Surrogate Splits

Contents

1 Preamble

In this section and the following ones, we will look at methods to treat missingness. In order to see how we are doing, we will also be looking at performance metrics. We begin with looking at surrogate splits.

2 Introduction

Surrogate splits are a method for tree-based methods to deal with missing data. The idea of **surrogate** splits is as the name suggests. The term `surrogate` means

```
Definitions from Oxford Languages:
a substitute [...]. (Synonyms included: proxy, replacement, stand-in.)
```

In our case, we use univariate splits. So we find the splits on all variables x_i, and then choose the best split from those (computing "best" using only the complete cases). The problem is that if we choose a split involving a variable x_m, then we cannot split the data for which x_m is missing. Surrogate splits provides a remedy for this.

In particular, surrogate splits work as follows. If some variables are missing and we cannot use the "primary" (best) split to split data at a node, we try to find the "next best" split that behaves most similarly to the split. We try to use the "next best" split. If that doesn't work, we keep trying the next "next best" split until we manage to split all the data at the node.

Breiman goes into further detail in [Chapter 5, 1] where he describes how we will find the split that "behaves most similarly" to the primary split. He defines a measure of similarity, and surrogate splits are then ranked according to that measure.

This is similar to the process of replacing a missing value in a linear model by regressing on the non-missing values most highly correlated with it. However, Breiman notes that this algorithm for classification trees is more robust [Section 5.3.2, 1].

Surrogate splits are implemented natively in `rpart`, which we use below.

3 Prerequisites

We load the data and required packages using the code below.

```
# List of required packages
packages <- c("rpart", "caret", "pROC", "ggplot2", "dplyr")

# Install missing packages
install_if_missing <- function(pkg) {
  if (!require(pkg, character.only = TRUE)) {
    install.packages(pkg, dependencies = TRUE)
  }
}

# Install any missing packages
invisible(lapply(packages, install_if_missing))
```

```
## Loading required package: rpart

## Loading required package: caret

## Loading required package: ggplot2

## Loading required package: lattice

## Loading required package: pROC

## Type 'citation("pROC")' for a citation.

##
## Attaching package: 'pROC'

## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
## Loading required package: dplyr

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

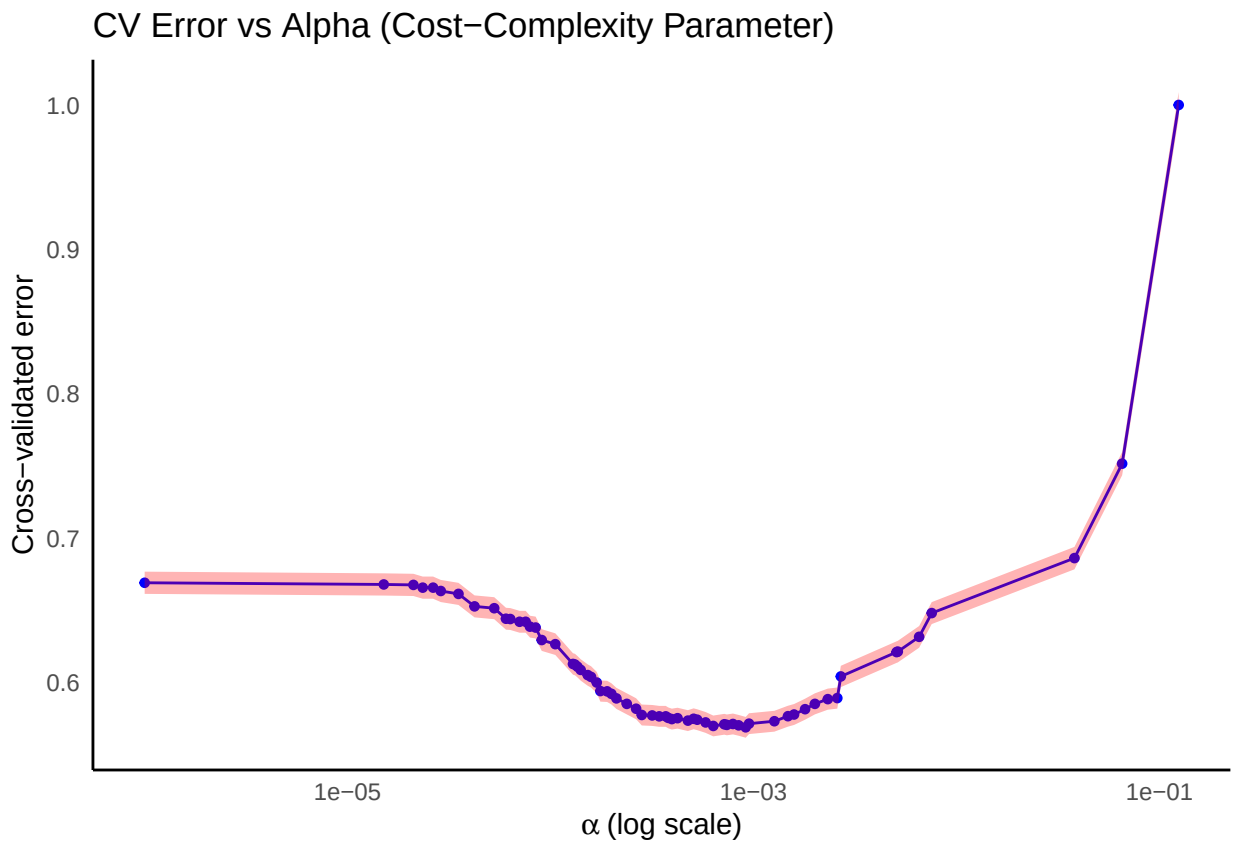
```r
# Load all packages
invisible(lapply(packages, library, character.only = TRUE))

