Tutorial 6: Classification and Regression Trees

Exercise 1: Iris Data

For the Iris data, construct a decision tree to predict the species of iris based on petal and sepal length and width. You'll find the code in the accompanying R-script. The package **rpart** is useful.

- Find the different classification rules that the tree produces.
- Predict the class of a new observation with sepal length, sepal width, petal length, petal width equal to $(6.5, 3.0, 5.2, 2.0)$.

Exercise 2: Diabetes Data

We'll use the `PimaIndiansDiabetes2` data set in the **mlbench** package for predicting the probability of being diabetes positive based on multiple clinical variables.

- Firstly, randomly split the data into a training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed so that the results can be reproduced.
- Now create a fully grown tree showing all predictor variables in the data set.
- Now use the test set to make predictions and evaluate accuracy of the model.
- Now prune the tree. Check whether the pruning has made the model substantially worse for prediction and accuracy.

Exercise 3: Boston Housing

Refer to the R script which accompanies the tutorial.

- We first load the libraries which contain good scripts for constructing trees.
- The data is found in the MASS package. The variable of interest is `medv` (median value of owner-occupied homes in $1000's.
- Carry out the diagnostics suggested in the script to get an idea of correlations between the variables and which explanatory variables may be useful.
- Now randomly split the data into training and testing (described in the script).
- Run a regression. How well does the fitted model predict new data?

Regression Tree (CART method) The `rpart` package has good routines for this. The data are recursively split into terminal nodes or leaves of the tree. To obtain a prediction for a new sample, we would follow the if-then statements defined by the tree using values of the new sample's predictors until reaching a terminal node. The model formula in the terminal node would then be used to generate the prediction. In simple (traditional) trees, the model is a simple numeric value (yes/no, or a given numeric value). In other cases, the terminal node may be defined by a more complex function of the predictors (terminal nodes have models within them).

Basic implementation is done by Growing, Examining, Pruning.

Grow a Tree To grow a traditional tree, we can use the `rpart()` function in the rpart package:

```
tree.fit <- rpart(formula, data=, method=,control=)
```

where:

- `formula` is in the format `outcome  predictor1+predictor2+predictor3+ ....`
- `data=` specifies the data frame
- `method= ''class''` for a classification tree, `''anova''` for a regression tree
- `control=` optional parameters for controlling tree growth.

For example,
`control=rpart.control(minsplit=30,cp=0.001)`
requires that the minimum number of observations in a node be 30 before attempting a split and that a split must decrease the overall lack of fit by a factor of 0.001 (cost complexity factor) before being attempted.

Examine the Tree A collection of functions helps us evaluate and examine the model.

- `printcp(tree.fit)` displays table of fits across cp (complexity parameter) values
- `rsq.rpart(tree.fit)` plots approximate R-squared and relative error for different splits (2 plots). Labels are only appropriate for the "anova" method.
- `plotcp(tree.fit)` plots the cross-validation results across cp values
- `print(tree.fit)` print results
- `summary(tree.fit)` detailed results including surrogate splits
- `plot(tree.fit)` plot decision tree
- `text(tree.fit)` label the decision tree plot

- `post(tree.fit, file=)` create postscript plot of decision tree (there may be better ways to get good looking tree plots)

First we look at what the error looks like across the range of complexity parameters (depth of tree). The command is:

```
printcp(rtree.fit) # display the results
```

(as used in the script).

A detailed summary of the tree is obtained by

```
summary(rtree.fit)
```

which gives a lot of information. We can also look at the predictors used in the tree and their relative importance in the prediction. We see specifically that `rm` (average number of rooms per dwelling) and `lstat` (lower status of the population, percent) are driving much of the prediction.

This particular tree methodology can also handle missing data. When building the tree, missing data are ignored. For each split, a variety of alternatives (called surrogate splits) are evaluated. A surrogate split is one whose results are similar to the original split actually used in the tree. If a surrogate split approximates the original split well, it can be used when the predictor data associated with the original split are not available. In practice, several surrogate splits may be saved for any particular split in the tree.

Plotting the tree may be done as follows:

```
plot(rtree.fit, uniform=TRUE,
    main="Regression Tree for Median Home Value")
```

Prune the Tree Prune back the tree to avoid overfitting the data. Hastie et al. (2008) suggest selecting the tree size associated with the numerically smallest error. That is, the size of the tree is selected by examining the error using cross-validation, specifically the minimum of the xerror column (cross-validation error) printed by `printcp( )`.

Pruning is easily done using the function `prune(fit, cp= )` by examining the cross-validated error results from printcp(), selecting the complexity parameter associated with minimum error, and placing it into the prune( ) function. Alternatively, this can be automated using

```
tree.fit$cptable[which.min(tree.fit\$cptable[,‘‘xerror’’]),‘‘CP’’].
```

In this case the pruned tree is not that much smaller than the original tree.

There are, of course other approaches for pruning. Breiman et al. (1984) suggest using the cross-validation approach and applying a one-standard-error rule on the optimization criteria for identifying the simplest tree. That is, find the smallest tree that is within one standard error of the tree with smallest absolute error, which is the leftmost cp value for which the mean lies below the horizontal line placed 1 SE above the minmum of the curve by the `minline` in the `plotcp()` function.

Test of Prediction Finally, for comparison with the regression model, we examine the R2 of the original and pruned trees.

We see here the tradeoff between "overfit" to training data and potential generalisability to new data. More formal evlauations would be done using cross-validation. But the smaller pruned tree is still doing pretty well (almost as well as the multiple regression).

Exercise 4: Boston Housing: Regression Tree

Randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

Create the regression tree. Here, the best `cp` value is the one that minimises the prediction error RMSE (root mean squared error).

The prediction error is measured by the RMSE, which corresponds to the average difference between the observed known values of the outcome and the predicted value by the model. RMSE is computed as

$$\text{RMSE} = \text{mean}((\text{observeds} - \text{predicteds})^2)\texttt{\% > \% sqrt()}.$$

The lower the RMSE, the better the model.

Plot the final tree model.

Exercise 5: Conditional inference tree

The conditional inference tree (ctree) uses significance test methods to select and split recursively the most related predictor variables to the outcome. This can limit overfitting compared to the classical rpart algorithm.

At each splitting step, the algorithm stops if there is no dependence between predictor variables and the outcome variable. Otherwise the variable that is the most associated to the outcome is selected for splitting.

The conditional tree can be easily computed using the caret workflow, which will invoke the function `ctree()` available in the **party** package.

- Use the data `PimaIndiansDiabetes2`. First split the data into training (80%) and test set (20%)
- Build conditional trees using the tuning parameters `maxdepth` and `mincriterion` for controlling the tree size. The **caret** package selects automatically the optimal tuning values for your data, but here we'll specify `maxdepth` and `mincriterion`.
- Now make predictions using the test data.