MATH5743M: Statistical Learning

Dr Seppo Virtanen, School of Mathematics, University of Leeds

Semester 2: 2022

Week 9: Random Forests

Week 8 introduced the idea of collective intelligence in statistical modelling. We saw various instances in which a combination of different statistical models (including human brains!) can do better at making predictions than a single model alone.

To demonstrate this we investigated the concept of Bootstrap Aggregating (BAGGING), where we used bootstrapping to generate lots of different data sets from which to train many different versions of the same model. We looked at this in practice using logistic regression models and saw that BAGGED models were able to fit better than a single logistic regression model when the underlying reality did not exactly fit the assumptions of the logistic model.

This week we will be looking at a tool that has taken the idea of BAGGING even further, and exploits collective wisdom to generate very accurate predictive models from hundreds or thousands of individually poor models. This tool is called the Random Forest. A Random Forest is an ensemble of *decision trees* (many trees make a forest).

BAGGING of decision trees: Random Forests

A fitted decision tree is an easily understood statistical model. You start from the top of the tree and progress down, making clearly defined decisions along the way based on the input values, until you reach a leaf node. Each leaf specifies either a single class for your output, or the probability that your output belongs to one of many possible classes. Fitting these models is also relatively straightforward, as we have seen, and fits well with how humans think about data in real life: spotting dividing lines that separate one type of object from another.

However, although these models are nice to use because of their simplicity and interpretability, they suffer from a crucial problem: they are often unstable and produce poor predictions for all but the simplest data sets. Small changes to the data can produce different optimal splits in the early parts of the decision tree, leading to completely different trees being produced from very similar data. As such it doesn't make sense to see a fitted decision tree as *the* tree to describe a given data set, but one of many that could do so equally well. Moreover, splitting the data into two parts at each split is a crude way to identify patterns, meaning that any individual tree is often poor at making predictions.

So we are left with the possibility of many equally valid, generally poor classifiers based on a given data set. This doesn't sound great, but if we return to what we learnt before, we remember that many poor classifiers can make excellent predictions when used *together* (recall Condorcet's Theorem). Remember as well that one of the conditions of collective intelligence was that the different agents or models would be *diverse*, i.e. different from each other. In the case of decision trees, this means that their instability, which might be problematic on their own, can be very useful when used together, since each tree will be quite different from the others.

All of these factors have motivated researchers to work with ensembles of decision trees, now formally known as Random Forests. The term 'forest' tells us that these are based on ensembles of trees. The use of the word 'random' is because they make use of random number generation, through the use of bootstrapping (recall that bootstrapping involves taking random samples from the original data).

The Random Forest algorithm

Creating a Random Forest works by applying the ideas we learnt for BAGGING to decision trees.

1. Create N bootstrap samples from the original data set
2. Fit one decision tree model for each bootstrap sample
3. Average over the predictions of all trees when making predictions of new data. Settle ties at random (for classification).

In practice there are a variety of additional 'tweaks' that users can apply to make Random Forests more effective. For example, trees are often learnt by considering splits over a small, randomly selected subset of inputs at each stage, making the trees more diverse. Nonetheless, the three steps presented above capture the essence of the Random Forest model.

