IEP Workshop Training: Classification Tree

Trinh Nguyen, CDFW IEP

March 23, 2022

Recommended Reference Material

Hastie et al. 2009 provides a comprehensive overview of tree-based methods and would be the recommended resource to develop a foundation of knowledge on the topic. From there, Kuhn and Johnson 2016 supplements a more applied approach, specifically providing in-depth examples and R codes to build and assess decision tree models. A very popular decision tree package in R is the `rpart` package, and the vignette is a mixture of both theoretical and applied information on the topic (Therneau and Atkinson 2022). Finally, Kaggle.com is a great online resource that contains various guides and applications of tree-based models in various programming languages, predominately Python and R. One extensive Kaggle guide on decision tree is https://www.kaggle.com/prashant111/decision-tree-classifier-tutorial/notebook.

Workflow diagram

A classification tree is a specific variant of the decision tree model in which the response variable is discrete categories instead of continuous values (regression). Following the general, simplified workflow of this training workshop, a classification tree is appropriate when the response variable is: 1) categorical, 2) the predictor(s) is/are not categorical, and 3) the response is multiclass, or more than 2 response categories (Figure 1). More generally, however, a classification tree model can be applied to a response variable with only 2 response categories (binary classification) and can handle predictor(s) that is/are categorical. Additionally, a decision tree approach can be applied to a continuous response variable (regression tree).

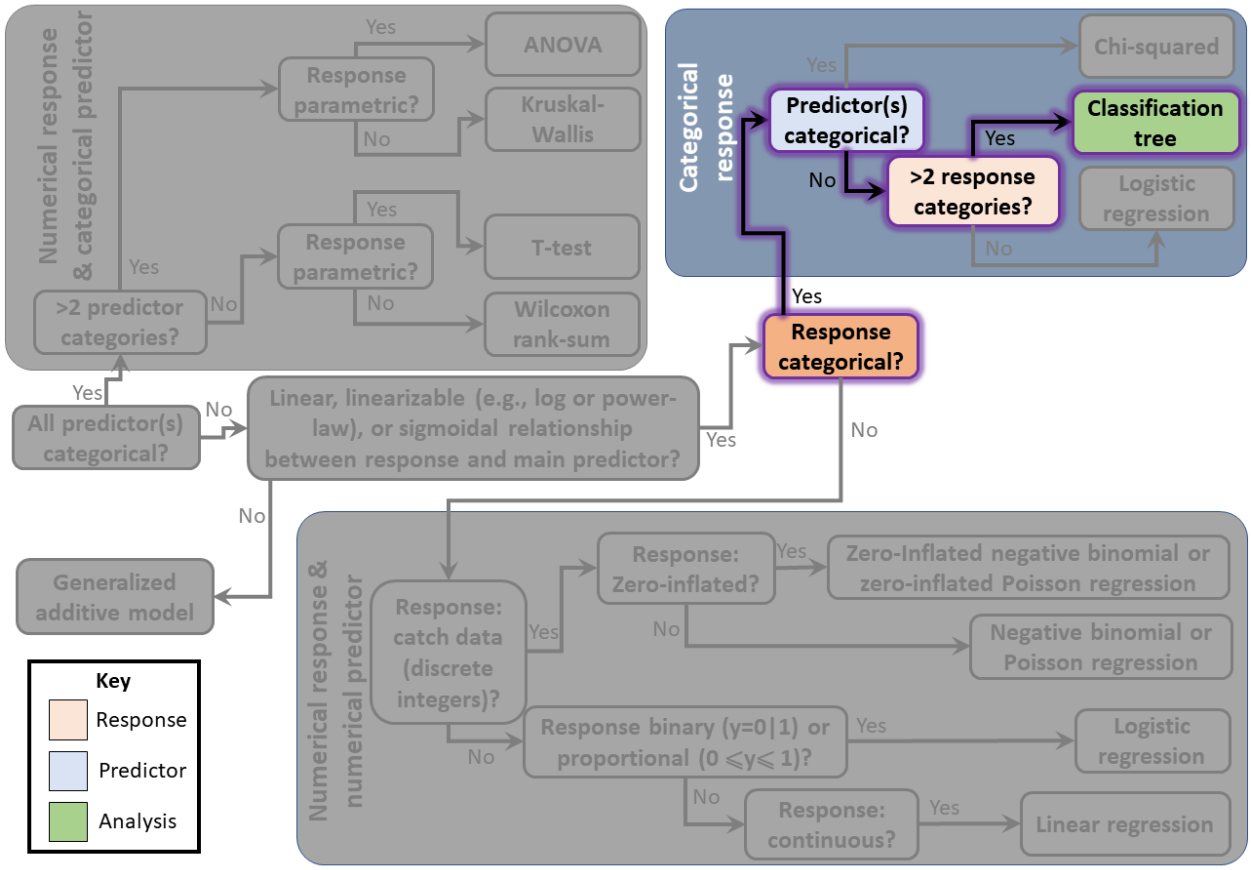

Figure 1: Univariate statistical analysis flowchart with classification tree modeling highlighted.

Introduction to the model

A decision tree model has a similar structure and is read similarly to a dendogram or a dichotomous key. More formally, it is a machine learning alogirthm that partitions the entire predictor space into sets of rectangles and fits a simple model (generally a constant) to each one (Hastie et al. 2009). Essentially, the model is composed of one or more nested "if-then" statements of its predictors (Kuhn and Johnson 2016). This results in a tree-like structure similar to a dichotomous key, in which an outcome (a decision) is reached by comparing the values of various predictors for a data point to the splitting rules outlined by the tree, until a terminal node (outcome) is reached. A classification tree predicting flower species based on petal length and petal width from the `Iris` dataset provides a concrete example of this tree structure (Figure 2, left) and partitioning process (Figure 2, right):

```
# The par command allows us to specify the number of plots on a single figure
# for base R plotting (without any additional packages). Within par(), `mfrow`
# determines the grid make up, so 1 row, 2 columns here and `mar` determines the
# margins of the single figure in the sequence of bottom, left, top, right
par(mfrow = c(1, 2),
    mar = c(7, 7, 2, 2))
# Visualization of the decision tree
# The rpart.plot function takes the final rpart decision tree output as an
# input. The coding syntax here of `modelFullGiniIRIS$finalModel` is specific to
# the `caret` package. The `$` is general R syntax to refer to an element of a
# list. Here, the final model is saved by `caret` as an element of the returned
# list.
rpart.plot(modelFullGiniIRIS$finalModel, extra = 1,
           cex = 2, legend.cex = 2, legend.x = 0.7)
# A decision tree partitions the feature space into sets of rectangles
plot(iris$Petal.Length, iris$Petal.Width,
     xlab = "Petal Length",
     ylab = "Petal Width",
     pch = 20,
     cex = 3, cex.lab = 3, cex.axis = 2,
     col = ifelse(iris$Species == "setosa", "#FB6A4A",
                  ifelse(iris$Species == "virginica", "#74C476",
                         ifelse(iris$Species == "versicolor",
                                "#999999", "white"))),
     mgp = c(4, 1.5, 0))

abline(v = 2.5)
abline(h = 1.8)
text(x = 1.6, y = 1.5, "setosa", col = "#FB6A4A", cex = 3)
text(x = 5, y = 0.5, "versicolor", col = "#999999", cex = 3)
text(x = 3.8, y = 2.3, "virginica", col = "#74C476", cex = 3)
```