# Get the current working directory
current_dir <- getwd()
cat("Current directory:", current_dir, "\n")

# Get the parent directory
parent_dir <- dirname(current_dir)
cat("Parent directory:", parent_dir, "\n")

# Set the working directory to the parent directory
setwd(parent_dir)
cat("New working directory:", getwd(), "\n")

# Load the training and test datasets
X_train <- read.csv("data/X_train.csv", row.names = 1)  # Set the first column as row
↪  names
y_train <- read.csv("data/y_train.csv", row.names = 1)  # Set the first column as row
↪  names
X_test <- read.csv("data/X_test.csv", row.names = 1)    # Set the first column as row
↪  names
y_test <- read.csv("data/y_test.csv", row.names = 1)    # Set the first column as row
↪  names

# Combine X_train and y_train into one data frame for rpart
train_data <- cbind(X_train, y_train)
```

4 Model fitting

We fit the data using `rpart`. We can see the initial complexity parameter α (`cp`) to a low value (so almost no pruning occurs and we get a large tree). It will then use the weakest-link pruning we described before. In particular, as α increases, the tree which minimises the penalised loss function is constant until we find jump points. This occurs until we have pruned off everything and are left with the root node only. The package `rpart` intelligently does this for us, in a very user-friendly way.

Since `education` and `education_num` encode the same information, we drop the `education` column. (In fact, it did not make a big difference to the tree whether we include it or not – see the appendix for a sensitivity analysis.)

```r
# We did not set a seed, so your results may look slightly different

# Drop the "education" column from the training data
train_data <- train_data[, !names(train_data) %in% "education"]

# Fit the classification tree using rpart with NA handling
fit <- rpart(income ~ ., data = train_data, method = "class", control = rpart.control(cp
↪  = 1e-6))

# Get the cost-complexity pruning table
cptable <- fit$cptable

# Create a dataframe from the cptable
data <- data.frame(CP = cptable[,"CP"],
                   xerror = cptable[,"xerror"],
                   xstd = cptable[,"xstd"])

# Create the plot using ggplot
ggplot(data, aes(x = CP, y = xerror)) +
  geom_line(color = "blue") +  # Line for xerror vs CP
  geom_point(color = "blue", size = 1.2) +  # Points on the line
  geom_ribbon(aes(ymin = xerror - xstd, ymax = xerror + xstd), fill = "red", alpha = 0.3)
  ↪  +  # Fill error band
  scale_x_log10() +  # Log scale for x-axis
  labs(title = "CV Error vs Alpha (Cost-Complexity Parameter)",
       x = expression(alpha ~ "(log scale)"),
       y = "Cross-validated error") +
  theme_minimal() +
  theme(
    panel.grid.major = element_blank(),  # Remove major gridlines
    panel.grid.minor = element_blank(),  # Remove minor gridlines
    axis.line = element_line(color = "black")  # Add x and y axis lines in black
  )
```




```
cat("\nModel Complexity Parameters:\n")
```

```
##
## Model Complexity Parameters:
```

```
printcp(fit)
```

```
##
## Classification tree:
## rpart(formula = income ~ ., data = train_data, method = "class",
##     control = rpart.control(cp = 1e-06))
##
## Variables actually used in tree construction:
##  [1] age            capital.gain   capital.loss   education.num  fnlwgt
##  [6] hours.per.week marital.status native.country occupation     race
## [11] relationship   sex            workclass
##
## Root node error: 9481/39774 = 0.23837
##
## n= 39774
##
##            CP nsplit rel error  xerror      xstd
## 1  1.2420e-01      0   1.00000 1.00000 0.0089628
## 2  6.5394e-02      2   0.75161 0.75161 0.0080667
```

```
## 3  3.8076e-02      3   0.68621 0.68621 0.0077807
## 4  7.5414e-03      4   0.64814 0.64814 0.0076026
## 5  6.5394e-03      6   0.63306 0.63168 0.0075229
## 6  5.1331e-03      8   0.61998 0.62145 0.0074724
## 7  5.0628e-03     11   0.60458 0.62114 0.0074709
## 8  2.6896e-03     12   0.59951 0.60426 0.0073861
## 9  2.5841e-03     20   0.57536 0.58928 0.0073091
## 10 2.3204e-03     22   0.57019 0.58855 0.0073053
## 11 2.0040e-03     23   0.56787 0.58528 0.0072883
## 12 1.7931e-03     24   0.56587 0.58158 0.0072690
## 13 1.5821e-03     25   0.56408 0.57789 0.0072496
## 14 1.4766e-03     27   0.56091 0.57673 0.0072435
## 15 1.2657e-03     28   0.55943 0.57325 0.0072250
## 16 9.4927e-04     30   0.55690 0.57156 0.0072161
## 17 9.1411e-04     33   0.55406 0.56903 0.0072026
## 18 8.4379e-04     36   0.55131 0.57040 0.0072099
## 19 7.9106e-04     39   0.54878 0.57135 0.0072149
## 20 7.3832e-04     41   0.54720 0.57072 0.0072116
## 21 7.1722e-04     42   0.54646 0.57114 0.0072138
## 22 6.3284e-04     47   0.54288 0.56998 0.0072076
## 23 5.8011e-04     51   0.54034 0.57241 0.0072206
## 24 5.2737e-04     55   0.53707 0.57431 0.0072306
## 25 5.0628e-04     61   0.53380 0.57494 0.0072340
## 26 4.7463e-04     75   0.52547 0.57367 0.0072273
## 27 4.2190e-04     81   0.52262 0.57536 0.0072362
## 28 3.9553e-04     90   0.51841 0.57473 0.0072329
## 29 3.7971e-04     95   0.51640 0.57557 0.0072373
## 30 3.6916e-04    101   0.51366 0.57663 0.0072429
## 31 3.4279e-04    109   0.51071 0.57663 0.0072429
## 32 3.1642e-04    115   0.50860 0.57726 0.0072462
## 33 2.8126e-04    136   0.50163 0.57758 0.0072479
## 34 2.6369e-04    146   0.49879 0.58190 0.0072706
## 35 2.3732e-04    173   0.49109 0.58528 0.0072883
## 36 2.1095e-04    177   0.49014 0.58907 0.0073080
## 37 1.9923e-04    209   0.48339 0.59224 0.0073244
## 38 1.8985e-04    273   0.46219 0.59392 0.0073331
## 39 1.7579e-04    280   0.46071 0.59414 0.0073342
## 40 1.6876e-04    307   0.45523 0.60004 0.0073645
## 41 1.5821e-04    315   0.45375 0.60394 0.0073844
## 42 1.5235e-04    348   0.44816 0.60521 0.0073909
## 43 1.4063e-04    360   0.44626 0.60880 0.0074090
## 44 1.3561e-04    369   0.44489 0.61101 0.0074202
## 45 1.3184e-04    378   0.44352 0.61249 0.0074276
## 46 1.2891e-04    382   0.44299 0.61291 0.0074298
## 47 1.0547e-04    418   0.43677 0.62662 0.0074980
## 48 9.0406e-05    533   0.42285 0.62947 0.0075120
## 49 8.4379e-05    541   0.42211 0.63812 0.0075543
## 50 7.9106e-05    551   0.42126 0.63875 0.0075574
## 51 7.5339e-05    569   0.41968 0.64213 0.0075737
## 52 7.0316e-05    576   0.41915 0.64213 0.0075737
## 53 6.3284e-05    618   0.41567 0.64402 0.0075829
## 54 6.0271e-05    643   0.41409 0.64424 0.0075839
## 55 5.2737e-05    650   0.41367 0.65151 0.0076188
## 56 4.2190e-05    722   0.40956 0.65278 0.0076248
```

