Trees and Random Forest

Jessica Young
University of Notre Dame
Center for Social Science Research
jyoung22@nd.edu
These packages will allow you to run trees (CART) and random forest as well as use the Pima Indian data set, reshape the data, and create plots.
Starting with Pima Indian

```r
## 'data.frame': 200 obs. of 8 variables:
## $ npreg: int 5 7 5 0 0 5 3 1 3 2 ...
## $ glu : int 86 195 77 165 107 97 83 193 142 128 ...
## $ bp : int 68 70 82 76 60 76 58 50 80 78 ...
## $ skin : int 28 33 41 43 25 27 31 16 15 37 ...
## $ bmi : num 30.2 25.1 35.8 47.9 26.4 35.6 34.3 25.9 32.4 43.3 ...
## $ ped : num 0.364 0.163 0.156 0.259 0.133 ...
## $ age : int 24 55 35 26 23 52 25 24 63 31 ...
## $ type : Factor w/ 2 levels "No","Yes": 1 2 1 1 1 2 1 1 ...
```

We’ll first load in the data set, and check out what it looks like.
The *rpart* base plot is shown on the next slide and can be used by calling *plot* on the *rpart* output.

```
tree_xy = rpart(type~., data=Pima.tr)
plot(tree_xy)
text(tree_xy)
```
Tree Plots

1. glu < 123.5
2. age < 28.5
   - glu < 90
     - bp >= 68
       - ped < 0.3095
         - glu < 166
           - bmi < 28.65
             - No
             - Yes
           - Yes
         - Yes
       - Yes
     - Yes
   - Yes
   - No
- No
For a prettier version of the same plot, we can use the `prp` function from the `rpart.plot` package. Note that one of the major differences between the two plots is that the pretty plot makes the decision points much more readable by bolding and encircling the information.

```
prp(tree_xy) # pretty version
```
Before we jump into code to run, we’ll first run through 2 common functions in the caret package that we’ll be using.

- `train`
- `confusionMatrix`

Both of these functions have a wide range of capabilities.
Using the caret package, we can easily create models from a multitude of different algorithms. For more information on which models can be created, you can call `train_model_list` or view the caret online help page. In general, the function requires at minimum:

- Either a formula \((y \sim x\_\text{columns})\) or a matrix for the independent variables and a vector for the dependent variable given as \((x = \text{matrix}, y = \text{vector})\). In this run through, formulas are used.
- The method to use (i.e. model you want to create). For our purposes `rpart` and `rf` are used to make trees and random forest respectively.
- The metric used to select the optimal model. For our purposes we’re using ‘Accuracy’.
Additionally, the `train` function also includes the option of specifying the `trControl` (also known as `trainControl` function in `caret`) argument which deals with how the model is actually created computationally. This can be done inside `train` directly, or can point to an object that specifies it outside of `train` (useful if the specifications become lengthy). Using this argument, you can specify:

- **method**: the re-sampling method (i.e. bootstrapping, CV, LOOCV, etc.)
- **number**: the number of re-sampling iterations/folds
- **search**: how the training parameter is tuned ("random" or "grid")
- **sampling**: the re-sampling method to do after normal re-sampling if there is a class imbalance (i.e. ‘up’, ‘down’, etc.)
- **seeds**: the seed to use for each re-sampling iteration, useful for reproducibility
After creating a model, we can evaluate fit using the `confusionMatrix` function. As the name implies, this function creates a confusion matrix with the following arguments: * ‘data’ (predictions) * ‘reference’ (actual response values) * ‘positive’ (used to signify the minority/‘positive’ class if the data is imbalanced; this argument can be omitted).

Using this function, we can extract numerous statistics to show how well the model fits the data.
Calling `$overall` on the `confusionMatrix` output produces a list that includes: * Overall Accuracy * Kappa

Calling `$byClass` on the `confusionMatrix` output produces a list that includes: * sensitivity * specificity * precision * recall * balanced accuracy
Before we can run any analyses, we’ll first initialize 2 empty matrices to store our training results and test results in.

```r
results_tbl = matrix(0, ncol=4, nrow=50)
test_values = matrix(0, ncol=2, nrow=2)
```
First we need to store an object for the `trControl` to use. If we wanted to use 2 different controls, then we could store them both in a list to avoid creating numerous separate train control objects.