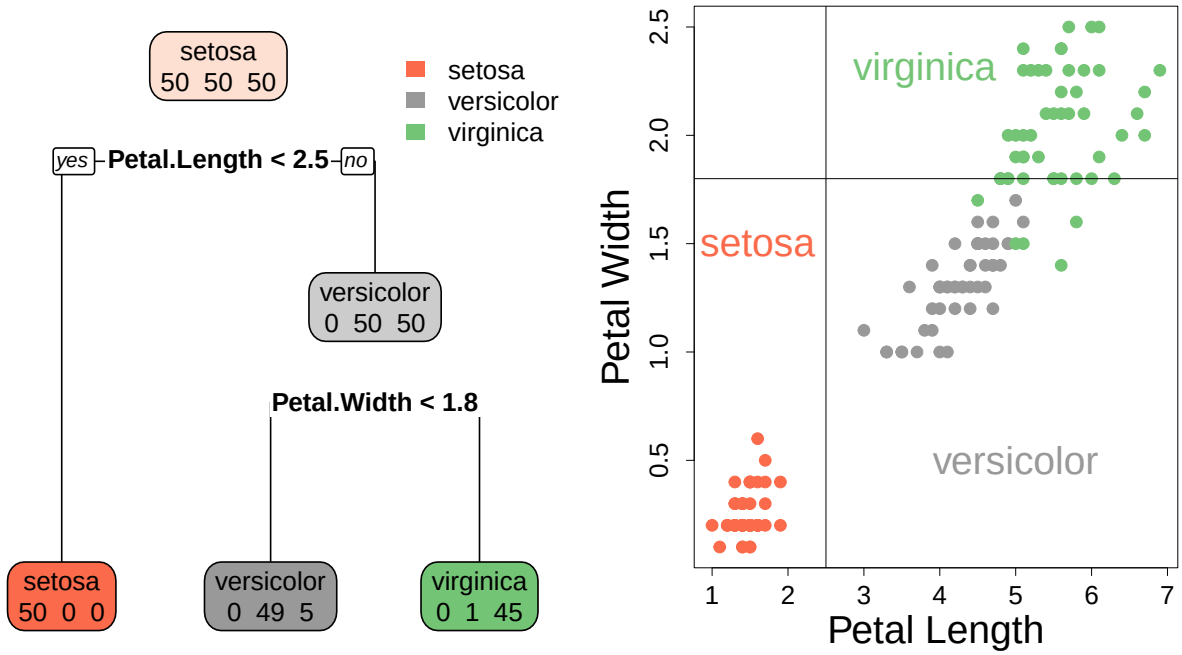

Figure 2: A modeled decision tree of the `Iris` dataset (left) and the resulting partitioning of the predictor space between petal length and petal width across the three flower species (right).

The general interpretation of the `Iris` classification tree is to classify an individual as `setosa` if its petal length is < 2.5 cm, as `versicolor` if its petal length is ≥ 2.5 cm *and* its petal width is < 1.8 cm, and as `virginica` if its petal length is ≥ 2.5 cm *and* its petal width is ≥ 1.8 cm. As mentioned before, this process of assigning an outcome based on splitting values of the predictors is very similar to how one reads a dichotomous key.

Terminology and general depiction of a tree

A decision tree has several specific structures whose terminologies will be used for the remainder of the document:

1. root node, which contains the entire population or sample,
2. splitting, the process in which a node is divided into two or more sub-nodes,
3. decision/internal node, a node that will be further divided into sub-nodes,
4. terminal node/leaf, a node that will no longer be divided into sub-nodes,
5. parent and children nodes, a parent node divides into children nodes,
6. and branch/sub-tree, a subsection of the tree.

Figure 3 visualizes these terms:

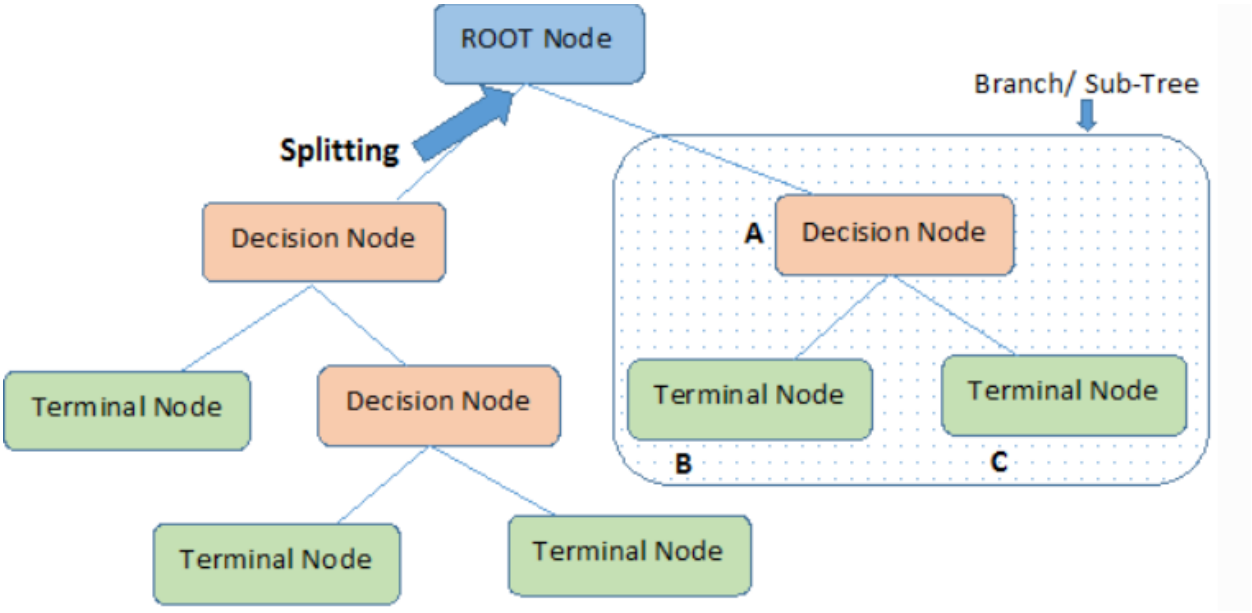

Figure 3: Relevant terminologies for decision trees. The letter A labels a parent node, while letters B and C are the resulting children nodes. Image sourced from https://www.kaggle.com/prashant111/decision-tree-classifier-tutorial/notebook.

The typical depiction a decision tree is:

1. the root node at the top of the tree,
2. the left side of a split is the path taken if the split condition is `TRUE`,
3. and color can be used to depict purity of the node, i.e., greater intensity with the increased proportion of a single class in that node (Therneau and Atkinson 2022).

The tree building process

The most popular tree building algorithm is the CART model (Classification and Regression Tree) (Hastie et al. 2009). The tree is grown using binary splits of the data into smaller groups that are more homogenous in regard to the response variable (Kuhn and Johnson 2016). Homogeneity is generally calculated using two metrics that measures the impurity of the response variable. The first impurity criterion is the Gini index, defined as:

$$I_G = 1 - \sum_{j=1}^{n} p_j^2$$

where I_G is the Gini index, n is the number of j response classes, and p_j is the proportion of the sample that belongs to class j at a particular node. This metric ranges between 0 to 1, where 0 indicates a pure node (i.e., all elements belong to a single class). The second impurity criterion is entropy, defined as:

$$I_H = -\sum_{j=1}^{n} p_j log_2(p_j)$$

where I_H is entropy, n is the number of j classes, and p_j is the proportion of the sample that belongs to a particular class j at a particular node. Entropy is 0 at a node if all samples belong to the same class and is not confined to a maximum of 1. Based on Gini or entropy, the algorithm chooses a split at a node that maximizes the information gain (IG), defined as:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N} I(D_{left}) - \frac{N_{right}}{N} I(D_{right})$$

where, f is the predictor and its splitting value, I is the impurity criterion Gini or entropy, D_p is the population or sample in the parent node, D_{left} is the population or sample in the left child node, D_{right} is the population or sample in the right child node, N is the total number of samples in the parent node, N_{left} is the total number samples in the left child node, and N_{right} is the total number of samples in the right child node. This splitting is recursive and occurs until a stopping criteria is met, which is generally the minimum number of samples in a terminal node or the maximum tree depth (Kuhn and Johnson 2016).

Fully grown trees are often susceptible to overfitting the training data. To remedy this, the fully grown tree is "pruned" to be made more generalized. In this process, every internal node can be collapsed all the way to the root node. The collapse that results in the lowest error on validation or testing data is the preferred final tree (Hastie et al. 2009).

Advantages and disadvantages of a classification tree model

The main advantages of a classification tree are:

1. highly interpretable when the tree is not complex or large,
2. easy to implement,
3. non-parametric (i.e., the data do not have to be normally distributed),
4. non-linear,
5. can handle interactions,
6. can handle different types of predictors,
7. can handle missing data in the predictors,
8. and provides variable importance rankings (i.e., how often a predictor is used to describe the dataset).

The main disadvantages of a classification tree are:

1. model instability, highly susceptible to changes in the underlying data,
2. may not produce the optimal model due to the greedy splitting rule of the algorithm,
3. highly correlated predictors will get used somewhat randomly when deciding splits,
4. and a selection bias towards predictors that have more distinct values (e.g., continuous vs discrete).

