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. IT IS

```
#add abalone dataset
load("abalone.rda")

#check that sex variable factor
is.factor(abalone$Sex)
# [1] TRUE
```

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)
```

```
n <- nrow(abalone)

p <- sample(1:n)

train_size <- 1253

train <- abalone[p[1:train_size],]
test <- abalone[p[(train_size+1):n],]
```

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? Normalizing independent variables is important because many algorithms are sensitive to feature scales, improving performance and preventing bias.

- Explain what each line in the code below does.

Check snippet!

```
- Why are the standard deviations computed from the training data variables
  used to scale the testing data variables?
```

Standard deviations from training data are used to scale testing data to prevent data leakage and ensure consistent transformations.

```
# 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
}
```

```r
dep_ndx <- which(colnames(abalone) == 'Sex') # index of dependent variable

for (v in 1:ncol(abalone)) { # look through each column
    if (v == dep_ndx) { next } # if V is sex skit it
    s <- sd(train[,v]) # implement standard deviation on training data
    train[,v] <- train[,v] / s # each value in the current column of the train dataset
    test[,v]  <- test[,v] / s # same as above, to ensure no data leakage
}
```

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?

```r
library(class)

train_predictions <- knn(train=train[,-1], test=train[,-1], cl=train[,1])

test_predictions <- knn(train=train[,-1], test=test[,-1], cl=train[,1])

confusionMatrix(train[,1], train_predictions)
#     F   I   M
# F 400   0   0
# I   0 405   0
# M   0   0 448
confusionMatrix(test[,1], test_predictions)
#     F   I   M
# F 356 145 406
# I 157 598 182
# M 402 215 462
```

Using k=1, the train accuracy is typically much higher than the test accuracy. They differ quite a bit. This tells us that the model is likely overfitting the training data. With k=1, the model is very sensitive to individual training examples and may not generalize well to unseen data.

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?

The model is overfitting when k is small. You can see this in the plots because the train accuracy is very high, while the test accuracy is significantly lower. The model is fitting the noise in the training data.

- Where is it underfitting?

The model appears to be underfitting when k is very large. As k increases, the test accuracy eventually starts to level off or even slightly decrease. The model becomes too simple and is not capturing the underlying patterns well enough.

- How can you tell if it is overfitting or underfitting?

You can tell by comparing the train and test accuracies.

- What k would you choose and why?

Based on the provided plots, you would look for the value of k where the test accuracy is highest or plateaus.

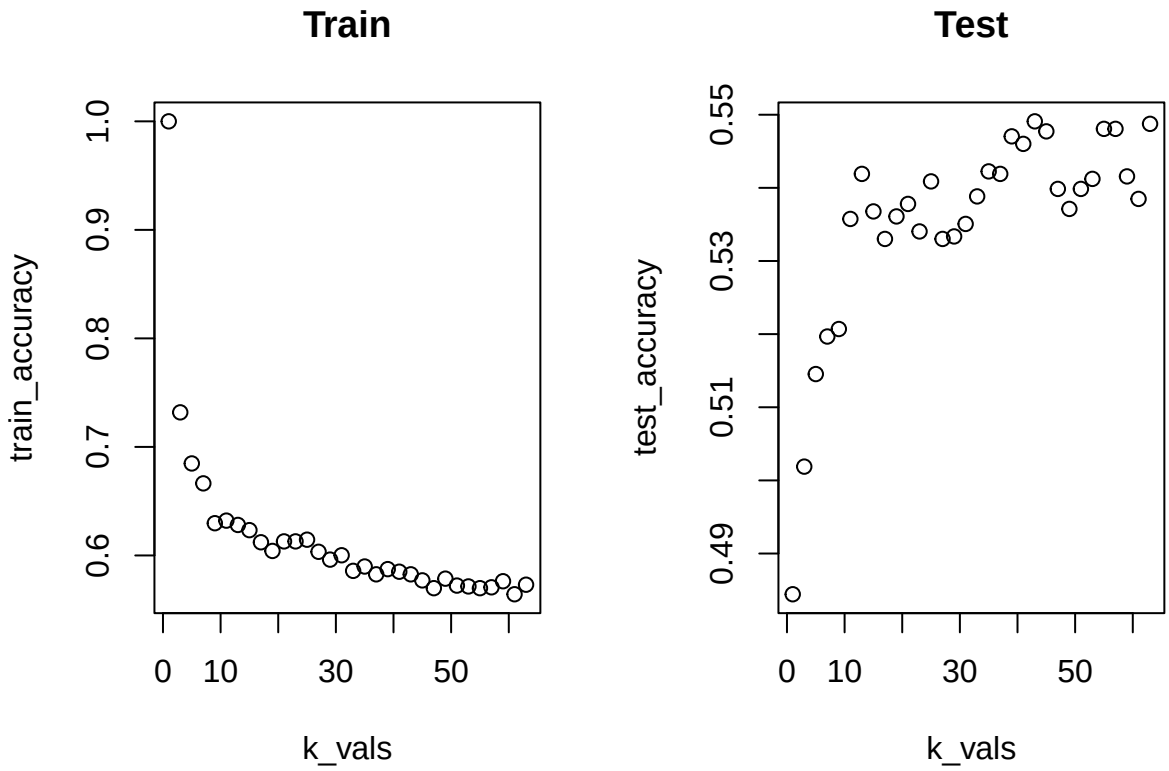
```
# Note: this code chunk doesn't have {r} on the line above; add it when you want
# to use it.

