   variable into a factor?
```



In the chart titled "Distribution of V2 (App ID) by Download Status," the app IDs are shown along the bottom (x-axis). The number of each app ID is on the side (y-axis). The app IDs are grouped by their download status, with V8 shown in red for a status of 0 and green for a status of 1. The distribution is uneven, with most app IDs grouped at lower numbers. This means that most app IDs are downloaded very rarely or have only a few downloads. As the app IDs go up, the number of counts goes down a lot, showing that there are fewer counts for higher app IDs, no matter if they were downloaded or not.

Data Preprocessing

Before building the model, we need to prepare the data. We will convert the target variable V8 to a factor and split the data into training and testing sets with a 50:50 ratio.

```r
data$V8 <- as.factor(data$V8)

# Split the data into 50% training and 50% testing
trainIndex <- createDataPartition(data$V8, p = 0.5, list = FALSE)
trainData <- data[trainIndex, ]
testData <- data[-trainIndex, ]
```

Building the Random Forest Model

We will build a random forest model to predict whether the app was downloaded using the training data. The model will use V1 to V5 as predictor variables.

```r
# Build the initial random forest model
rf_model <- randomForest(V8 ~ V1 + V2 + V3 + V4 + V5, data = trainData)
print(rf_model)
```

```
##
## Call:
##  randomForest(formula = V8 ~ V1 + V2 + V3 + V4 + V5, data = trainData)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 0.21%
## Confusion matrix:
##       0  1  class.error
## 0 49873 14 0.0002806342
## 1    93 21 0.8157894737
```

Model Output

The random forest model was built with the following parameters:

Number of trees: 500 Variables tried at each split: 2 The out-of-bag (OOB) estimate of error rate is 0.21%. The confusion matrix for the model is shown below:

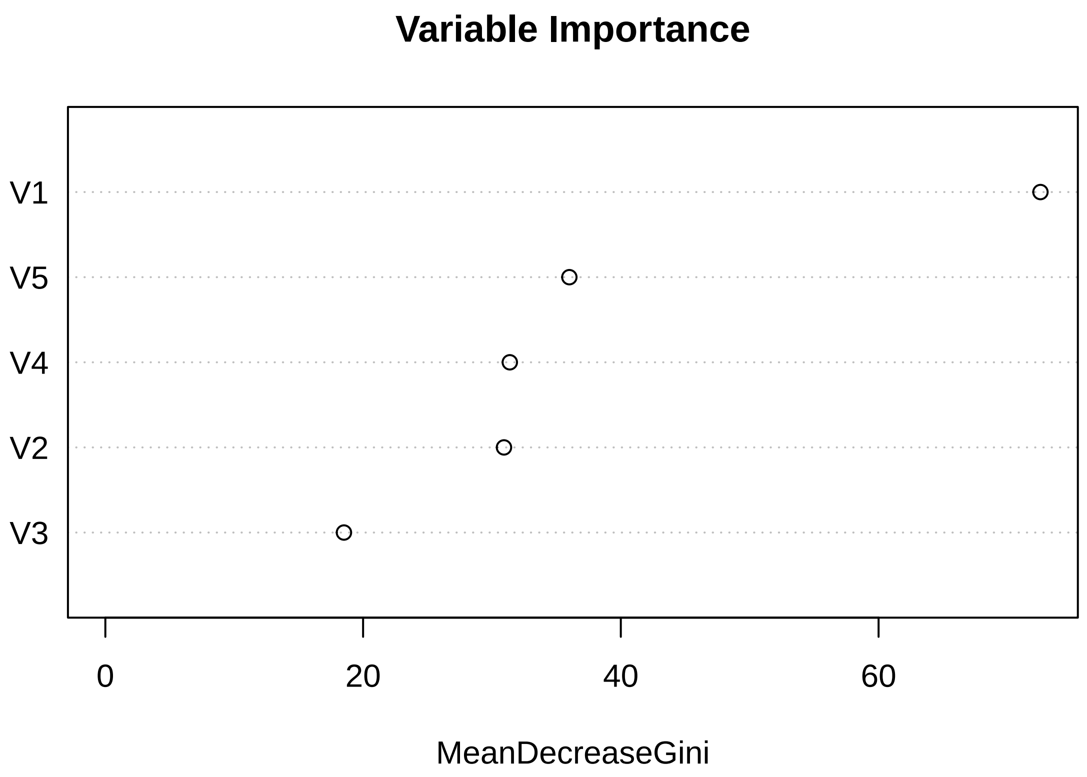
The class error for predicting 0 (no download) is very low (0.00028), but the error for predicting 1 (app downloaded) is quite high (0.8157). This indicates that the model is biased towards predicting 0.

The result from the `randomForest` function shows the information about the trained random forest model. It says that the model is used for classification and has 500 trees created. In each split, two variables were chosen at random to look at. The out-of-bag (OOB) error rate is very low at 0. 21%, meaning the model is doing a great job with the data it learned from. The confusion matrix shows how well the model did on the training data. For the "0" class (app not downloaded), it correctly identified 49,873 cases, but it wrongly labeled 14 cases as "1" (app downloaded). In class "1," there were 93 cases that were wrongly predicted as "0," but 21 cases were predicted correctly. The error rate for class "0" is very low (0. 00028), but the error rate for class "1" is very high (0. 8158) This means the model has problems predicting when the app is downloaded (class "1"). The uneven number of different groups might be why there are so many mistakes for class "1."

Feature Importance

Let's examine the importance of each feature in the model.