Case study illustration

The Chipps Island Trawl dataset from 2011-08-02 to 2021-06-30 was chosen as the case study for this demonstration. The United State Fish and Wildlife Service (USFWS) runs this midwater trawl year-round at Chipps Island (USFWS 2020). In this case study, a multiclass classification tree model will be constructed to predict catch of longfin smelt (LFS) in the survey using water quality metrics recorded during the trawl. The Chipps Island Trawl increases its theoretical sampling frequency to 10 tows a day, 7 days a week during the months of December and January (USFWS 2020), making it an ideal candidate for detecting LFS which migrates into the Delta during the same period of the year (Rosenfield 2010). To simplify the dataset, data will be aggregated into daily values, i.e., catch into daily sums and all predictors into daily means. Catch of LFS will be treated as a multiclass variable purely in the spirit of the workflow provided with workshop (Figure 1). Specifically, LFS catch will be transformed into a three ordered class response variable as: 1)

`no catch`, 2) `low catch` when daily total catch is >0 but $<$ mean daily catch of LFS calculated across all sampling events (1.34 fish), and 3) `high catch` when daily total catch is $\geq$ mean daily catch of LFS calculated across all sampling events. The six environmental predictors will be volume, Secchi, DO, water temperature, turbidity, and conductivity.

```
# Modeling workflow
library(caret)
# Data manipulation
library(dplyr)
# Graphics
library(ggplot2)
# Alternative visualization of the decision tree
library(rpart.plot)

# As a quick aside, there are two main schools of coding in R: 1) "base R" and
# 2) "tidyverse". I generally code using tidyverse so some base R users may be
# confused at the use of "%>%" in the code. This is known as a pipe. The pipe
# takes the dataframe from the previous step and brings it forward.
# Essentially, the dataset is passed from one manipulation to the next, and
# when the pipe ends, the dataframe or operation is finished.

# Reading in the data
# This uses the read_csv() function from the `readr` package. This function
# provides finer control on how the columns are read into R, which is what
# `col_types` is doing in the code below. If `col_types` is not specify, R takes
# a best stab at guessing what format the data is in and may get it wrong,
# e.g., when a column has many missing data, R may think that it is a logical
# column instead of numeric.
data <- read_csv("Chipps Island Trawls CHN & POD Species 2012-2021.csv",
                 col_types = cols(
                   Location = col_character(),
                   Station = col_character(),
                   Date = col_date(format = "%m/%d/%Y"),
                   Time = col_time(format = ""),
                   Method = col_character(),
                   TowNumber = col_double(),
                   TowDuration = col_double(),
                   TowDirection = col_character(),
                   Volume = col_double(),
                   Secchi = col_double(),
                   DO = col_double(),
                   WaterTemp = col_double(),
                   Turbidity = col_double(),
                   Conductivity = col_double(),
                   Weather = col_character(),
                   Species = col_character(),
                   Mark = col_character(),
                   Catch = col_double(),
                   FL = col_double(),
                   Stage = col_character(),
                   Maturation = col_character(),
                   RaceByLength = col_character()
                 )) %>%
  # Focusing on catch of LFS only
```

```
  mutate(Catch = ifelse(Species == "LFS", Catch, 0)) %>%
  # Summing catch and averaging relevant environmental predictors to a daily
  # time step
  # Ignoring station and the number of tows in this demonstration
  group_by(Date) %>%
  summarise(across(c(Volume, Secchi, DO, WaterTemp, Turbidity, Conductivity),
                   ~mean(.x, na.rm = T)),
            Catch = sum(Catch, na.rm = T))
# Using this operation as a specific example of the piping structure in
# tidyverse: each step is a seperate manipulation to the underlying data,
# passing the manipulated data onto the next step. Overall, this operation can
# be translated to: 1) read in the data, 2) change the `Catch` column to replace
# catch of fish not equal to "LFS" to 0, 3) group the data by each day, and 4)
# summarize the dataset to daily mean volume, Secchi, DO, water temperature,
# turbidity, and conductivity and total daily catch. The pipe allows sequential
# manipulations without needing to assign the dataframe to a variable in the
# environment at each manipulation step

# Converting the catch variable to an ordered 3-class response variable.
dataModel <- data %>%
  mutate(Catch = factor(case_when(Catch == 0 ~ "noCatch",
                                  Catch > 0 & Catch < mean(Catch) ~ "lowCatch",
                                  Catch >= mean(Catch) ~ "highCatch"),
                        levels = c("noCatch", "lowCatch", "highCatch"))) %>%
  # Removing date here as that will not be a predictor in the model
  select(-Date)
```

The main study question in this demonstration is: ***"what are the environmental conditions typically associated with no catch of LFS, low catches of LFS, and high catches of LFS?"***

Exploratory data analysis

Following the general modeling framework provided with this workshop (Figure 4), the first step is to explore the structure of the dataset.

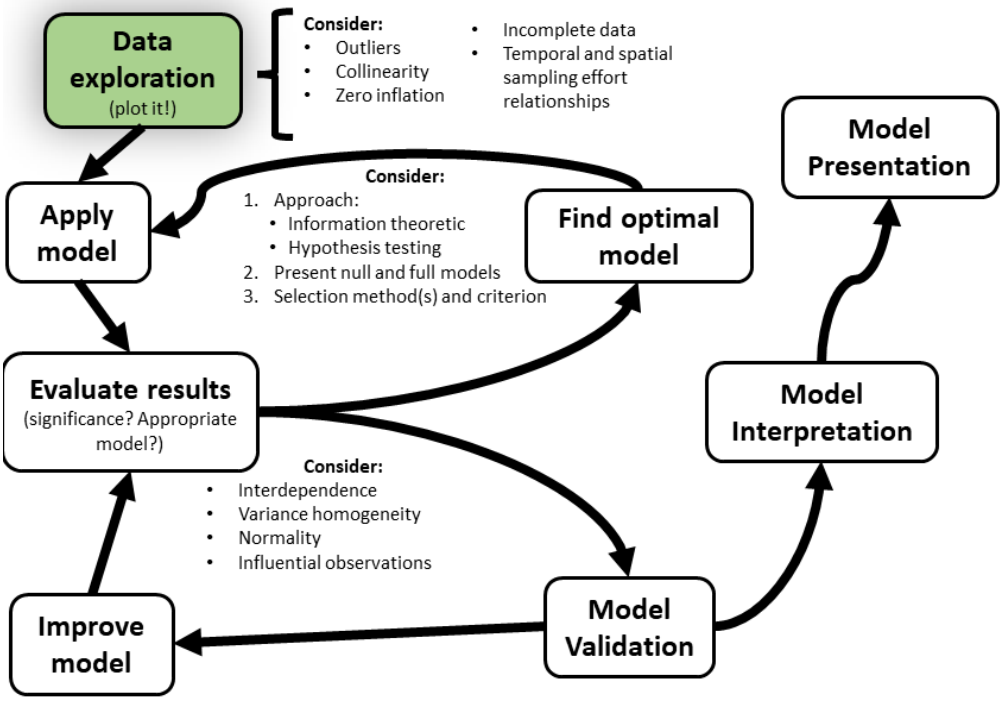

Figure 4: General modeling workflow approach to building a model.

A glimpse of the dataset, its dimensions, and the number of missing datapoints:

```
# head() lets you see the first n number of data rows (default n = 6)
head(dataModel)
```

```
## # A tibble: 6 x 7
##   Volume Secchi    DO WaterTemp Turbidity Conductivity Catch
##    <dbl>  <dbl> <dbl>     <dbl>     <dbl>        <dbl> <fct>
## 1 22592.  0.374   NaN      19.9      42.2          NaN noCatch
## 2 22403.  0.465   NaN      19.9      28.4          NaN noCatch
## 3 20625.  0.438   NaN      20.3      31.9          NaN noCatch
## 4 22671.  0.516   NaN      20.7      29.6          NaN noCatch
## 5 19623.  0.432   NaN      20.8     NaN            NaN noCatch
## 6 22771.  0.378   NaN      20.5      42.3          NaN noCatch
```

```
# This lets us quickly see what data we have, e.g., Secchi is in meters

# Total dimension of the dataset
dim(dataModel)
```

```
## [1] 1669    7
```

```
# Our dataset has 1669 rows and 7 columns, of which 1 is the response variable
# This allows us to then see if any of our predictors suffer from many missing
# datapoints:
```

```
# Number of missing datapoints:
colSums(is.na(dataModel))
```

```
##       Volume       Secchi           DO    WaterTemp    Turbidity Conductivity
##            0            4           60            8           15           56
##        Catch
##            0
```

There are several missing datapoints in the predictors, however, there are not many. Although these datapoints can be imputed, a decision tree model can account for missing data in the predictors. There is no missing data in the response variable (catch data) which allow for the use of the full dataset. A simple tally of the response variable shows a very skewed distribution across the three classes, with instances of `no catch` being the dominant class, followed by `high catch` and `low catch`:

```
dataModel %>%
  group_by(Catch) %>%
  tally()
```

```
## # A tibble: 3 x 2
##   Catch         n
##   <fct>     <int>
## 1 noCatch    1237
## 2 lowCatch    192
## 3 highCatch   240
```