# 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")
```

```r
# 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")
```



Model is overfitting when K is small, the training accuracy is much higher. When k is too big the model underfits the test accuracy starts to drop again.

6. Let's try building a tree classifier on the training data.

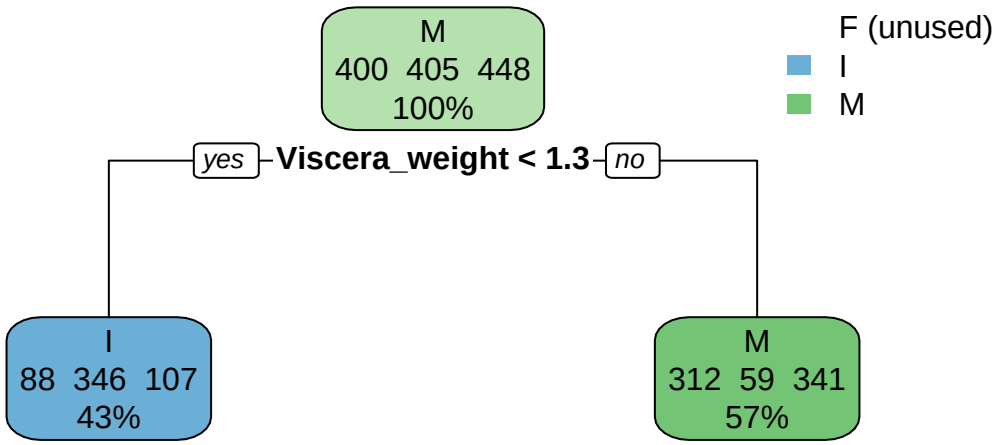
   - Load the `rpart` library.
   - Construct a tree model using the `train` data that predicts `Sex` based on all other variables. Assign this model to the variable name `m`.
   - Plot the tree (`margin=0.2` as an argument in the `plot` command helps it fit), and label the nodes (`text(m, use.n=T)`).
   - 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?

It looks good!

```r
library(rpart)
library(rpart.plot) # A good library for plotting rpart trees nicely

# Construct a tree model using the train data with a reasonable cp value.
# A starting point like 0.01 is often better than the default or 0.
# You might need to experiment with this value to get a readable tree.
m <- rpart(Sex ~ ., data=train, cp=0.01) # Using a reasonable cp

# Plot the tree using rpart.plot for better visualization
rpart.plot(m, extra=101) # extra=101 shows class probabilities
```




