FIT2086 Studio 9
Supervised Machine Learning Methods

Daniel F. Schmidt

September 12, 2018

Contents

1 Introduction

Studio 8 introduces you to several supervised machine learning techniques; in particular, you will look at using decision trees for regression and classification, as well as k nearest neighbours methods. To complete this Studio, you will need to install two packages: `tree` and `kknn` in R.

During your Studio session, your demonstrator will go through the answers with you, both on the board and on the projector as appropriate. Any questions you do not complete during the session should be completed out of class before the next Studio. Complete solutions will be released on the Friday after your Studio.

2 Decision Trees

In the first part of this Studio we will look at how to learn a basic decision trees from data, how to visualise/interpret the tree, and how to make predictions. We we will look at both continuous targets (regression trees) as well as categorical targets (classification trees). Begin by ensuring that the `tree` package is loaded.

1. Load the `diabetes.train.csv` and `diabetes.test.csv` data into R. Use `summary()` to inspect your training data; you will see that it has 10 predictors, `AGE`, `SEX`, `BMI`, `BP` (blood pressure) and six blood serum measurements `S1` through to `S6`; it also has a target variable `Y`, which is a measure of diabetes progression over a fixed period of time. The higher this value, the worse the diabetes progression.

2. Let us fit a decision tree to our training data. To do this, use

   ```
   tree.diabetes = tree(Y ~ ., diabetes.train)
   ```

   which fits a decision tree to the data using some basic heuristics to decide when to stop growing the tree. We can use the `summary()` function on the tree

   ```
   summary(tree.diabetes)
   ```

   This tells us some basic statistics of the tree. What variables have been used by the decision tree to predict the output?

3. We can also explore the relationships between the diabetes progression outcome variable (`Y`) and the predictor variables that were used by the decision tree package. We will first do this by examining the decision tree in the console:

   ```
   tree.diabetes
   ```

   This displays the tree in text form. The asterisks "`*`" denote the terminal (leaf) nodes of the tree, and the nodes without asterisks are split nodes; the information contains which variables are split on, what the splits are, and for each leaf node, what the predicted value of `Y` is. How many leaf nodes are there in this tree?

4. The output of the above command all the information related to the tree we have learned but is hard to understand. It is easier to visualise the decision tree by using the `plot()` function to get a graphical representation of the relationships:

   ```
   plot(tree.diabetes)
   text(tree.diabetes, digits=3)
   ```

   This displays the tree, along with the various decision rules at each split node, and the predicted value of `Y` at each leaf. Using this information:

(a) What is the estimated average diabetes progression for individuals with `BMI` = 28.0, blood pressure (`BP`) = 96 and `S6` = 110?

(b) What is the estimated average diabetes progression for individuals with `BMI` = 20.1, `S5` = 4.7 and `S3` = 38?

(c) Find the characteristics of the individuals with the worst (highest) predicted average diabetes progression.

5. We can now test to see how well this tree predicts onto future data. We can use the `predict()` function to get predictions for new data, and then calculate the root-mean squared error (RMSE):

```
sqrt(mean((predict(tree.diabetes, diabetes.test) - diabetes.test$Y)^2))
```

How can we interpret this score?

6. The `tree` package provides the ability to use cross-validation to try and "prune" the tree down, removing extra predictors and simplifying the tree without damaging the predictions too much (and potentially improving them). The code is a little involved, so I have included a wrapper function in the file `wrappers.R`; source this file to load the wrapper functions into memory. Then, to perform CV

```
cv = learn.tree.cv(tree.diabetes,nfolds=10,m=100)
```

The `nfolds` parameter tells the code how many different ways to break up the data, and a value of 10 is usually fine. The `m` parameter tells the code how many times to repeat the cross-validation process (randomly dividing the data up, training on some of the data, testing on the remaining data) – the higher this number is, the less the trees found by CV will vary from run to run, as it reduces the random variability in the cross-validation scores. The `cv` object returned by `learn.tree.cv()` contains three entries. The `cv$cv` object contains the statistics of the cross-validation. We can visual this using:

```
plot(cv$cv)
```

This shows the cross-validation score against the tree size (bottom x-axis) and "complexity" (top x-axis). The `cv$best.k` entry is the best value of the complexity parameter for our dataset, as estimated by CV, and can be passed to `prune.tree()` to prune our tree down.