Next, a correlation matrix and biplots can be explored to understand the basic relationships between the predictors with one another and with the response variable. The Spearman's rank coeffficient of correlation is used here to account for non-linear relationships between the predictors and the response variable (Figure 5).

```
# This function is derived from the help page of ?pairs to show numberical cor
# Note the changes to the cor() function below
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  usr <- par("usr"); on.exit(par(usr))
  par(usr = c(0, 1, 0, 1), mgp = c(3, 2.5, 0))
  # Adding in round() here to keep the text font consistent
  # Note that cor() here is changed to calculate Spearman's correlation values
  # instead of the default Pearson. Additionally, we are asking for this
  # calculation on only complete pairs to ignore the NAs in the dataset
  r <- round(abs(cor(x, y, use = "pairwise.complete.obs", method = "spearman")), 2)
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  # By default, cex (which controls the size of the outputted text) is scaled to
  # the absolute correlation value so that larger values are bigger. Due to ADA
  # compliance, this scaling is removed.
  text(0.5, 0.5, txt, cex = cex.cor * 0.8)
}

# Correlation matrix
dataModel %>%
```

```
  mutate(Catch = factor(Catch)) %>%
  pairs(upper.panel = panel.cor, cex.labels = 3, cex.axis = 3,
        oma = c(5, 5, 4, 5), mgp = c(3, 2.5, 0))
```

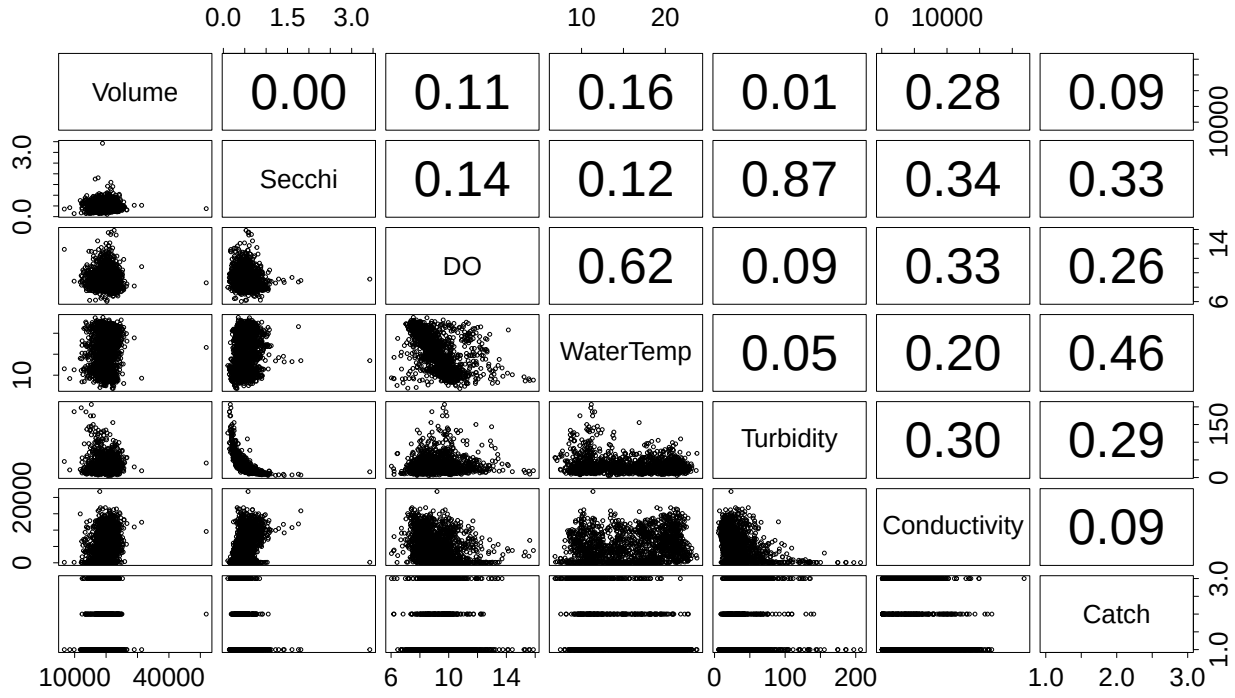

Figure 5: Correlation plot and biplots of the predictors and response variable.

Water temperature is the most correlated predictor to catch of LFS (transformed into an ordered factor) followed by Secchi and turbidity. Secchi and turbidity are highly correlated with one another based on a threshold of 0.65. Although highly correlated predictors will not affect the predictive performance of the classification tree, it may have an effect on the variable importance ranking due to the randomness associated with the selection process of which highly correlated predictor to use to split a node (Kuhn and Johnson 2016). In this demonstration, these highly correlated predictors will not be removed from the analysis, although the removal of turbidity is justified (lower correlation to the response than Secchi). After this quick exploration of the dataset, the model can be built.

Building the model

To build the classification model, 75% of the dataset will randomly be chosen as training data, while the remaining 25% of the dataset will serve as testing data. Due to the susceptability of decision tree models to overfit the data, the testing dataset provides a check for overfitting and a glimpse at the predictive performance of the model.

```
# Use a 75/25 randomized data split to train/test the model.
# This 75/25 split is arbitrary. Generally, with more data, you can afford to
# reserve more to the testing dataset

# The set.seed function is the psuedo-random number generator in R. Before any
# simulation or random object is created in R, the program first draws a number
# to determine the path of the operation. This psuedo-random
```

```r
# process can be made replicable by setting the specific number to draw before
# generation: set.seed(n) below. The number 135 can be replaced with any number
set.seed(135)
splitIndex <- createDataPartition(y = dataModel$Catch, p = 0.75, list = F)
# createDatePartition() is a `caret` function that attempts to create a split
# that balances the classes if the response is a factor.

dataTrain <- dataModel[splitIndex, ]
dataTest <- dataModel[-splitIndex, ]
```

To prune the model, a 10-fold, 3 repeats cross validation approach will be used to tune how large and complex (number of splits) of a tree to build. Cross validation restricts the model from using all of the training data, providing a fairer assessment of model performance (for a more indepth discussion, see https://scikit-learn.org/stable/modules/cross_validation.html). In `rpart`, the complexity parameter (`cp`) controls the complexity of the tree. As for which impurity criterion to use, both can be tried. Generally, Gini is the default choice in the `rpart` package and for the CART algorithm. Finally, the evaluation metric Kappa will be chosen to account for the highly imbalance nature of the response variable (the `no catch` class vastly outnumbers the `low catch` and `high catch` classes).

```r
# Defining the seeds so that each train via cross validation (CV) can be
# reproduced. The length of the list is the number of total resamples you want
# to use + 1 final model using the "optimal" tuning value(s). Here, we will use
# a repeated CV approach with 10 folds and 3 repeats, so there are
# 10 folds * 3 repeats + 1 final model train = 31 models and required seeds.
seedList <- vector("list", 31)
set.seed(135)
for (i in 1:31) seedList[[i]] <- sample.int(n = 1000000, 1)

# First, fit the model against every feature (predictor) in the dataset
# The evaluation metric will be "Kappa" here as the response is not balanced

# This variant uses the default Gini index to make node splits
modelFullGini <- train(Catch ~ .,
                       data = dataTrain,
                       method = "rpart",
                       na.action = na.rpart,
                       trControl = trainControl(method = "repeatedcv",
                                                repeats = 3,
                                                seeds = seedList,
                                                savePredictions = "final",
                                                classProb = T),
                       tuneLength = 10,
                       # # Can specify the specific tuning values if you do not
                       # # want to use tuneLength. This involves using tuneGrid
                       # # and the expand.grid() function for all tunable
                       # # parameters of your model, each specified as a column in
                       # # the grid. `rpart` only has one tunable
                       # # parameter, `cp`. The values below are what
                       # # tuneLength = 10 picked for those that want to replicate it
                       # tuneGrid = expand.grid(
                       #   cp = c(0, 0.006001372, 0.012002743, 0.018004115,
                       #          0.024005487, 0.030006859, 0.036008230,
                       #          0.042009602, 0.048010974, 0.054012346)
```

```
                        #   ),
                        metric = "Kappa")

# This variant uses entropy to make node splits
modelFullInfGain <- train(Catch ~ .,
                          data = dataTrain,
                          method = "rpart",
                          na.action = na.rpart,
                          parms = list(split = "information"),
                          trControl = trainControl(method = "repeatedcv",
                                                   repeats = 3,
                                                   seeds = seedList,
                                                   savePredictions = "final",
                                                   classProb = T),
                          tuneLength = 10,
                          metric = "Kappa")
# tuneLength automatically chooses the range of tuning values to try. More
# information can be found via: getModelInfo("rpart")[[1]]$grid

# For those interested in modeling using the base rpart() function itself:
# the relevant options for a classification tree are
rpart(
  formula = y ~ x,
  data = dataset,
  method = "class",
  na.action = na.rpart,
  parms(list(loss = c(gini, information))),
  # numerical values provided here are the default for the function
  # see ?rpart.control()
  rpart.control(minsplit = 20,
                minbucket = round(minsplit/3),
                cp = 0.01,
                xval = 10)
)
# where y = response variable, x = predictors,
# method = "class" when y = categorical,
# na.action = na.rpart will remove data rows where y is missing but keep rows
# that have missing predictor values,
# parms(list(loss = c(gini, information))) determines which impurity criterion
# to use, gini or entropy,
# minsplit = minimum number obs that must exist in a node before a split can occur,
# minbucket = minimum number obs that must exist in a terminal node,
# cp = complexity parameter that a split must improve the model greater than to occur
# xval = number of cross validation across dfferent cp
```