```
## 57 3.5158e-05    727   0.40935 0.66143 0.0076658
## 58 2.8766e-05    766   0.40797 0.66343 0.0076752
## 59 2.6369e-05    777   0.40766 0.66575 0.0076861
## 60 2.3439e-05    797   0.40713 0.66575 0.0076861
## 61 2.1095e-05    808   0.40681 0.66765 0.0076950
## 62 1.5068e-05    813   0.40671 0.66797 0.0076965
## 63 1.0000e-06    820   0.40660 0.66913 0.0077019
```

We can see that it finds discrete points (in blue) where the actual tree changes and a branch is pruned off. Beyond $\alpha = 0.1$, there is no more pruning as we are just left with the root node. We can also have a look at the surrogate splits that were used.

```
# Get surrogate splits information
surrogate_info <- fit$frame[fit$frame$var != "<leaf>", c("var", "n", "ncompete",
↪  "nsurrogate")]
surrogate_info <- surrogate_info[surrogate_info$nsurrogate > 0, ]  # Filter for nodes
↪  with surrogates

# Display the surrogate splits
cat("\nSurrogate Splits Information:\n")
```

```
##
## Surrogate Splits Information:
```

```
print(head(surrogate_info))
```

```
##                var     n ncompete nsurrogate
## 1     relationship 39774        4          5
## 32      occupation 16482        4          2
## 64   education.num 13439        4          1
## 128            age 12458        4          5
## 515   capital.gain    30        4          4
## 129 marital.status   981        4          1
```

This is interpreted as follows:

- The `var` column represents the primary splitting variable for the node and `n` is the number of observations in that node.
- The `ncompete` is the number of competing splits available for that node. A competing split is an alternative variable or threshold that could have been chosen to split the node but wasn't selected as the primary split. Essentially, it provides insight into other potential splits that the tree considered at that node.
- The `nsurrogate` indicates the number of surrogate splits available for the node.

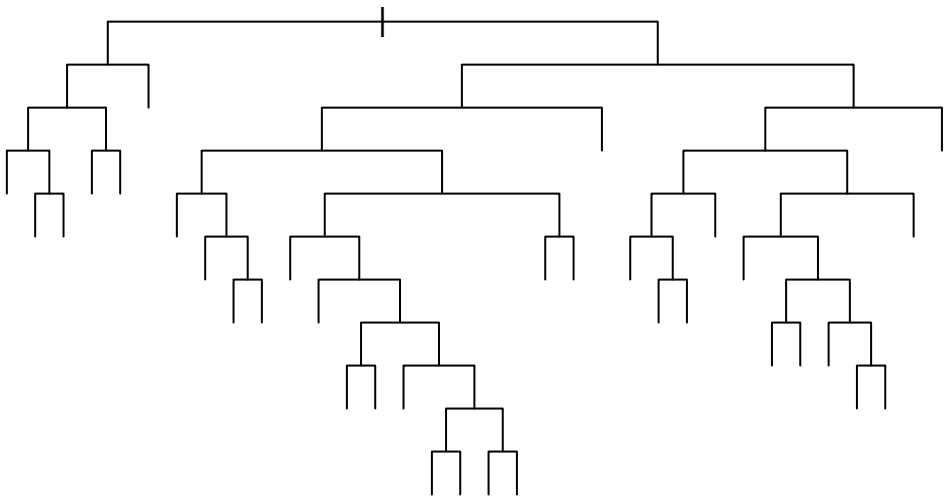
The code filters for nodes with at least one surrogate split (surrogate_info$nsurrogate > 0), focusing on nodes where surrogate splitting is relevant.

We can identify the best value of α by using that value which minimises the cross-validated error.