7. The `cv$best.tree` object contains the pruned tree, using `cv$best.k` as the pruning complexity parameter. Plot this tree, and compare it to the previous tree `tree.diabetes`. How do they compare?

(a) Has cross-validation removed any predictor variables from the original tree `tree.diabetes`?

(b) What are the characteristics that predict the worst diabetes progression in this new tree?

(c) What is the RMSE for this new tree `cv$best.tree` on the test data?

8. We can now compare the performance of our decision tree to a standard linear model. Use the `glmnet` package to fit a linear model using the lasso, and calculate the RMSE for the fitted model:

```
lasso.fit = cv.glmnet.f(Y ~ ., data=diabetes.train)
glmnet.tidy.coef(lasso.fit)
sqrt(mean((predict.glmnet.f(lasso.fit,diabetes.test)-diabetes.test$Y)^2))
```

How does the linear model compare in terms of which predictors it has chosen to use with the tree selected by CV?

3 Random Forests

A random forest is a collection of classification or regression trees that are grown by controlled, random splitting. Once grown, all the trees in a random forest are used to make predictions and to determine which predictors are associated with the outcome variable. However, the relationships between the predictors and the target are much more opaque than for a decision tree or linear model. In order to use random forests in R you must install and load the `randomForest` package.

1. First, use R to learn a random forest from the `diabetes.train` data set:

   ```
   rf.diabetes = randomForest(Y ~ ., data=diabetes.train)
   ```

   This trains a forest of decision trees on our data.

2. Unlike a single decision tree, a random forest is difficult to visualise and interpret as it consists of many hundreds or thousands of trees. After learning the random forest from the data, we can inspect the model by typing:

   ```
   rf.diabetes
   ```

   This returns some basic information about the model, such as the percentage of variance explained by the tree (roughly equivalent to $100\,R^2$).

3. To see how well our random forest predicts onto our testing data we can use the `predict()` function and calculate RMSE:

   ```
   sqrt(mean((predict(rf.diabetes, diabetes.test) - diabetes.test$Y)^2))
   ```

   We can see that the random forest performs quite a bit better than our single best decision tree, and is basically the same as the linear model in this case.

4. So far we have been run the random forest package using the default settings for all parameters. Although the package `randomForest` has many interesting user-settable options (see the help for more details), the following three options are most useful for common use:

   - `ntree`: Specifies the number of trees to grow. This should not be set to too small a number, to ensure that every input row gets predicted at least a few times (Default: 500)
   - `importance`: Should importance of predictors be computed? (Default: `FALSE`)

   Let's explore how we can use these options when analysing our diabetes data set.

   ```
   rf.diabetes = randomForest(Y ~ ., data=diabetes.train, importance=TRUE, ntree=5000)
   ```

   The number of trees in this example is set to $5,000$. In general, using more trees leads to improvements in prediction error. However, the computation complexity of the algorithm grows with the number of trees which means large forests can take a long time to learn and use for prediction. Calculate RMSE on the test data for this new random forest.

5. The option `importance` tells the random forest package that we wish to rank our predictor variables in terms of their strength of association with the outcome. To view the final ranking, we need to run:

   ```
   round( importance( rf.diabetes ), 2)
   ```

   The output of the command contains several columns which are different measures of variable importance. When used for continuous outcome variables, the command produces `%IncMSE` which corresponds to the estimated increase in mean square prediction error that occurs if a particular exposure variable is omitted. Which variables seem to be the most important using this measure?

4 k-Nearest Neighbours

The last supervised machine learning technique we will examine are called k-Nearest Neighbours (kNN) classifiers. A big advantage of kNN based methods is their great flexibility, speed and lack of assumptions. The obvious disadvantage is that like random forests they are difficult to interpret; they also suffer from the fact that selecting important predictors is not naturally handled by most packages. To install and load the `kknn` package.