Note that the models are built using the `caret` package workflow (see https://topepo.github.io/caret/). The main advantage of this workflow is the generalizability across many different modeling techniques and the ability to quickly tune various fitting parameters of the algorithm. For `rpart`, the only tunable parameter is the complexity parameter (`cp`). The `tuneLength` argument in the `train()` function is a `caret` wrapper to automatically choose the specified number (10 here) of tuning values to fit the model; specifically for `rpart`, an initial model is fitted to calculated the range of possible `cp` values. The `plot()` command can then be used on the trained object to understand if an optimal tuning value discovered.

```
# In caret, the tuning process can be visualized with plot() or ggplot()

# 'split' is specific to trellis plots in base R and takes 4 arguments:
# x, y, nx, ny. The first 2 determine position of the plot while the second 2
# determine the grid layout. The 'more' argument tells R that there are
# additional plot(s) to add to the same figure
print(plot(modelFullGini,
           cex = 1.5,
           xlab = list(cex = 3),
           ylab = list(cex = 3),
           scales = list(x = list(cex = 2),
                         y = list(cex = 2)),
           main = list("Gini variant", cex = 2),
           ylim = c(0.05, 0.4)),
      split = c(1, 1, 2, 1), more = T)
print(plot(modelFullInfGain,
           cex = 1.5,
           xlab = list(cex = 3),
           ylab = list(cex = 3),
           scales = list(x = list(cex = 2),
                         y = list(cex = 2)),
           main = list("Entropy variant", cex = 2),
           ylim = c(0.05, 0.4)),
      split = c(2, 1, 2, 1), more = F)
```

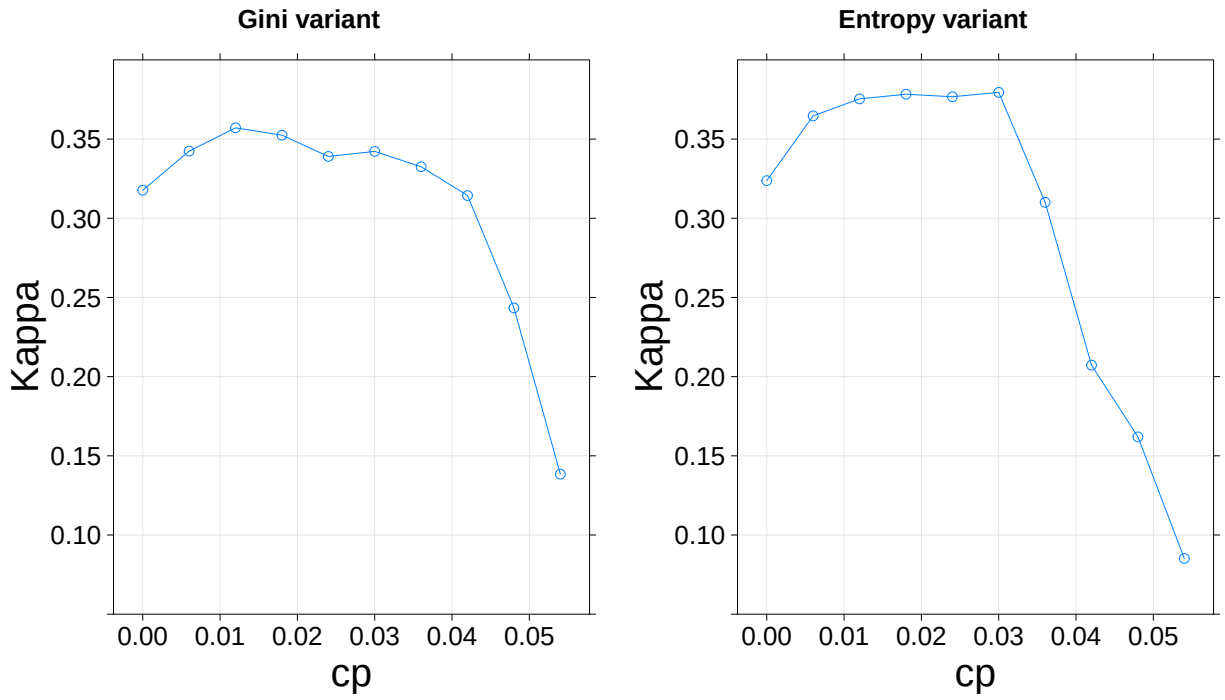

Figure 6: Cross validated Kappa scores across 10 tuning values of the complexity parameter (`cp`) for the Gini variant (left) and entropy variant (right)

```
# Caret does support the use of ggplot() instead of the base R trellis plot.
# Multiple ggplots can be plotted together using the cowplot package or by
# combining the two underlying datasets, specifying a classifier column, and
# then drawn as facets
# ggplot(modelFullGini) # base plot only
```

Ideally, the tuning value that results in the best evaluation metric (highest Kappa here) is bordered by lower and higher tuning values to indicate that an optimal tune has been approximated. This is the case for both variants here (Figure 6). If this was not the case, additional tuning values should be explored.

Evaluating model fit

The trained model is be evaluated on the unseen testing data to assess model fit. A properly trained model should perform similarly across both training and testing datasets; this would mean that the model did not overfit the training data, learning uninformative elements of the dataset.

```
# First, build a data.frame() with four columns: 1) which evaluation variant
# (Gini or entropy), 2) which dataset (training or testing), 3) what was the
# associated tuned value for cp (complexity parameter), and 4) what was the
# resulting Kappa (evaluation metric). Note that the Kappa for the training set
# here is the cross validated mean while Kappa for the testing set is a singular
# value.
evaluationMetricTable <- data.frame(
  variant = c("Gini", "Gini", "entropy", "entropy"),
  dataSet = c("Train", "Test", "Train", "Test"),
  cpFinal = c(rep(filter(modelFullGini$results,
                         Kappa == max(Kappa))[["cp"]], 2),
              rep(filter(modelFullInfGain$results,
                         Kappa == max(Kappa))[["cp"]], 2)),
  Kappa = c(# Gini variant
            filter(modelFullGini$results,
                   Kappa == max(Kappa))[["Kappa"]],
            confusionMatrix(predict(modelFullGini, newdata = dataTest,
                                    na.action = na.rpart),
                            dataTest$Catch)$overall[["Kappa"]],
            # Entropy variant
            filter(modelFullInfGain$results,
                   Kappa == max(Kappa))[["Kappa"]],
            confusionMatrix(predict(modelFullInfGain, newdata = dataTest,
                                    na.action = na.rpart),
                            dataTest$Catch)$overall[["Kappa"]]))

evaluationMetricTable
```

```
##   variant dataSet    cpFinal     Kappa
## 1    Gini   Train 0.01200274 0.3570991
## 2    Gini    Test 0.01200274 0.3733982
## 3 entropy   Train 0.03000686 0.3794874
## 4 entropy    Test 0.03000686 0.4174789
```

```
# This table can be visualized to quickly evaluate the performances of both
# variants (final cp's not visualized)
# ggplot plots in layers connected by the "+" symbol that serves the same
# purpose as the "%>%" pipe. Here, the base layer with the axes is drawn first,
# then the datapoints, and then the lines
ggplot(evaluationMetricTable,
       aes(x = factor(dataSet, levels = c("Train", "Test")),
           y = Kappa,
           shape = variant,
           color = variant)) +
    geom_point(size = 4) +
    geom_line(aes(x = c(1, 2, 1, 2)), size = 1.1) +
    labs(x = "Dataset",
         color = "Variant",
         shape = "Variant") +
    theme_bw(base_size = 24)
```