```
# Identify the best cp based on minimum xerror
best_cp <- cptable[which.min(cptable[,"xerror"]), "CP"]
cat("Best CP:", best_cp, "\n")
```

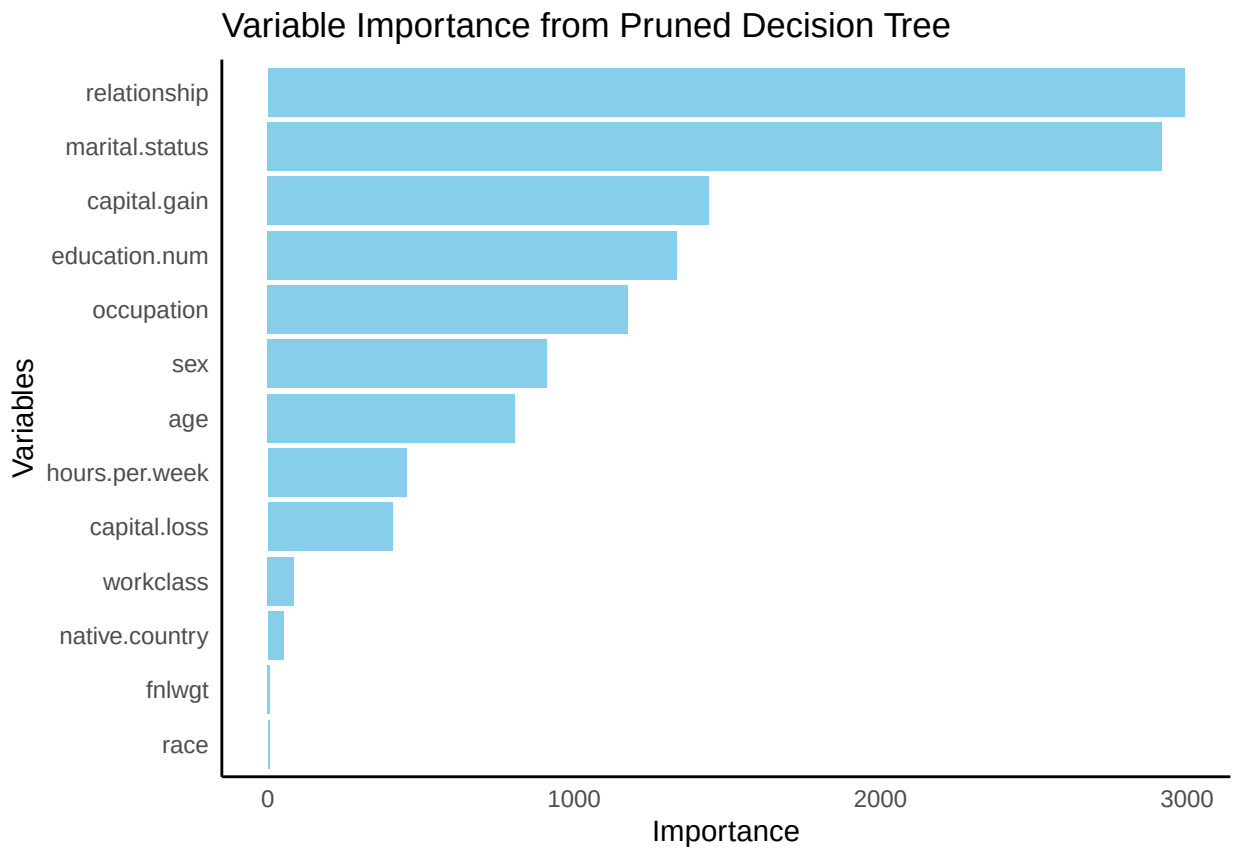
```
## Best CP: 0.0009141089
```

```
# Prune the tree using the best cp
pruned_tree <- prune(fit, cp = best_cp)

# Plot the pruned tree
plot(pruned_tree, uniform = TRUE, main = "Pruned Classification Tree")
```




```
# Extract variable importance from the pruned tree
variable_importance <- data.frame(Variable = names(pruned_tree$variable.importance),
                                  Importance = pruned_tree$variable.importance)

# Sort the variable importance in descending order
variable_importance <- variable_importance %>%
  arrange(desc(Importance))

# Create a bar plot using ggplot
ggplot(variable_importance, aes(x = reorder(Variable, Importance), y = Importance)) +
  geom_bar(stat = "identity", fill = "skyblue") +  # Create bars
  coord_flip() +  # Flip coordinates for better readability
  labs(title = "Variable Importance from Pruned Decision Tree",
       x = "Variables",
       y = "Importance") +
  theme_minimal() +
  theme(
```