```r
cat("Independent variable test at the root:\n")
# Independent variable test at the root:
print(m$frame[1, "var"])
# [1] "Viscera_weight"

train_predictions_tree <- predict(m, train, type="class")
test_predictions_tree <- predict(m, test, type="class")

cat("\nConfusion Matrix (Train Data - Tree):\n")
```

```r
#
# Confusion Matrix (Train Data - Tree):
print(confusionMatrix(train$Sex, train_predictions_tree))
#   F   I   M
# F 0  88 312
# I 0 346  59
# M 0 107 341
cat("\nAccuracy (Train Data - Tree):\n")
#
# Accuracy (Train Data - Tree):
print(mean(train$Sex == train_predictions_tree))
# [1] 0.5482841

cat("\nConfusion Matrix (Test Data - Tree):\n")
#
# Confusion Matrix (Test Data - Tree):
print(confusionMatrix(test$Sex, test_predictions_tree))
#   F   I   M
# F 0 201 706
# I 0 767 170
# M 0 288 791
cat("\nAccuracy (Test Data - Tree):\n")
#
# Accuracy (Test Data - Tree):
print(mean(test$Sex == test_predictions_tree))
# [1] 0.533014
```

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? As cp decreases, train accuracy increases, test accuracy first increases then decreases, and tree complexity increases.
   - Is there a point where the model is underfitting? The model is underfitting when cp is large, resulting in a simple tree with low train and test accuracies.
   - Is there a point where the model is overfitting? The model is overfitting when cp is small, leading to a complex tree with high train accuracy but lower test accuracy.

```
# 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)))
```

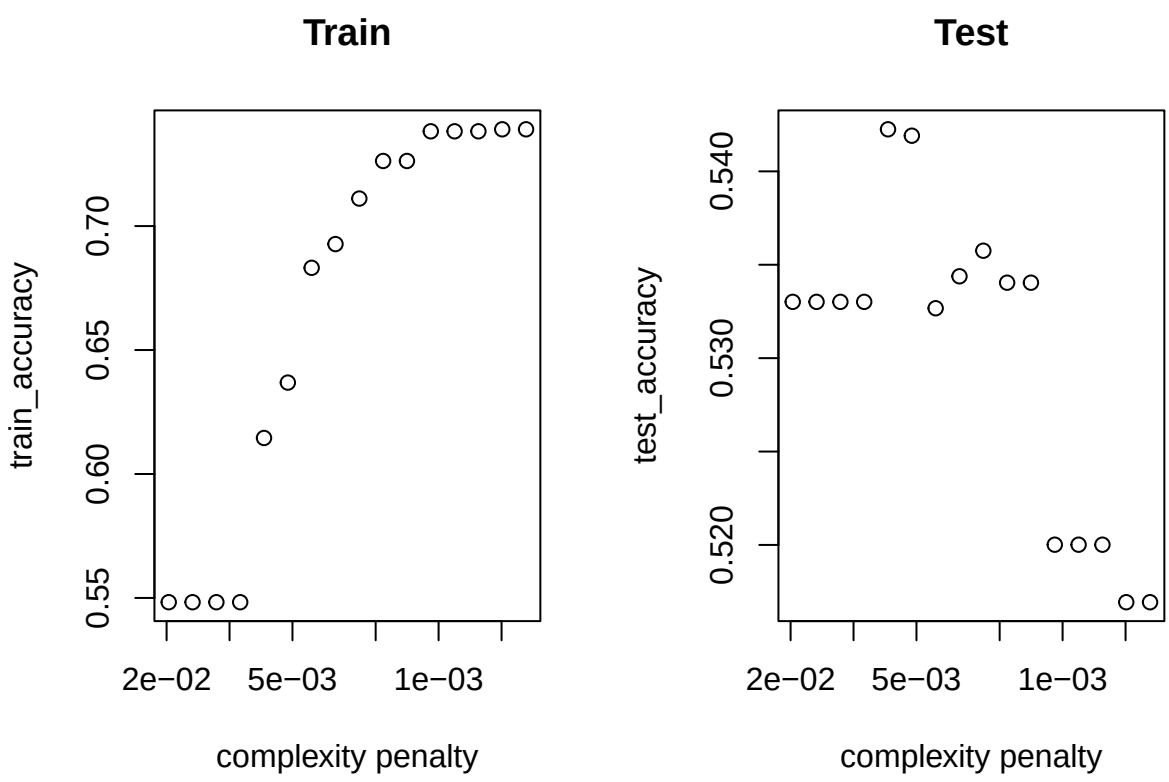
```
# declare three vectors
train_accuracy <- test_accuracy <- num_tree_nodes <- NULL
cp_vals <- 1.3 ^ seq(-15, -30) # try different (small) values of cp
range(cp_vals)
# [1] 0.00038168 0.01953663
for (cp in cp_vals) {
    # train the tree
    m <- rpart(Sex ~ ., data=train, cp=cp)