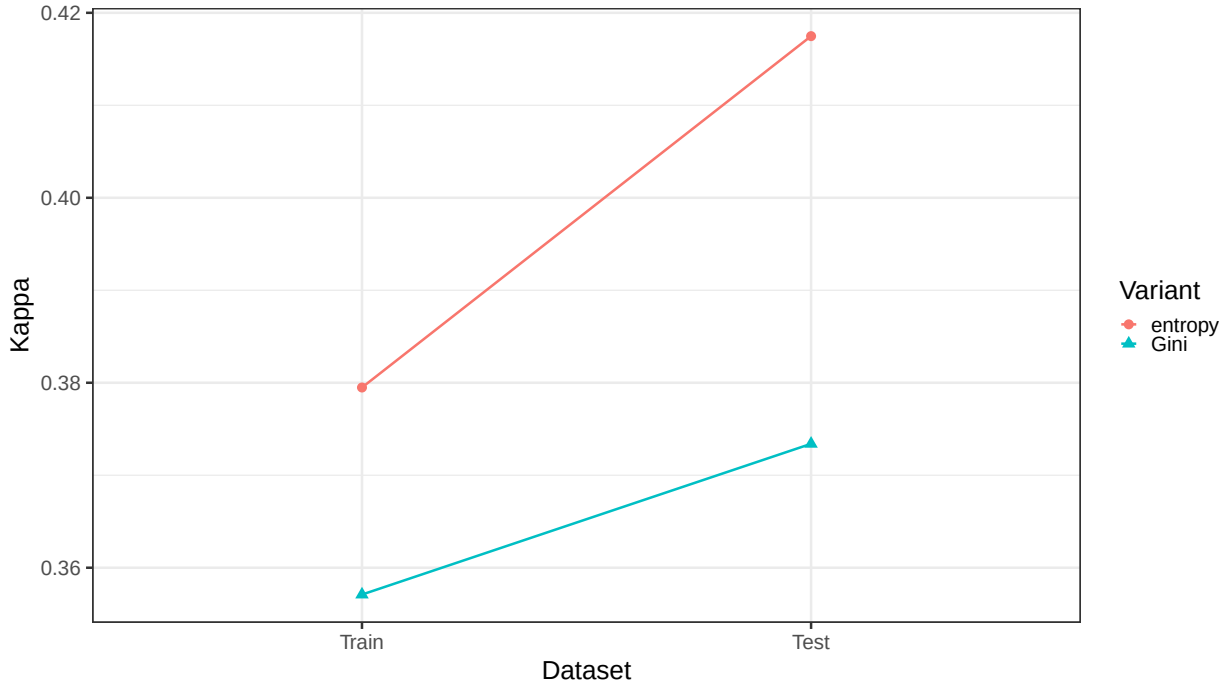

Figure 7: Kappa values of both variants on the training and the testing datasets.

The variant trained to split using entropy performed better than the Gini variant on both the training and testing datasets (Figure 7). Unfortunately, even for the better performing entropy variant, performance is poor based on the low evaluation metric Kappa. A confusion matrix of the training and testing datasets can provide insights as to why:

```
# Gini model, training dataset:
confusionMatrix(modelFullGini, norm = "none")$table
```

```
##            Reference
## Prediction  noCatch lowCatch highCatch
##   noCatch      2630      330       239
##   lowCatch       18        5        19
##   highCatch     136       97       282
```

```
# Entropy model, training dataset:
confusionMatrix(modelFullInfGain, norm = "none")$table
```

```
##            Reference
## Prediction  noCatch lowCatch highCatch
##   noCatch      2579      302       210
##   lowCatch        0        0         0
##   highCatch     205      130       330
```

```
# Gini model, testing dataset, the na.action is required here given that there
# are NAs in the predictors of the model
confusionMatrix(predict(modelFullGini, dataTest, na.action = na.rpart),
                dataTest$Catch)$table
```

```
##            Reference
## Prediction  noCatch lowCatch highCatch
##   noCatch       297       34        29
##   lowCatch        0        0         0
##   highCatch      12       14        31
```

```
# Entropy model, testing dataset:
confusionMatrix(predict(modelFullInfGain, dataTest, na.action = na.rpart),
                dataTest$Catch)$table
```

```
##            Reference
## Prediction  noCatch lowCatch highCatch
##   noCatch       290       30        22
##   lowCatch        0        0         0
##   highCatch      19       18        38
```

A confusion matrix compares the counts of model predicted (depicted as rows) against observed (depicted as columns) across each class and provides a visualization of model performance. In the training dataset, the entropy variant never predicted a `low catch` event, while the Gini variant predicted a few `low catch` events but still much less than the truth (21.875%). In the testing dataset, both variants failed to predict any `low catch` events. This could be because the testing data did not possess any low catch datapoints, however, this is not the case:

```
dataTest %>% group_by(Catch) %>% tally()
```

```
## # A tibble: 3 x 2
##   Catch         n
##   <fct>     <int>
## 1 noCatch     309
## 2 lowCatch     48
## 3 highCatch    60
```

Relevant visualizations

The decision tree can be visualized from the `rpart` package (Figure 8 and 9):

```
plot(modelFullGini$finalModel, margin = 0.05, main = "Gini variant", cex.main = 3)
text(modelFullGini$finalModel, cex = 2, use.n = T)
```

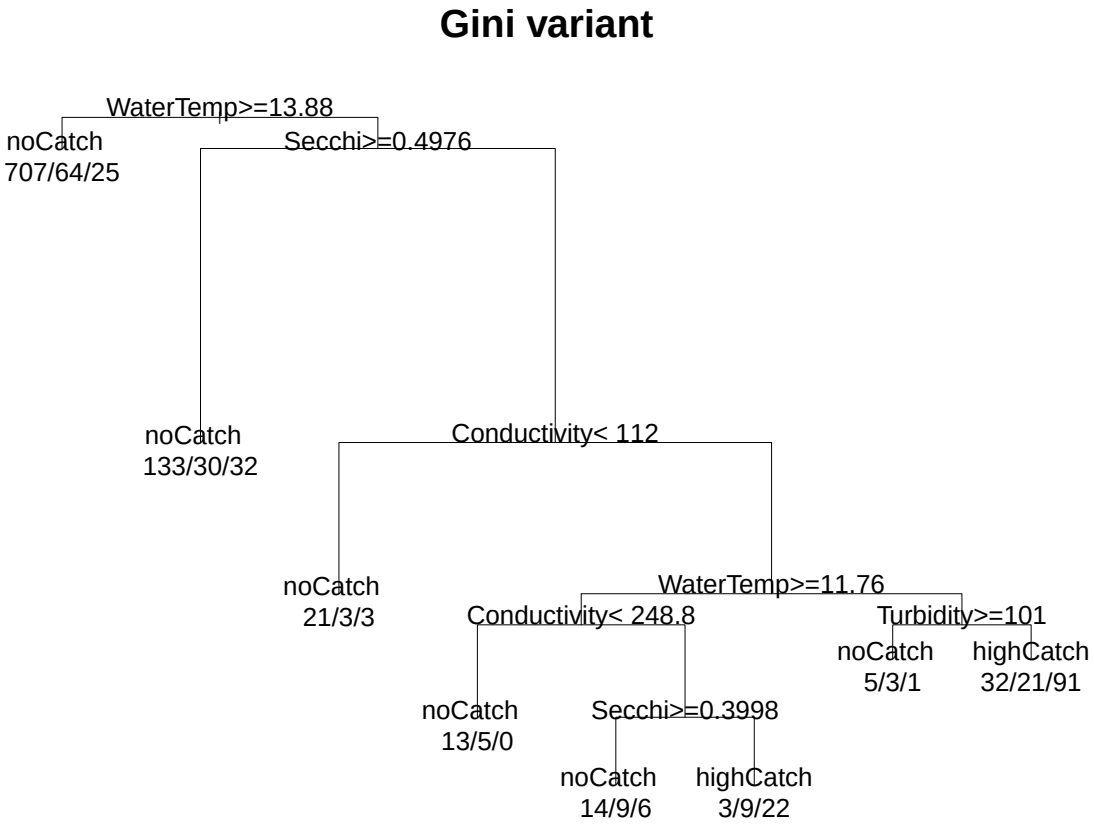

Figure 8: Decision tree visualized from the rpart package, Gini variant.

```
# Gini variant
```

```
plot(modelFullInfGain$finalModel, margin = 0.05, main = "Entropy variant", cex.main = 3)
text(modelFullInfGain$finalModel, cex = 2, use.n = T)
```