```
    panel.grid.major = element_blank(),  # Remove major gridlines
    panel.grid.minor = element_blank(),  # Remove minor gridlines
    axis.line = element_line(color = "black")  # Add x and y axis lines in black
  )
```



The results are expected from the EDA we did. In particular, relationship and marital status were closely related, so it makes sense that their importances are about the same. We can see that capital gain is much better at predicting income than capital loss. This again makes sense as we saw that the distributions of capital gain for each income class were markedly different, whereas the distributions of capital loss for each income class were not very different.

5 Evaluation of Model

It is important to note that the data is imbalanced. Therefore, using accuracy as a performance metric can be misleading. We explore several metrics below and explore how the model performs.

5.1 Confusion Matrix, Specificity, Sensitivity

The confusion matrix shows the predicted class against the actual class.

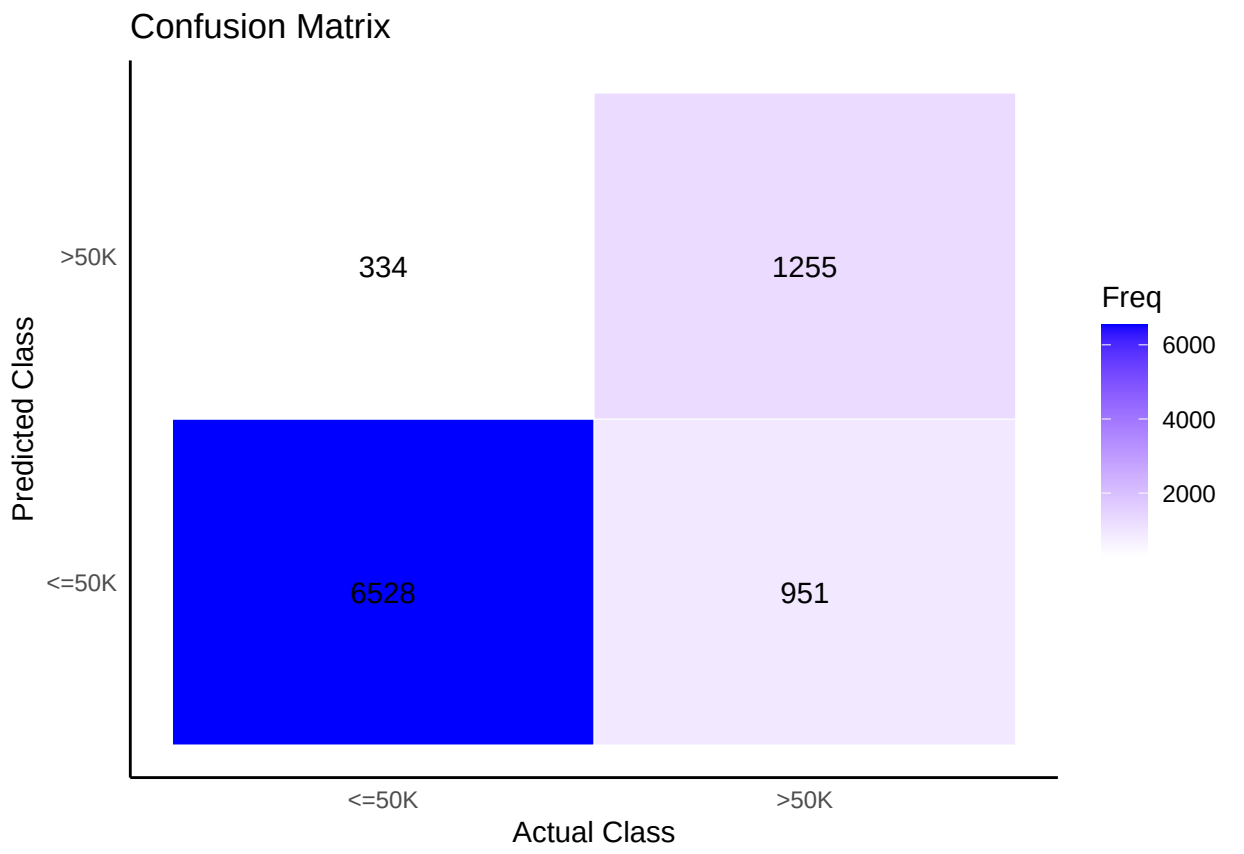
```
# Make predictions on the test set
predictions <- predict(pruned_tree, newdata = X_test, type = "class")
```

```r
# Load necessary libraries
library(caret)  # For confusionMatrix function
library(ggplot2)  # For plotting
library(reshape2)  # For reshaping the data

# Assuming 'predictions' and 'y_test' are defined
# Create a confusion matrix
confusion_mat <- confusionMatrix(as.factor(predictions), as.factor(y_test[,1]))

# Print confusion matrix statistics
print(confusion_mat)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction <=50K >50K
##      <=50K  6528  951
##      >50K    334 1255
##
##                Accuracy : 0.8583
##                  95% CI : (0.8509, 0.8654)
##     No Information Rate : 0.7567
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5748
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9513
##             Specificity : 0.5689
##          Pos Pred Value : 0.8728
##          Neg Pred Value : 0.7898
##              Prevalence : 0.7567
##          Detection Rate : 0.7199
##    Detection Prevalence : 0.8248
##       Balanced Accuracy : 0.7601
##
##        'Positive' Class : <=50K
##
```

```r
# Extract the confusion matrix data for plotting
cm_table <- confusion_mat$table

# Convert the confusion matrix to a data frame for ggplot
cm_df <- as.data.frame(cm_table)

# Create a heatmap of the confusion matrix
ggplot(data = cm_df, aes(x = Reference, y = Prediction)) +
  geom_tile(aes(fill = Freq), color = "white") +  # Create tiles filled with frequency
  scale_fill_gradient(low = "white", high = "blue") +  # Color gradient from white to
  ↪  blue
  geom_text(aes(label = Freq), vjust = 1) +  # Add frequency text in the tiles
  labs(title = "Confusion Matrix", x = "Actual Class", y = "Predicted Class") +
```