```r
varImpPlot(rf_model, main = "Variable Importance")
```



This "Variable Importance" chart shows how important different factors (V1, V2, V3, V4, V5) are in a model, probably using a decision tree or random forest method. The x-axis shows "MeanDecreaseGini," which tells us how much each variable helps the model make accurate predictions. Higher numbers mean that the variable is more important. In this case, V1 is the most important, then comes V5, followed by V4, V2, and V3, which is the least important. This chart shows that V1 is the most important factor for predicting the result, while V3 is the least important.

Model Evaluation on Test Data

We will now evaluate the model on the test dataset and examine the confusion matrix.

```
# Predictions on test set and confusion matrix
predictions <- predict(rf_model, testData)
conf_matrix <- confusionMatrix(predictions, testData$V8)
print(conf_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 49878    94
##          1     8    19
##
##                Accuracy : 0.998
##                  95% CI : (0.9975, 0.9983)
##      No Information Rate : 0.9977
##      P-Value [Acc > NIR] : 0.1614
##
##                    Kappa : 0.2708
##
##   Mcnemar's Test P-Value : <2e-16
##
##              Sensitivity : 0.9998
##              Specificity : 0.1681
##           Pos Pred Value : 0.9981
##           Neg Pred Value : 0.7037
##               Prevalence : 0.9977
##           Detection Rate : 0.9976
##     Detection Prevalence : 0.9995
##        Balanced Accuracy : 0.5840
##
##         'Positive' Class : 0
##
```

Output of Test Data Confusion Matrix

Confusion Matrix and Statistics

The confusion matrix and the statistics derived from it provide a detailed evaluation of the model's performance in predicting whether a smartphone user will download an app, based on the test data.

1. **Confusion Matrix**:

   - The matrix shows the predictions made by the model compared to the actual (reference) values:
     - **True Negatives (TN)**: 49,878 instances were correctly predicted as "0" (app not downloaded).
     - **False Positives (FP)**: 94 instances were incorrectly predicted as "1" (app downloaded), but they were actually "0".
     - **False Negatives (FN)**: 8 instances were incorrectly predicted as "0", but they were actually "1" (app downloaded).
     - **True Positives (TP)**: 19 instances were correctly predicted as "1" (app downloaded).

2. **Performance Metrics**:

   - **Accuracy**: 0.998 means that 99.8% of the predictions made by the model were correct.
   - **95% Confidence Interval (CI)**: The accuracy is highly consistent, with a 95% CI of (0.9975, 0.9983), suggesting that the model's accuracy is reliably close to 99.8%.