For our control, we want to be able to optimize accuracy (which may or may not be the best decision) by repeated CV (10 folds, repeated 5 times).

```r
fitControl = trainControl(method = "repeatedcv",
                           number = 10,
                           repeats = 5,
                           returnData = TRUE,
                           returnResamp = "final",
                           classProbs = TRUE)
```
Alternatively, we will could also add a train control option to use re-sampling techniques, such as over-sampling by using the code below.

```r
fitControl = trainControl(method = "repeatedcv",
                          number = 10,
                          repeats = 5,
                          returnData = TRUE,
                          returnResamp = "final",
                          sampling = 'up',
                          classProbs = TRUE)
```
Starting with regular trees, we’ll train the model and store the training errors.

```r
model_tr = train(type~., # response
                 Pima.tr, # training set
                 method = 'rpart', # regular trees
                 metric = 'Accuracy',
                 trControl = fitControl)

results_tbl[,1] = model_tr$resample[,1] # accuracy
results_tbl[,2] = model_tr$resample[,2] # kappa
```
Now we’ll test our final tree model with the test set and store those results.

```r
y_hat = predict(model_tr, Pima.te[, -8]) # predict test values

model_cm = confusionMatrix(data = y_hat, reference = Pima.te[, 8], positive = 'Yes')

test_values[1,1] = model_cm$overall[[1]] # accuracy
test_values[2,1] = model_cm$overall[[2]] # kappa
```
Using `caret` for Random Forest Optimizing Accuracy

```r
model_rf = train(type~., # response
                 Pima.tr, # training set
                 method = 'rf', # random forest
                 metric = 'Accuracy',
                 trControl = fitControl)

results_tbl[,3] = model_tr$resample[,1] # accuracy
results_tbl[,4] = model_tr$resample[,2] # kappa
```
Using *caret* for Random Forest Optimizing Accuracy

```r
y_hat = predict(model_tr, Pima.te[, -8])  # predict test values

model_cm = confusionMatrix(data = y_hat, reference = Pima.te[, 8], positive = 'Yes')

test_values[1,2] = model_cm$overall[[1]]  # accuracy

test_values[2,2] = model_cm$overall[[2]]  # kappa
```
It’s very useful to have progress updates when running code—especially if you’re looping in your code. In R, we can use the `cat` function to achieve this. Below is a snippet of code that tells R to output a sentence to the console so that we can tell when a chunk of code is done. The `\n` is useful to separate these messages with new lines in your console for easier readability. This code can be modified and inserted between each of the chunks so we can keep track of model progress.

```r
cat("Done with trees \n")
```
Creating a Boxplot

Using base R plotting abilities, we can create a comparative box plot of our results.

Unfortunately though, we’re limited in how comparative we can get. We could continually add more box plots to showcase models and the metrics they were trained with, but that would require too many box plots if we also want to compare everything across different metrics.

Instead, we can use ggplot2 to create a slicker plot.
In order to use the `ggplot` function, we need to melt the data into a format such that all of the numeric values are in one column called ‘value’, with other columns that represent the model and metric.

```r
# label columns according to model type
colnames(results_tbl) = rep(c('Tree', 'Random Forest'), each = 2)

melted_tbl = cbind('Metric' = rep(c('Accuracy', 'Kappa'), each = 50, times = 2), # metric
melt(results_tbl, # melt the data into 3 columns:
    value.name = 'value', # value
    varnames = c('Rep', 'Model')))) # rep, & model
Training Results

```r
ggplot(melted_tbl, aes(x = Model, 
                        y = value, 
                        fill = Model)) +
    facet_wrap('Metric') +
    geom_boxplot()
```
Training Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
<th>Kappa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>0.6</td>
<td>0.3</td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.6</td>
<td>0.3</td>
</tr>
</tbody>
</table>

The box plots show the distribution of accuracy and kappa for Tree and Random Forest models.
Now we can compare our training results to our actual test results. In general, the test results will be worse than the training results. Is that true here?

```r
c(\'Accuracy\', \'Kappa\')
c\('\text{Tree}', '\text{RF}'\)
print(test_values)
```

```
##        Tree       RF
## Accuracy 0.7289157 0.7650602
## Kappa    0.3675655 0.4492322
```