```r
theme_minimal() +
theme(
  panel.grid.major = element_blank(),  # Remove major gridlines
  panel.grid.minor = element_blank(),  # Remove minor gridlines
  axis.line = element_line(color = "black")  # Add x and y axis lines in black
)
```



There's a lot to unpack here. We go through each value one by one and provide a brief interpretation.

- The confusion matrix provides a comprehensive overview of the model's classification performance, focusing on two classes: <=50K (the positive class) and >50K (the negative class).
- The model achieves an *accuracy* of 85.95%, meaning it correctly classified about 86% of all instances in the dataset. Accuracy is a straightforward metric that reflects the overall correctness of the model's predictions. However, it can be misleading here because the class distribution is imbalanced. In this context, the No Information Rate (NIR) is 75.67%, which indicates the accuracy that could be achieved by always predicting the majority class. The model significantly outperforms this baseline, with a p-value of less than 2.2e-16, indicating a statistically significant improvement over just predicting the majority class. However, we should not take this at face value, as we have a lot of data here. Therefore, even small deviations from a null can be statistically significant. It is therefore up to us to check if there is practical significance as well.
- *Sensitivity* is calculated at 94.74%, revealing that the model correctly identifies approximately 95% of actual positive cases (<=50K). This measure is crucial in applications where identifying positive instances is essential, as it indicates the model's effectiveness in detecting true positives.

- Conversely, *specificity* stands at 58.61%, which measures the model's ability to correctly identify negative instances (>50K). A specificity of 58.61% suggests that the model has difficulty accurately predicting the negative class, resulting in a higher rate of false positives. This imbalance may have adverse consequences depending on the application context.
- The *Kappa statistic*, which is 0.583, assesses the level of agreement between predicted and actual classifications beyond what would be expected by chance. A Kappa value between 0.41 and 0.60 indicates moderate agreement, suggesting that while the model performs reasonably well, there is still room for improvement.
- The *positive predictive value* (PPV) is 87.69%, meaning that when the model predicts <=50K, it is correct about 88% of the time. This is also called *precision*.
- The *negative predictive value* (NPV), recorded at 78.17%, shows that when the model predicts >50K, it is correct about 78% of the time. While this is a decent score, it reflects some room for improvement in these predictions.
- The *prevalence* is the proportion of actual instances that belong to the positive class (<=50K). Here it is 75.67%. This indicates the imbalance in the data.
- The *detection rate*, which is 71.69%, indicates the proportion of actual positives that were correctly identified by the model. This metric is essential as it reflects how effectively the model can find positive cases within the dataset.
- The *detection prevalence*, at 81.76%, reflects the proportion of positive predictions made by the model (both true and false). This higher value compared to the detection rate suggests that the model is predicting a larger number of positives than it is correctly identifying, aligning with the previously noted issues with specificity.
- Lastly, the *balanced accuracy*, calculated at 76.68%, is the average of sensitivity and specificity. This metric provides a more balanced view of the model's performance, especially in cases of class imbalance. A balanced accuracy of 76.68% indicates that while the model performs well in detecting positives, there is still a notable gap in accurately identifying negatives.

In summary, while the model demonstrates strong sensitivity and accuracy in predicting the positive class (<=50K), it struggles with specificity, leading to potential issues with false positives. Improvements could be made by adjusting the decision threshold or experimenting with different algorithms to enhance the model's ability to accurately classify negative instances (>50K).

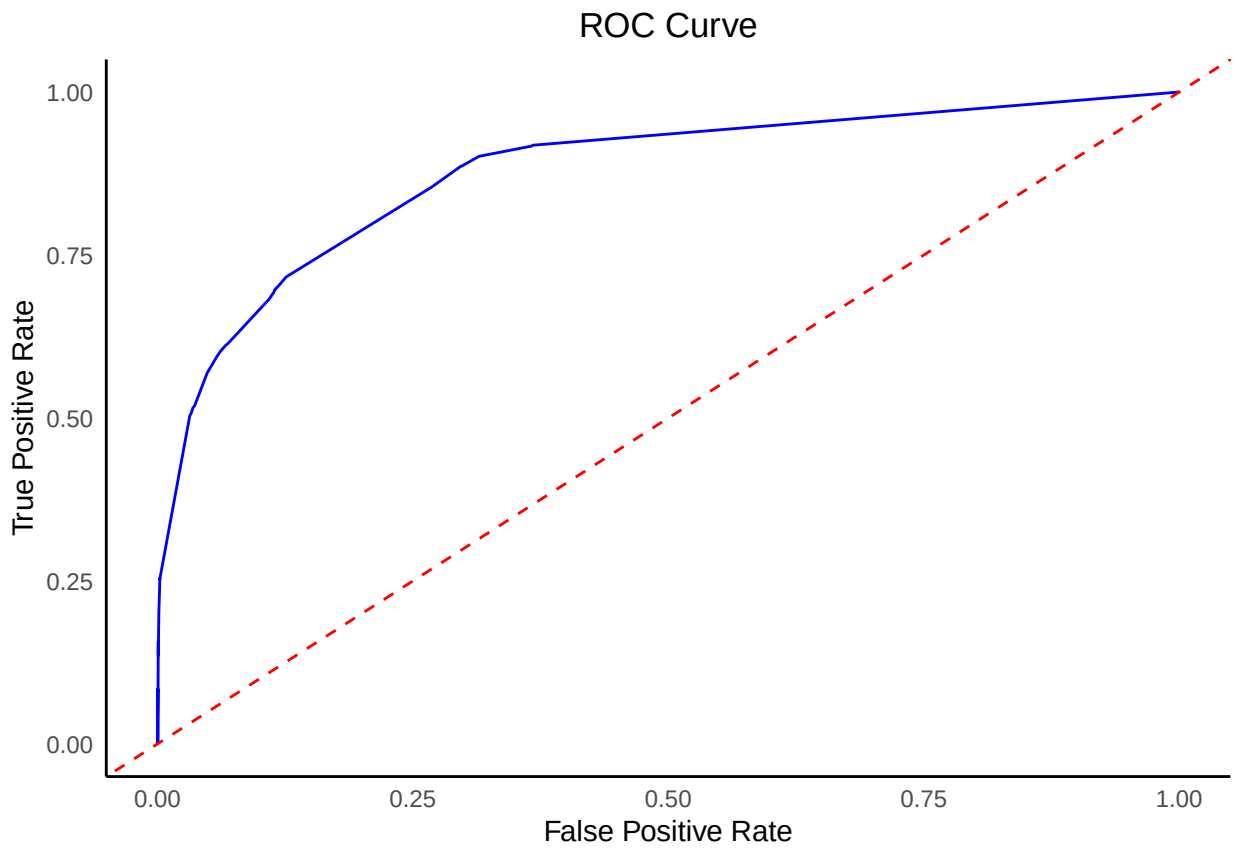
5.2 ROC Curve and F1 Statistic

We next look at other measures called the precision, recall and the ROC curve.