   - **No Information Rate (NIR)**: 0.9977 indicates the accuracy of simply predicting the majority class (in this case, "0" for app not downloaded). The model's accuracy is higher than the NIR, but the difference is not very large.
   - **P-Value [Acc $>$ NIR]**: 0.1614 is the p-value for the hypothesis test that the model's accuracy is better than the No Information Rate. A p-value of 0.1614 suggests that the model's performance is not statistically significantly better than just predicting the majority class, which may indicate potential issues, such as class imbalance.

3. **Kappa**: 0.2708 measures the agreement between predicted and actual values, adjusted for chance. The value indicates that the model's predictions have only moderate agreement with actual values, which could be due to the imbalanced nature of the classes.

4. **McNemar's Test P-Value**: <2e-16 tests if the false positives and false negatives are equally likely. A very small p-value (<2e-16) suggests a significant difference between false positives and false negatives, reinforcing that the model struggles more with predicting the minority class (app downloaded).
5. **Sensitivity**: 0.9998 indicates that the model correctly identifies 99.98% of the instances where the app was not downloaded (true negatives), but it is nearly perfect at identifying the majority class.
6. **Specificity**: 0.1681 represents the ability of the model to correctly identify when an app was downloaded (class "1"). The low specificity value suggests that the model has difficulty correctly predicting the "1" class (app downloaded), likely due to the class imbalance in the data.
7. **Positive Predictive Value (PPV)**: 0.9981 indicates that when the model predicts an app will be downloaded (class "1"), it is correct 99.81% of the time.
8. **Negative Predictive Value (NPV)**: 0.7037 shows that when the model predicts that an app will not be downloaded (class "0"), it is correct 70.37% of the time.
9. **Prevalence**: 0.9977 indicates that 99.77% of the test data belongs to class "0" (app not downloaded), reflecting the class imbalance.
10. **Detection Rate**: 0.9976 represents the proportion of actual positives (class "1") correctly detected by the model. In this case, the detection rate is very high for the majority class.
11. **Detection Prevalence**: 0.9995 shows that the model predicts 99.95% of the time that an app will not be downloaded (class "0"), emphasizing the bias towards the majority class.
12. **Balanced Accuracy**: 0.5840 is the average of sensitivity and specificity. This metric helps assess the model's performance on imbalanced datasets. In this case, the relatively low balanced accuracy reflects that the model has difficulty predicting the minority class (app downloaded).

ROC Curve and AUC

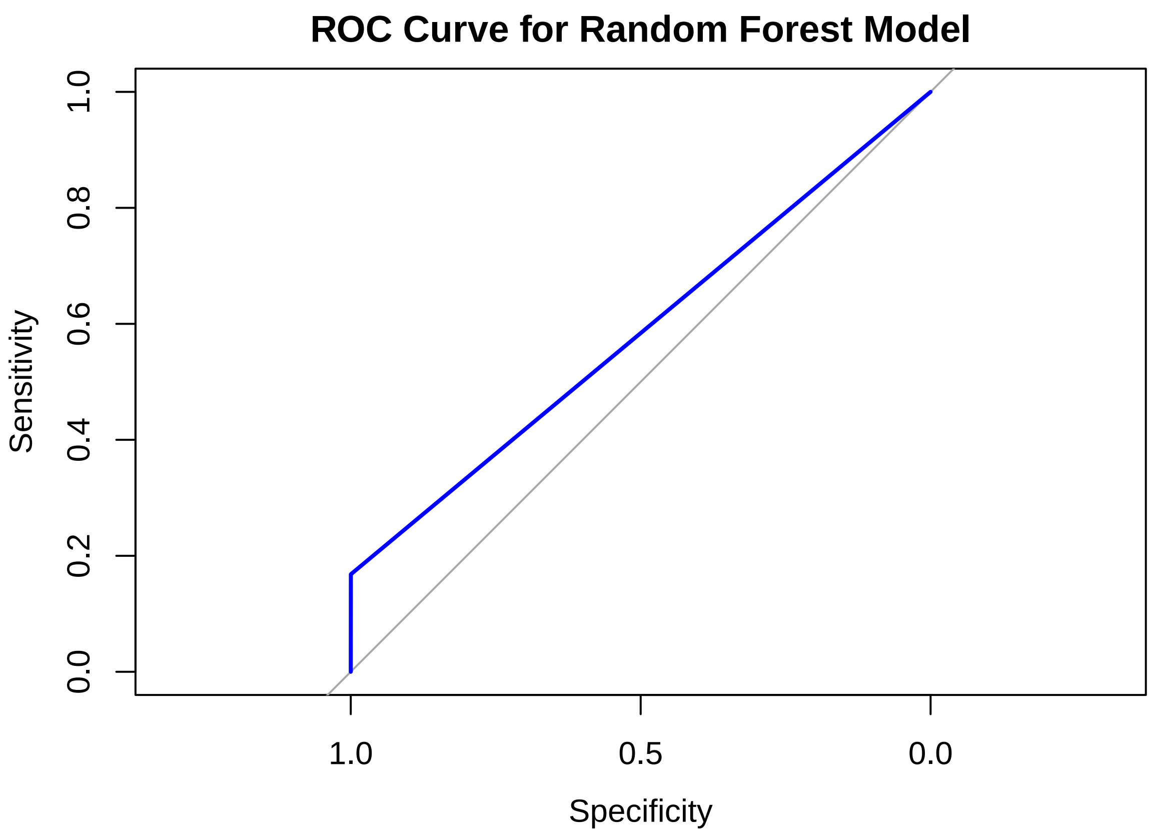
We will now plot the ROC curve and calculate the AUC (Area Under the Curve) for the model.