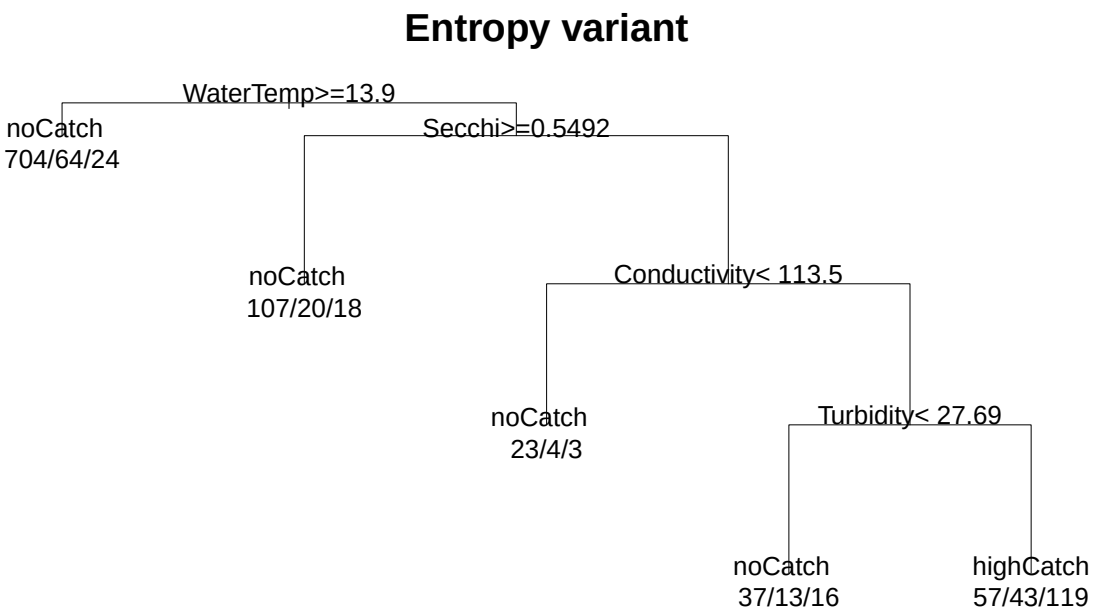

Figure 9: Decision tree visualized from the rpart package, Entropy variant.

```
# Entropy variant
```

As discussed earlier, the tree is generally drawn with the root node at the top of the tree, the splitting predictor and its splitting value at each node, and the outcome that satisfy the splitting rule to the left of the split. The final classification of the terminal node/leaf is displayed alongside the number of data points contained in that leaf. An alternative visualization of the tree is to use the `rpart.plot` package, which adds color intensity to the image proportional to the purity of the node (Figure 10 and 11):

```
rpart.plot(modelFullGini$finalModel, type = 1, extra = 1,
           cex = 2, legend.cex = 2,
           main = "Gini Variant", cex.main = 2)
```

```
# The more complex a tree, the harder it is to depict. There is some
# flexibility in depicting the tree via rpart.plot(), but complex trees are
# overall hard to visualize cleanly

# Gini variant
```

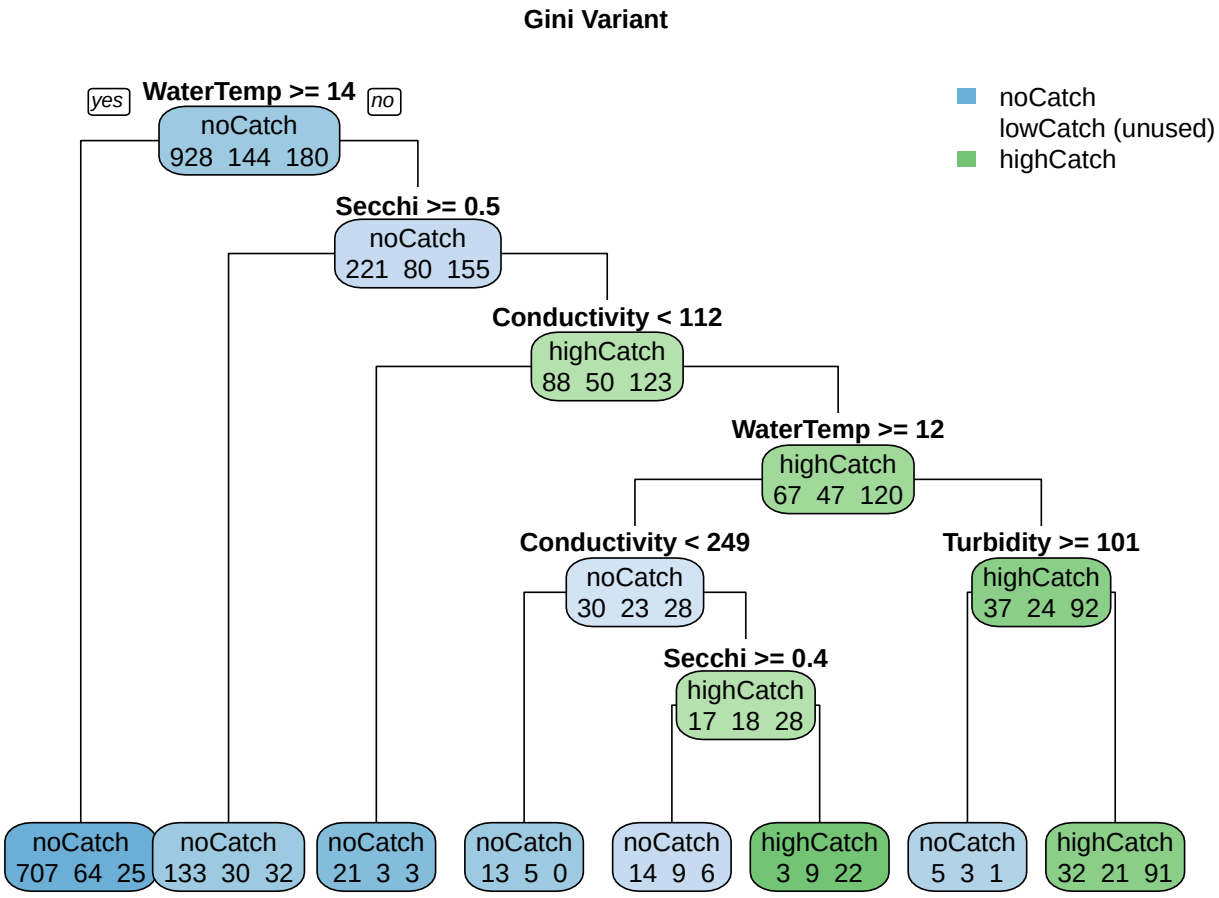

Figure 10: An alternative visualization of the decision tree using the rpart.plot package, Gini variant.

```
rpart.plot(modelFullInfGain$finalModel, type = 1, extra = 1,
           cex = 2, legend.cex = 2,
           main = "Entropy Variant", cex.main = 2)
```