```
# Convert predictions and actual values to factors
predictions_factor <- as.factor(predictions)
y_test_factor <- as.factor(y_test[, 1])  # Adjust as needed for the outcome variable

# Extract predicted probabilities for the positive class from the model=
pred_probs <- predict(pruned_tree, X_test, type = "prob")[, "<=50K"]  # Use the
↪  probabilities for the positive class

# Compute ROC curve using the predicted probabilities
roc_curve <- roc(y_test_factor, pred_probs, levels = c("<=50K", ">50K"))
```

```
## Setting direction: controls > cases
```

```r
# Create a data frame for ROC curve data
roc_data <- data.frame(
  FPR = 1 - roc_curve$specificities,  # False Positive Rate
  TPR = roc_curve$sensitivities         # True Positive Rate
)

# Plot ROC curve
ggplot(roc_data, aes(x = FPR, y = TPR)) +
  geom_line(color = "blue") +           # ROC curve
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +  # Diagonal
  ↪  line
  labs(title = "ROC Curve",
       x = "False Positive Rate",
       y = "True Positive Rate") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust = 0.5),  # Center the title
    panel.grid.major = element_blank(),         # Remove major gridlines
    panel.grid.minor = element_blank(),         # Remove minor gridlines
    axis.line = element_line(color = "black")  # Add x and y axis lines in black
  ) +
  xlim(0, 1) +  # Set x-axis limits
  ylim(0, 1)    # Set y-axis limits
```



```r
# Calculate the AUC
auc_value <- auc(roc_curve)
cat("AUC:", auc_value, "\n")  # Print AUC value
```

```
## AUC: 0.8776763
```

```r
# Calculate Precision, Recall, and F1 Score
precision <- posPredValue(predictions_factor, y_test_factor, positive = "<=50K")
recall <- sensitivity(predictions_factor, y_test_factor, positive = "<=50K")
F1 <- (2 * precision * recall) / (precision + recall)

# Print the metrics
cat("Precision:", precision, "\n")
```

```
## Precision: 0.872844
```

```r
cat("Recall:", recall, "\n")
```

```
## Recall: 0.9513261
```

```r
cat("F1 Score:", F1, "\n")
```

```
## F1 Score: 0.9103968
```

```r
# Create the 'roc_data' directory if it doesn't exist
if (!dir.exists("roc_data")) {
  dir.create("roc_data")
}

# Save ROC data to a CSV file
write.csv(roc_data, "roc_data/roc_data_model_surrogate.csv", row.names = FALSE)
```

We explain what each metric means in context of this data.

- The *ROC curve* is a plot illustrating the trade-off between the sensitivity and specificity. It is obtained by plotting the true positive rate (sensitivity) against the false positive rate (1 - specificity). Some sources also plot the sensitivity against the specificity (see [2]). This is equivalent as it corresponds to a reflection about $x = 0.5$ in the x-axis (horizontal axis). The ROC curve is insensitive to imbalance, as the sensitivity and specificity are probabilities conditioned on the value of Y and therefore do not depend on the distribution of Y [Section 4.2, 3]. The ROC curve shows that the performance is decent, with an area under curve (AUC) of about 0.88. Ideally, we would like a true positive rate of 1 when the false positive rate is 0.
- The *precision* is another word for the positive predictive value which we saw earlier.
- The *recall* of 0.95 indicates that the model correctly identifies about 95% of all actual positive cases (true positives) in the dataset. This means the model is effective at capturing most of the instances that truly belong to the positive class.

- The *F1 score* of about 0.91 is the harmonic mean of precision and recall, providing a single metric that balances both the concerns of false positives and false negatives.

Before moving on, we highlight some other measures which were not used here. The AUC turns out to be identical to the value of another measure of predictive power called the *concordance index*. For further details, the reader is invited to consult [4]. We also highlight to the reader that other measures of predictive power also exist, such as the F-measures, G-measures. and precision-recall curves. The reader may consult [5] for further details.

6 Conclusion

In this section, we have explored surrogate splits and looked at several measures of performance. The latter included measures such as accuracy, sensitivity, specificity, F1 statistics and many more. We found that accuracy is not a good measure of performance here due to the imbalance. A classifier predicting the majority class will have an accuracy equal to the no information rate which is about 75%. We will therefore use the ROC curve instead, which will visually indicate the trade off between specificity and sensitivity.

Overall, the classifier has a lot of room for improvement. It is clear that dealing with missingness is a challenging problem, so we try to use other methods of dealing with missingness in the next section.

7 References

[1] Breiman, Leo. Classification and regression trees. Routledge, 2017.

[2] Hastie, Trevor, et al. The elements of statistical learning: data mining, inference, and prediction. Vol. 2. New York: springer, 2009.

[3] Fawcett, Tom. "An introduction to ROC analysis." Pattern recognition letters 27.8 (2006): 861-874.

[4] Agresti, Alan. Categorical data analysis. Vol. 792. John Wiley & Sons, 2012.

[5] He, Haibo, and Edwardo A. Garcia. "Learning from imbalanced data." IEEE Transactions on knowledge and data engineering 21.9 (2009): 1263-1284.

8 Appendix

8.1 Sensitivity Analysis

We dropped the `education` variable in the above analysis. In the code below, we show that the results are nearly unchanged whether or not it is there. This is reassuring as we know that the same information is already encoded in `education_num`.