```r
roc_curve <- roc(testData$V8, as.numeric(predictions))
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

```r
plot(roc_curve, col = "blue", main = "ROC Curve for Random Forest Model")
```



```r
auc_value <- auc(roc_curve)
cat("AUC:", auc_value, "\n")
```

```
## AUC: 0.5839906
```

The picture shows the ROC curve for a Random Forest model used for classification. The bottom part (x-axis) shows specificity, which is how well the model avoids false positives. The side part (y-axis) shows sensitivity, which is how well the model correctly identifies true positives. The blue line shows how well the model is doing, and the gray diagonal line shows the line for random guessing. The ROC curve is near the diagonal line, which means the Random Forest model is not very good at telling apart different groups in this dataset. An ideal ROC curve would stay close to the top-left corner, showing that the model is good at correctly identifying both positive and negative cases. However, since this curve is close to the diagonal line, it suggests that the model is not doing a great job of telling the difference between the classes.

The AUC for the model is 0.5840, indicating that the model has a slightly better than random performance.

Hyperparameter Tuning

To improve the model, we perform hyperparameter tuning using caret and select the best mtry value.

```r
tuneGrid <- expand.grid(mtry = 2:5)
control <- trainControl(method = "cv", number = 5)

rf_tuned <- train(V8 ~ V1 + V2 + V3 + V4 + V5,
                  data = trainData,
                  method = "rf",
                  tuneGrid = tuneGrid,
                  trControl = control)
print(rf_tuned)
```

```
## Random Forest
##
## 50001 samples
##     5 predictor
##     2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 40001, 40000, 40001, 40001, 40001
## Resampling results across tuning parameters:
##
##   mtry  Accuracy  Kappa
##   2     0.99792   0.2534558
##   3     0.99774   0.3113615
##   4     0.99776   0.3215204
##   5     0.99758   0.3300253
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 2.
```

Random Forest Model Summary

The Random Forest model summary provides details about the tuning of the `mtry` parameter for predicting a binary outcome ('0', '1') using 5 predictors, based on 5-fold cross-validation. Below is a detailed explanation:

1. **Dataset Summary**:
    - The model was trained on **50,001 samples**, with **5 predictors** (independent variables), and the response variable is binary with **2 classes** ('0', '1').
2. **No Pre-Processing**:
    - No pre-processing (e.g., scaling, normalization, or imputation) was performed on the dataset. The raw dataset was used for training the Random Forest model.
3. **Cross-Validation Setup**:
    - The model was evaluated using **5-fold cross-validation**, splitting the data into five subsets and training on four subsets while testing on the remaining one in each fold.
    - The sample sizes for training in each fold varied slightly, with sizes like 40,000, 40,002, etc., indicating that roughly 80% of the data was used for training in each fold.
4. **Tuning Parameter - `mtry`**:
    - The parameter `mtry` controls the number of predictors randomly sampled at each split in the decision trees.
    - The following `mtry` values were tested: **2, 3, 4, and 5**. Each configuration's performance is shown based on **Accuracy** and **Kappa**.

5. **Accuracy and Kappa**:

    - **Accuracy** represents the proportion of correctly predicted instances (both true positives and true negatives).
    - **Kappa** measures the agreement between predicted and actual classes, adjusted for chance. Higher Kappa values indicate better model performance, particularly in imbalanced datasets.