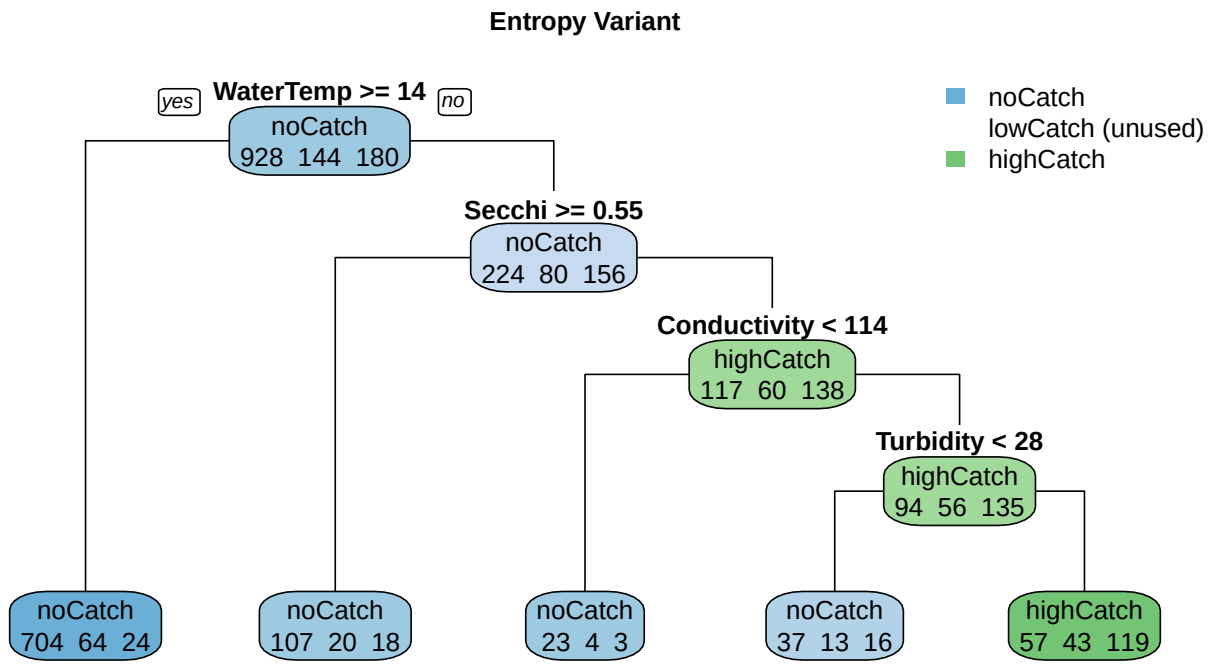

Figure 11: An alternative visualization of the decision tree using the rpart.plot package, entropy variant.

```
# Entropy variant
```

Finally, the variable importance rankings of the decision tree can also be plotted:

```
print(plot(varImp(modelFullGini),
           cex = 1.5,
           xlab = list(cex = 3),
           ylab = list(cex = 3),
           scales = list(x = list(cex = 2),
                         y = list(cex = 2)),
           main = list("Gini variant", cex = 2)),
      split = c(1, 1, 2, 1),
      more = T)
print(plot(varImp(modelFullInfGain),
           cex = 1.5,
           xlab = list(cex = 3),
           ylab = list(cex = 3),
           scales = list(x = list(cex = 2),
                         y = list(cex = 2)),
           main = list("Entropy variant", cex = 2)),
      split = c(2, 1, 2, 1),
      more = F)
```

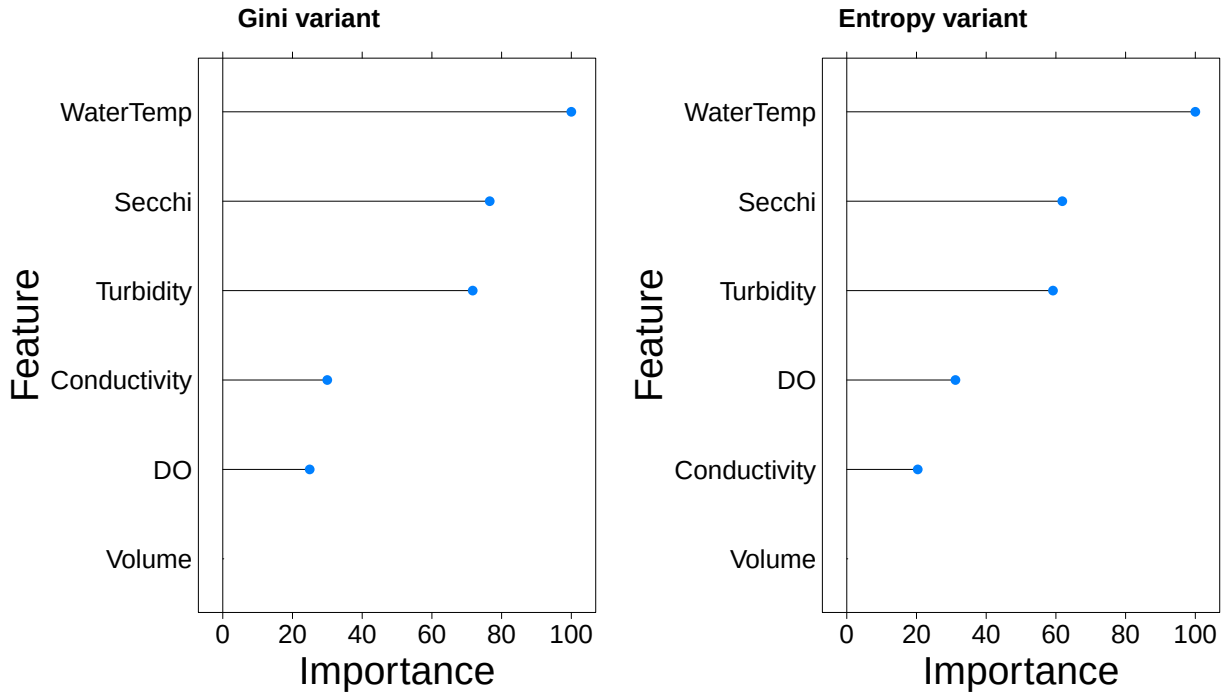

Figure 12: Variable importance rankings of the Gini variant (left) and entropy variant (right).

Variable importance is a measure of the improvement a predictor provides in the overall tree. At each split, improvement to the splitting criterion is attributed to the predictor responsible for the split; once the tree is fully grown, an aggregation of the overall improvement per predictor yields a variable importance ranking. Predictors that appear higher or multiple times in the tree will rank higher than predictors lower or appear less in the tree (Kuhn and Johnson 2016). For this demonstration, water temperature is the most important variable, followed by Secchi and turbidity (Figure 12). This is a similar conclusion to the correlation table constructed prior to modeling (Figure 5). Since Secchi and turbidity is highly correlated, the removal of turbidity (less correlated to catch than Secchi) may increase the importance of Secchi.

Conclusion to the case study

A classification tree was built to predict `no catch`, `low catch`, and `high catch` of LFS in the Chipps Island Trawl dataset (2011-08-02 to 2021-06-30) using environmental data collected during the trawl. The model was trained using a 75/25% data split. A 10-fold, 3 repeats cross validation approach using Kappa as the evaluation metric was used to prune the tree. The entropy variant performed slightly better on the training and testing data than the Gini variant. However, both models performed poorly overall in terms of Kappa. This was due to the inability by the model to predict `low-catch` conditions. Nevertheless, the resulting decision trees (Figures 8-11) do provide sensible conclusions on when LFS catch occurs at Chipps Island: when water temperature is cool and when water clarity is low. The poor performance of the model is likely due to the arbitrary definition of the response variable into 3 classes.

Publication considerations

Relevant parameters to present in a publication are:

1. the splitting criterion, e.g., Gini index, entropy,
2. the evaluation metric, e.g., Kappa, accuracy, Sensitivity, Specificity,
3. the tuning values of the complexity parameter,
4. model performance via testing data, e.g., a confusion matrix and the evaluation metric,
5. visualization of the final tree,
6. and visualization of model variable importance ranking.

Some example publications are Friedl and Brodley 1997, Shouman et al. 2011, and Xu et al. 2005.

Literature Cited

CDFW (2020). California Endangered Species Act Incidental Take Permit No. 2081-2019-066-00, California Department of Fish and Wildlife, Bay Delta Region (CDFW), West Sacramento, CA.

Friedl, Mark A., and Carla E. Brodley. "Decision tree classification of land cover from remotely sensed data." Remote sensing of environment 61.3 (1997): 399-409.

Hastie, T., R. Tibshirani and J. Friedman (2009). The elements of statistical learning: data mining, inference, and prediction. Second edition. Springer Science & Business Media.

Kuhn, M. and K. Johnson (2016). Applied predictive modeling. 5 edition. Springer Nature, New York.

Rosenfield, J. (2010). Life history conceptual model and sub-models for longfin smelt, San Francisco Estuary population. Report submitted to the Sacramento-San Joaquin Delta Regional Ecosystem Restoration Implementation Plan (DRERIP).

Shouman, Mai, Tim Turner, and Rob Stocker. "Using decision tree for diagnosing heart disease patients." Proceedings of the Ninth Australasian Data Mining Conference-Volume 121. 2011.

Therneau, Terry M., and Elizabeth J. Atkinson. "An Introduction to Recursive Partitioning Using the RPART Routines." (2022).

USFWS (2020). Metadata for the Lodi Fish and WIldlife Office's Delta Juvenile Fish Monitoring Program. Lodi Fish and Wildlife Office, Lodi, CA.

Xu, Min, et al. "Decision tree regression for soft classification of remote sensing data." Remote Sensing of Environment 97.3 (2005): 322-336.