CSI 2300: Intro to Data Science

In-Class Exercise 24: Modeling – Classification

In this lecture, we're going to do several things:

- Load in and normalize the `abalone` dataset. The abalone is a type of marine snail.
- Apply classification methods to predict the class of abalone (Male, Female, Infant) based on its physical measurements (length, diameter, height, several weights, and number of rings).
- Look at how the training and test accuracy change as the model complexity changes.

Inside of an abalone (source: https://commons.wikimedia.org/wiki/File:AbaloneInside.jpg)

We'll break this down into a number of steps, as usual.

1. The original source of this dataset was https://archive.ics.uci.edu/ml/datasets/Abalone; we have added variable names and saved it in the R data format for you. The variables are described at the web page above. The class we are predicting is `Sex`, which has labels Male, Female, and Infant.

- Load in the `abalone` dataset from the file `abalone.rda`.
- Check that the `Sex` variable is indeed a factor.

2. Start with the code below. First, add `{r}` to the triple backtick. Then add a few lines to create a random split of the abalone dataset into data frames `train` and `test`, with 30% (1253 observations) in the train set, and 70% (2924 observations) in the test set.

```
# Note: this code chunk doesn't have {r} on the line above; add it when you want
# to use it.
n <- nrow(abalone)

# create a vector that contains the numbers 1 through n, but in
# shuffled order.
p <- sample(1:n)

train_size <- 1253

# now add code to create the "train" and "test" data frames (these can be
# written as one line of code each, using what we have so far)
```

3. Normalize the scale of the independent variables (all but `Sex`) using the code below, and then answer the following questions:
    - Why is it important to normalize the independent variables?
    - Explain what each line in the code below does.
    - Why are the standard deviations computed from the training data variables used to scale the testing data variables?

```
# Note: this code chunk doesn't have {r} on the line above; add it when you want
# to use it.
dep_ndx <- which(colnames(abalone) == 'Sex')

for (v in 1:ncol(abalone)) {
    if (v == dep_ndx) { next }
    s <- sd(train[,v])
    train[,v] <- train[,v] / s
    test[,v]  <- test[,v] / s
}
```

4. Let's try classifying the `Sex` of the abalone observations using a nearest neighbor classifier. Remember that when using `knn(...)`, you should leave the dependent variable **out** of the first and second arguments (only independent variables are used to compute neighbors).

   - Load the `class` library.
   - Run `knn` on the **training** data to make predictions on the (same) **training** data. Give the confusion matrix and accuracy.
   - Run `knn` on the **training** data to make predictions on the **test** data. Give the confusion matrix and accuracy.
   - Compare the train and test accuracies. Do they differ a lot? What does this tell us?

5. Using $k = 1$ was likely not a great model for our test data. We may be able to do better if we "smooth" out our KNN classifier using a larger k. Run the code below. Then answer these questions from the plots:

   - Where is the model overfitting?
   - Where is it underfitting?
   - How can you tell if it is overfitting or underfitting?
   - What k would you choose and why?

```
# Note: this code chunk doesn't have {r} on the line above; add it when you want
# to use it.

# pre-declare our results vectors
train_accuracy <- test_accuracy <- NULL
dep_ndx <- which(colnames(train) == 'Sex')
k_vals <- seq(1, 63, by=2)
for (k in k_vals) {
    train_predictions <- knn(train[,-dep_ndx], train[,-dep_ndx], train[,dep_ndx], k=k)
    test_predictions <- knn(train[,-dep_ndx], test[,-dep_ndx], train[,dep_ndx], k=k)
    train_accuracy <- append(train_accuracy, mean(train$Sex == train_predictions))
    test_accuracy <- append(test_accuracy, mean(test$Sex == test_predictions))
}

par(mfrow=c(1,2))
plot(k_vals, train_accuracy, main="Train")
plot(k_vals, test_accuracy, main="Test")
```

6. Let's try building a tree classifier on the training data.
   - Load the `rpart` library.
   - Construct a tree model using the `train` data that predicts `Sex` based on all other variables. Assign this model to the variable name `m`.
   - Plot the tree (`margin=0.2` as an argument in the `plot` command helps it fit), and label the nodes (`text(m, use.n=T)`).
   - What independent variable test is used at the root of the tree?
   - Answer the following questions about the performance of the model:
     - What is the confusion matrix on the `train` data?
     - What is the confusion matrix on the `test` data?
     - What is the accuracy of the model on the `train` data?
     - What is the accuracy of the model on the `test` data?
   - Does this look like a good model?

7. The tree model you got was likely *underfitting* – it may have never even predicted one of the classes.

   We can control the complexity penalty in `rpart` using the `cp` argument to `rpart`. The lower that `cp` is, the more complicated the tree can grow, and the more flexible the tree becomes.

   Try the following loop and plots, and **describe what you see in the plots**.

   - What happens to the train accuracy, test accuracy, and complexity of the tree as `cp` decreases?
   - Is there a point where the model is underfitting?
   - Is there a point where the model is overfitting?

```
# Note: this code chunk doesn't have {r} on the line above; add it when you want
# to use it.

# declare three vectors
train_accuracy <- test_accuracy <- num_tree_nodes <- NULL
cp_vals <- 1.3 ^ seq(-15, -30) # try different (small) values of cp
range(cp_vals)
for (cp in cp_vals) {
    # train the tree
    m <- rpart(Sex ~ ., data=train, cp=cp)

    train_predictions <- predict(m, train, type="class")
    train_accuracy <- append(train_accuracy, mean(train$Sex == train_predictions))
    test_predictions <- predict(m, test, type="class")
    test_accuracy <- append(test_accuracy, mean(test$Sex == test_predictions))

    num_tree_nodes <- append(num_tree_nodes, nrow(m$frame))
}
# Note: below we use ‘xlim‘ to plot cp_vals from *greatest* to *least*,
# reversing the normal x-axis order for plotting. This is because as the
# complexity penalty decreases, the model gets more complex, and it feels
# natural to think about the complexity increasing as we go right on the graph.
par(mfrow=c(1,2))
plot(cp_vals, train_accuracy, xlab="complexity penalty",
     main="Train", log="x", xlim=c(max(cp_vals), min(cp_vals)))

plot(cp_vals, test_accuracy, xlab="complexity penalty",
     main="Test", log="x", xlim=c(max(cp_vals), min(cp_vals)))

par(mfrow=c(1,1))
plot(cp_vals, num_tree_nodes, xlab="complexity penalty",
     main="# of tree nodes", log="x", xlim=c(max(cp_vals), min(cp_vals)))
```