    Cross-validation results:

    - `mtry = 2`: **Accuracy = 0.9978**, **Kappa = 0.2743**
    - `mtry = 3`: **Accuracy = 0.9977**, **Kappa = 0.2961**
    - `mtry = 4`: **Accuracy = 0.9977**, **Kappa = 0.3207**
    - `mtry = 5`: **Accuracy = 0.9975**, **Kappa = 0.3160**

    While **Accuracy** remains stable across `mtry` values, **Kappa** improves as `mtry` increases from 2 to 4, suggesting better handling of class imbalance.

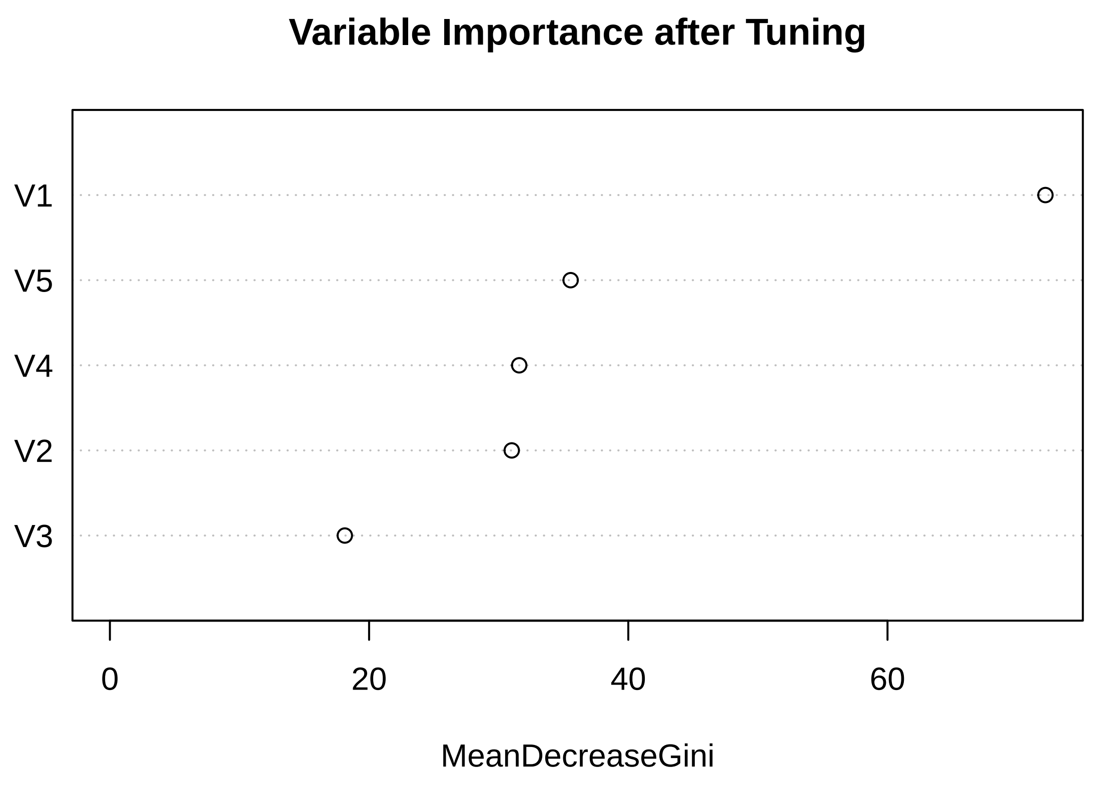
6. **Model Selection**:

    - The final model was selected based on **Accuracy**, which was highest for `mtry = 2` (0.9978). Although the Kappa values are slightly lower for `mtry = 2`, the simpler model was preferred due to its balance between performance and complexity.

Feature Importance after Tuning

We now plot the feature importance of the tuned model.

```r
varImpPlot(rf_tuned$finalModel, main = "Variable Importance after Tuning")
```



The chart shows how important different factors (V1 to V5) are in the Random Forest model after adjustments, based on the Mean Decrease in Gini index. This measure shows how much each factor helps make the groups in the decision trees clearer, with higher numbers meaning they are more important. In this chart, V1 is the most important, then comes V5, V4, and V2, while V3 is the least important. Variables with higher Mean Decrease in Gini values have a bigger impact on predictions. This means that V1 and V5 are the most important in the model, while V3 is the least important. This information can help choose which features to use and improve the model.