```
# Create a function to fit a model and return cross-validated error and cptable
get_cv_error <- function(formula, train_data) {
  fit <- rpart(formula, data = train_data, method = "class", control = rpart.control(cp =
↪  1e-6))

  # Get the best (lowest) cross-validated error from the cptable
  min_xerror <- min(fit$cptable[,"xerror"])

  # Return the fitted model and cross-validated error table (cptable)
```

```r
  return(list(fit = fit, min_xerror = min_xerror, cptable = fit$cptable))
}

# Model 1: Using both "education" and "education_num"
cv_error_with_education <- get_cv_error(income ~ ., train_data)

# Model 2: Dropping the "education" variable, using only "education_num"
train_data_no_education <- train_data[, !names(train_data) %in% "education"]
cv_error_no_education <- get_cv_error(income ~ ., train_data_no_education)

# Print the best cross-validated errors for both models
cat("\nBest Cross-validated Error (with education):", cv_error_with_education$min_xerror)
```

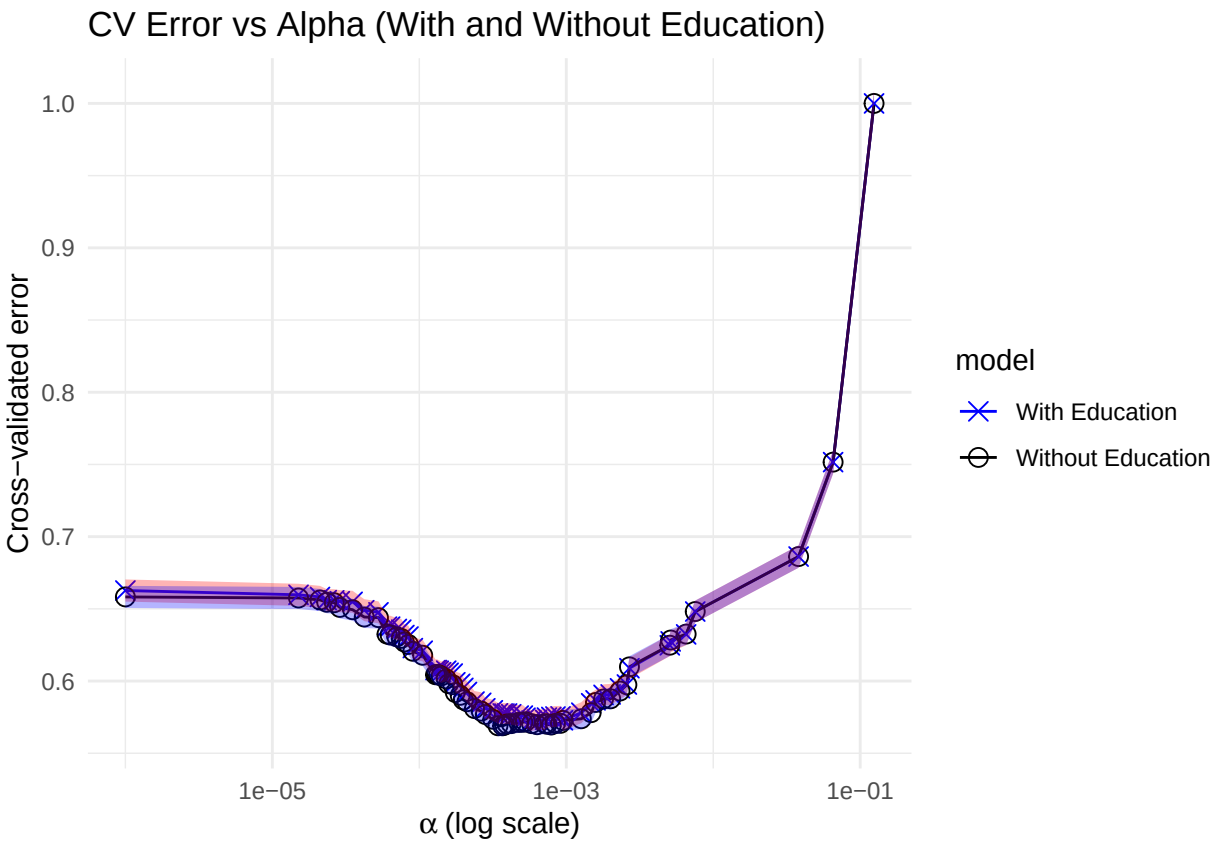
```
##
## Best Cross-validated Error (with education): 0.5726189
```

```r
cat("\nBest Cross-validated Error (without education):",
↪  cv_error_no_education$min_xerror)
```

```
##
## Best Cross-validated Error (without education): 0.5690328
```

```r
# Create data frames from the cptable of both models
data_with_education <- data.frame(CP = cv_error_with_education$cptable[,"CP"],
                                  xerror = cv_error_with_education$cptable[,"xerror"],
                                  xstd = cv_error_with_education$cptable[,"xstd"])

data_no_education <- data.frame(CP = cv_error_no_education$cptable[,"CP"],
                                xerror = cv_error_no_education$cptable[,"xerror"],
                                xstd = cv_error_no_education$cptable[,"xstd"])

# Combine the data for both models, adding a column to distinguish between them
data_with_education$model <- "With Education"
data_no_education$model <- "Without Education"

combined_data <- rbind(
  data_with_education,
  data_no_education
)

# Plot the combined cross-validated error curves for both models
ggplot(combined_data, aes(x = CP, y = xerror, color = model, shape = model)) +
  geom_line() +
  geom_point(size = 3) +
  geom_ribbon(data = data_with_education, aes(x = CP, ymin = xerror - xstd, ymax = xerror
  ↪  + xstd),
              fill = "red", alpha = 0.3, inherit.aes = FALSE) +
  geom_ribbon(data = data_no_education, aes(x = CP, ymin = xerror - xstd, ymax = xerror +
  ↪  xstd),
              fill = "blue", alpha = 0.3, inherit.aes = FALSE) +
  scale_x_log10() +
```

```
labs(title = "CV Error vs Alpha (With and Without Education)",
     x = expression(alpha ~ "(log scale)"),
     y = "Cross-validated error") +
theme_minimal() +
scale_shape_manual(values = c("With Education" = 4, "Without Education" = 1)) +  # 'x'
↪  for one, '.' for other
scale_color_manual(values = c("With Education" = "blue", "Without Education" =
↪  "black"))
```



Since the best models with or without `education` perform similarly, we will not be very fussy about this in later sections.