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

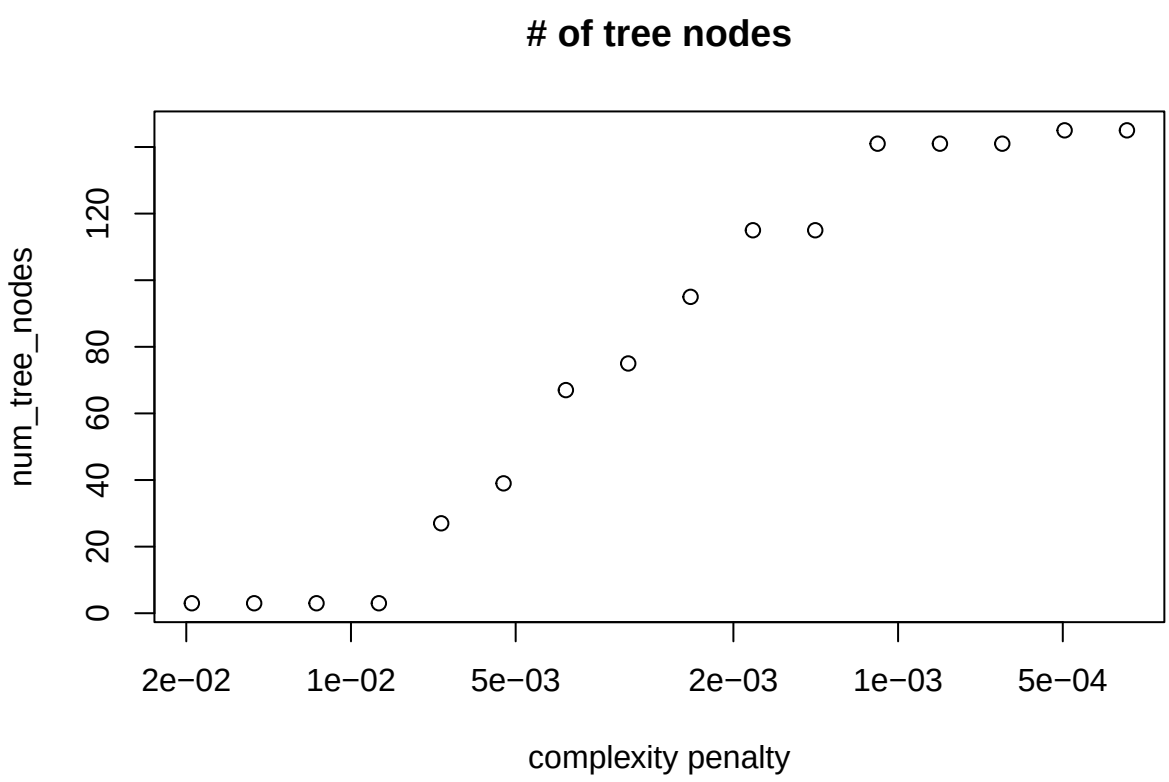
```
    test_predictions <- predict(m, test, type="class")
    test_accuracy <- append(test_accuracy, mean(test$Sex == test_predictions))

    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)))
```


# of tree nodes
num_tree_nodes
0
20
40
60
80
120
2e−02
1e−02
5e−03
2e−03
1e−03
5e−04
complexity penalty