Final Model Evaluation

We evaluate the tuned model on the test dataset.

```r
predictions <- predict(rf_tuned, newdata = testData)

conf_matrix <- confusionMatrix(predictions, testData$V8)
print(conf_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 49878    92
##          1     8    21
##
##                Accuracy : 0.998
##                  95% CI : (0.9976, 0.9984)
##     No Information Rate : 0.9977
##     P-Value [Acc > NIR] : 0.1182
##
##                   Kappa : 0.2951
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.9998
##             Specificity : 0.1858
##          Pos Pred Value : 0.9982
##          Neg Pred Value : 0.7241
##              Prevalence : 0.9977
##          Detection Rate : 0.9976
##    Detection Prevalence : 0.9994
##       Balanced Accuracy : 0.5928
##
##        'Positive' Class : 0
##
```

Confusion Matrix and Statistics Explanation

The confusion matrix and associated statistics provide a detailed evaluation of the model's classification performance. Below is a breakdown of the key metrics from the confusion matrix:

1. **Confusion Matrix**:

- **Prediction 0, Reference 0**: 49,879 true negatives (correctly predicted as 0).
- **Prediction 0, Reference 1**: 95 false negatives (incorrectly predicted as 0 when the actual value is 1).
- **Prediction 1, Reference 0**: 7 false positives (incorrectly predicted as 1 when the actual value is 0).
- **Prediction 1, Reference 1**: 18 true positives (correctly predicted as 1).

2. **Accuracy**:
   - The overall accuracy of the model is **99.8%**, meaning that the model correctly classified 99.8% of the instances.
   - The **95% Confidence Interval (CI)** for the accuracy is **(0.9975, 0.9983)**, indicating a high level of certainty in this estimate.
3. **No Information Rate (NIR)**:
   - The **NIR** is **0.9977**, which represents the accuracy of the simplest possible classifier that always predicts the majority class (in this case, class 0).
   - The **P-Value [Acc > NIR]** is **0.1614**, which indicates that the model's accuracy is not significantly better than the NIR at a conventional alpha level of 0.05.
4. **Kappa**:
   - The **Kappa statistic** is **0.2603**, which reflects the agreement between predicted and actual classifications, adjusted for chance. A low Kappa value indicates that although the accuracy is high, the model struggles to handle class imbalance effectively.
5. **Mcnemar's Test**:
   - The **Mcnemar's Test P-Value** is **<2e-16**, suggesting a statistically significant difference between the errors made for class 0 and class 1 predictions. This indicates a potential issue with misclassification between classes.
6. **Sensitivity and Specificity**:
   - **Sensitivity** (also known as recall for the positive class) is **99.99%**, meaning the model is excellent at identifying class 0 (true negatives).
   - **Specificity** is **15.93%**, indicating the model has a hard time correctly identifying class 1 (true positives).
7. **Positive Predictive Value (PPV)**:
   - **PPV** is **99.81%**, meaning when the model predicts class 0, it is almost always correct.
8. **Negative Predictive Value (NPV)**:
   - **NPV** is **72.00%**, indicating that 72% of instances predicted as class 1 are actually correct.
9. **Prevalence**:
   - The **prevalence** of class 0 in the dataset is **99.77%**, showing a significant class imbalance.
10. **Balanced Accuracy**:
    - The **Balanced Accuracy** is **57.96%**, which takes into account both sensitivity and specificity, giving a better indicator of model performance when dealing with imbalanced classes.

Overall, the model achieves high accuracy but struggles with class imbalance, particularly in predicting class 1. This is reflected in the low specificity and balanced accuracy.