1. Once again, let's look at the diabetes data. Using the default settings, a kNN can be used to make predictions with the code below:

   ```
   ytest.hat = fitted( kknn(Y ~ ., diabetes.train, diabetes.test) )
   ```

   The fitted command tells the R package to use the k-NN algorithm to make predictions about future (test) data. Calculate the RMSE on the test data using:

   ```
   sqrt(mean((ytest.hat - diabetes.test$Y)^2))
   ```

   How does this compare to the prediction errors achieved by the linear model, decision tree and random forests?

   The kNN approach does not actually build a model to describe the data. Instead, to make a prediction about an individuals outcome, a k-NN finds the k closest individuals (in terms of predictor variables) in the training data to the individual in question and uses a combination of their target variables to make predictions on future data. Practically, this means that when using the `kknn` package, we need to specify both the training and test dataset whenever we need to make predictions using `fitted()`. A downside of this model-free approach is that the larger our training data, the slower it becomes to produce predictions onto new data.

2. As with random forests the k-NN procedure has several options that control the behaviour of the algorithm. The most important of these are `k` and `kernel`. The `k` option controls the size of neighbourhood used when making predictions, i.e., how many individuals from the training data are used to form a prediction on the test data. The `kernel` option determines how the individuals in the neighbourhood are combined together when making predictions (see Lecture 9 for details on these types of parameters). The best values for these parameters will depend on the particular dataset, and as with lasso hyperparameters, or the size of a tree, can be automatically be chosen using cross-validation. To do this, use the following (provided wrapper function):

   ```
   kernels = c("rectangular","triangular","epanechnikov","gaussian","rank","optimal")
   knn = train.kknn(Y ~ ., data = diabetes.train, kmax=25, kernel=kernels)
   ytest.hat = fitted( kknn(Y ~ ., diabetes.train, diabetes.test,
              kernel = knn$best.parameters$kernel, k = knn$best.parameters$k) )
   ```

   The above code uses `train.kknn()` to try a combination of different `k` values (1 to `kmax`) and different kernels, and stores all these inside the `knn` object. We can then use the k and kernel nominated as best by cross-validation in conjunction with the `fitted()` command, to get improved predictions.

   Calculate the RMSE on the testing data for the predictions made by the k-NN method with the k and kernel chosen by cross-validation. How do they compare to the RMSE scores obtained by the linear model, decision tree and random forest?

5 Additional Questions

You have now learned how to use the `tree`, `randomForest` and `kknn` packages to build machine learning models for predicting, and in the case of trees, learning which variables are important. You can use this newfound knowledge apply these models to some of the datasets we have examined over the last few weeks:

All three methods can be applied to categorical target variables (i.e., classification). The only changes you need to make are when computing predictions. When using decision trees, you the predictions are the probabilities of the target being in each of the classes; for our binary classification problems, you need to take the second column as the probabilities to pass to `my.prediction.stats()`, i.e.,

```
my.prediction.stats(my.pred.stats(predict(tree, pima.test)[,2], pima.test$DIABETES)
```

To produce probabilities of classification for random forests you need to use `predict()` with the `type` argument set appropriately, i.e. if `rf.pima` is our random forest trained on the Pima indians data, then we can predict on future data using

```
predict(rf.pima,pima.test,type="prob")[,2]
```

The k-NN package can only produce the best guesses at our target classes, so we cannot compute AUC scores or log-loss. Instead, if `ytest.hat` are predictions then can compute classification accuracy using something like

```
mean(yhat.test == pima.test$DIABETES)*100
```

Using this, do the following:

1. Explore the genetic data we examined last week using decision trees, random forests and k-NN methods. What variables do the tree-based methods select?

2. Explore the Pima indians data using these three methods. How do the predictions compare to the logistic regression based methods we examined? What variables do the tree-based methods select?