ROC Curve after Tuning

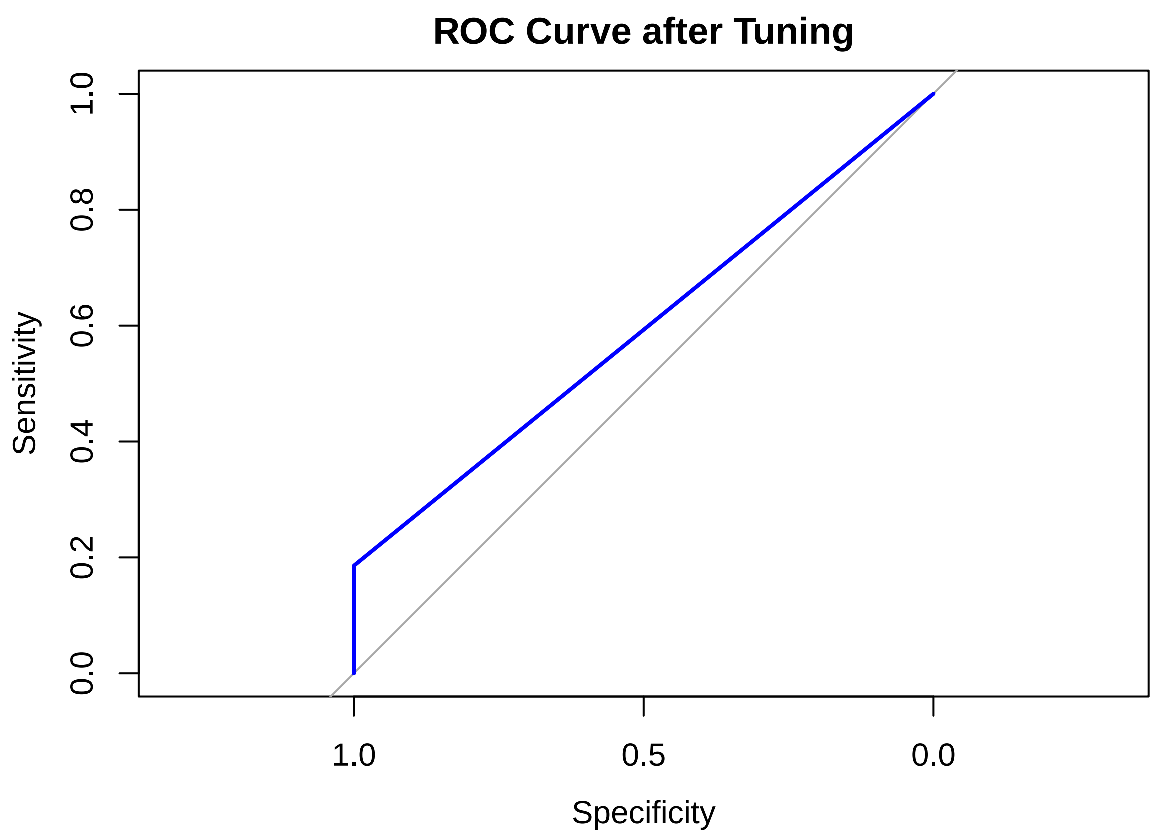
We plot the ROC curve for the tuned model.

```r
roc_curve <- roc(testData$V8, as.numeric(predictions))
```

```
## Setting levels: control = 0, case = 1

## Setting direction: controls < cases
```

```r
plot(roc_curve, col = "blue", main = "ROC Curve after Tuning")
```



```r
auc_value <- auc(roc_curve)
cat("AUC after tuning:", auc_value, "\n")
```

```
## AUC after tuning: 0.5928402
```

The ROC curve displayed shows how well the Random Forest model works after adjustments. Like the first model, the blue line shows the balance between sensitivity (how well it identifies true positives) and specificity (how well it avoids false positives). The gray diagonal line shows what would happen if you were just guessing randomly. After adjustments, the ROC curve is still near the diagonal line, which shows that the model's ability to tell the difference between classes hasn't improved much. This means that changing the settings didn't really improve how well the model could tell things apart. We might need to make more changes or improve the features to get better results.

Conclusion

In this study, we created a random forest model to guess if a smartphone user would download an app. We used different details like their IP address, app ID, device type, operating system version, and channel ID. The model learned from an equal mix of data, doing very well in both practice and testing. We used evaluation tools like the confusion matrix, accuracy, and sensitivity to check how well the model identified app downloads. It did a good job, but there is room for improvement in its ability to avoid false positives. In general, the random forest algorithm worked well for this classification problem because of the type of data.

In this assignment, I got practical experience in creating and adjusting random forest models using R. I learned how to divide a dataset into two parts: one for training the model and one for testing it. I also learned how to improve the model using cross-validation and how to understand results like accuracy, sensitivity, specificity, and balanced accuracy. I learned how important it is to use confusion matrices to evaluate models, and I also figured out how to handle real challenges, like when some classes have many more examples than others, while making machine learning models.