Random Forests in R

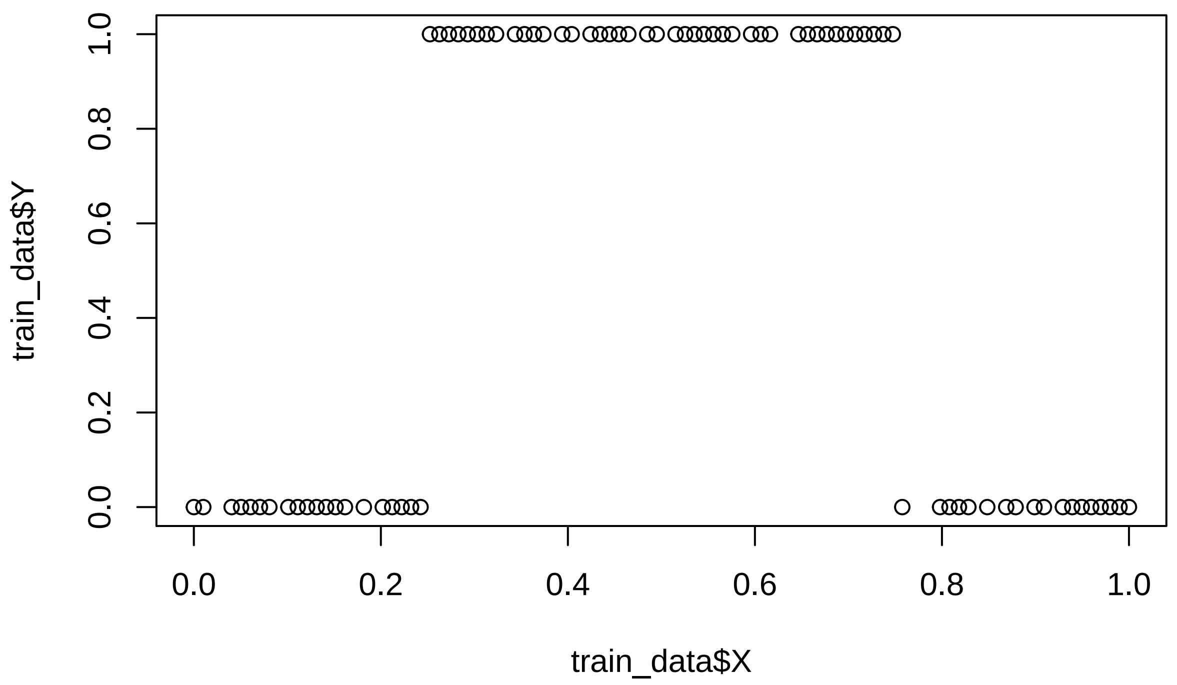
The randomForest package in R provides a very complete set of tools for implementing and using Random Forest models. To install it simply use the following command:

```r
install.packages("randomForest")
```

Use of the Random Forest package follows a very similar pattern to our previous use of glm. There is a function **randomForest** that is the equivalent of **glm**, for specifying and fitting a model, and a **predict** method that operates on a fitted model to make predictions for new data.

Let's follow on from our previous look at BAGGING of logistic models by considering data of the same form (I've set the random seed here so that the data will actually be exactly the same):

```r
myfunction <- function(x){0+1*(x>0.25 & x < 0.75)}
X = seq(0,1, length.out=100)
P = myfunction(X)
Y = as.numeric(runif(100)<P)
mydata = data.frame(Y, X)
test_idx = sample(dim(mydata)[1], 20)
test_data = mydata[test_idx, ]
train_data = mydata[-test_idx, ]
plot(train_data$X, train_data$Y)
```



Again, to reiterate from our previous look at this data, it is not well described by a logistic model, because the probability function is not monotonic. As we saw before, a BAGGED ensemble of logistic models can fit this data better than one logistic model alone, but the fit was still quite poor. Now we will try using a Random Forest instead. Note how the syntax below follows the same form as for **glm**. However, one extra detail is that we need to tell the Random Forest that we want to do *classification*, since regression is the default mode for numerical outputs. To do this we need to redefine the output as a *factor*: a variable that corresponds to a discrete set of classes

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
train_data$Y = as.factor(train_data$Y)
myForest = randomForest(Y ~ X, data=train_data)
```

Now we can use the fitted model to predict the test data. Note that we have to tell the **predict** function that we want the class probabilities, rather than the predicted classes. This is different to the 'type' we had to use to get probabilities from **glm** when doing logistic regression; sadly life isn't perfect!

```
prediction = predict(myForest, newdata=test_data, type="prob")
head(prediction)
```

```
##    0 1
## 39 0 1
## 51 0 1
## 89 1 0
## 18 1 0
## 3  1 0
## 86 1 0
```

The output is a 2-column matrix giving the probabilities of each data point belonging to class 0 or 1. Looking at the column headers we see that the second column is the probability for Y to be equal to 1, so only need the second column (i.e. the probability of being in class 1), to evaluate the quality our prediction.

```
prediction = prediction[, 2]
```

Important!: Now here we need to introduce a certain problem with these probability predictions. The way the Random Forest package predicts the probability for a class is to send the input data through each decision tree in the forest, and count how many of those trees predict each possible class for the output. The probability for the output to be in class '1' is then given as the proportion of trees which 'voted' for that class. This is a reasonable way to assign probabilities, but it can lead to problems when all the trees agree (i.e. they all predict the same class). In this case we will get predicted class probabilities that are either zero or one. This is unrealistic - we can never be 100% sure in our predictions! And it will cause problems for us if the output in the test data set is different from the prediction, as the predictive log-likelihood will be -INF!

This problem arises because we only have a finite number of trees. If we used more trees we would eventually find one that disagreed with the consensus. So to fix our problem we can use a simple 'trick'. We imagine that we create two more trees, one of which predicts class '1', the other of which predicts class '0'. This way there is always at least one tree that disagrees with the majority. To implement this trick we need to know how many trees we used originally, and then rescale the predicted probabilities. If you don't know how many trees you used (perhaps because you allowed **randomForest** to use the default number), you can find out through the **ntree** property:

```
number_of_trees = myForest$ntree
print(number_of_trees)
```

```
## [1] 500
```

Then we can rescale the predictions by considering the number of trees predicting class '1' (including our one additional tree now), and the total number of trees (including both our added trees):

```
prediction = (number_of_trees*prediction + 1)/(number_of_trees + 2)
```

For those of you with a keen eye for how everything fits together and a good memory of your earlier modules, you may notice this is exactly the same as Bayesian inference for a binomial probability with a conjugate Beta-distribution prior!

Now, armed with our adjusted prediction, we can calculate the predictive log-likelihood of the test data to evaluate how good our prediction actually is:

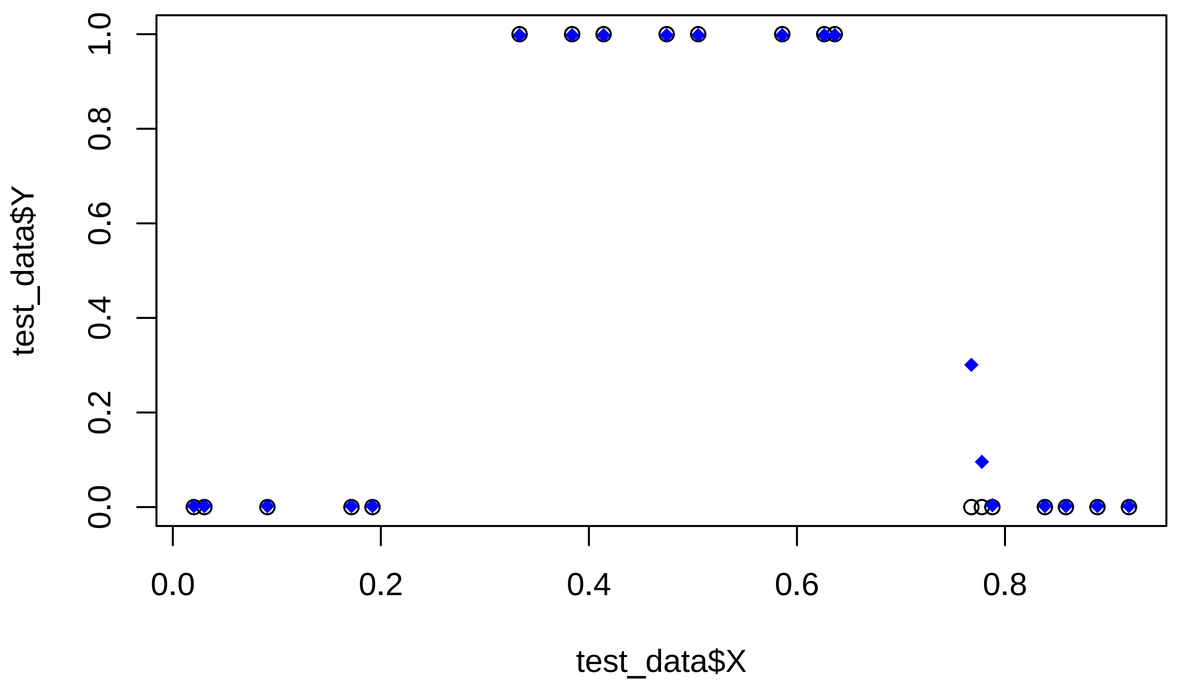
```
testLL = sum(log(prediction[which(test_data$Y==1)])) + sum(log(1-prediction[which(test_data$Y==0)]))
print(testLL)
```

```
## [1] -0.4962072
```

If you look back to the example from the lecture on bootstrapping logistic models, you can compare this predictive accuracy to that - you should see that this model makes *much* better predictions. In fact, the log-likelihood here is very close to 0, which is what we would get if every prediction was perfect (i.e. we predicted a probability of 1 when an example was in class 1, and 0 when an example was in class 0). Therefore this model is doing about as well as we can ever expect to do.

We can also visualise these predictions to see by eye how well we did:

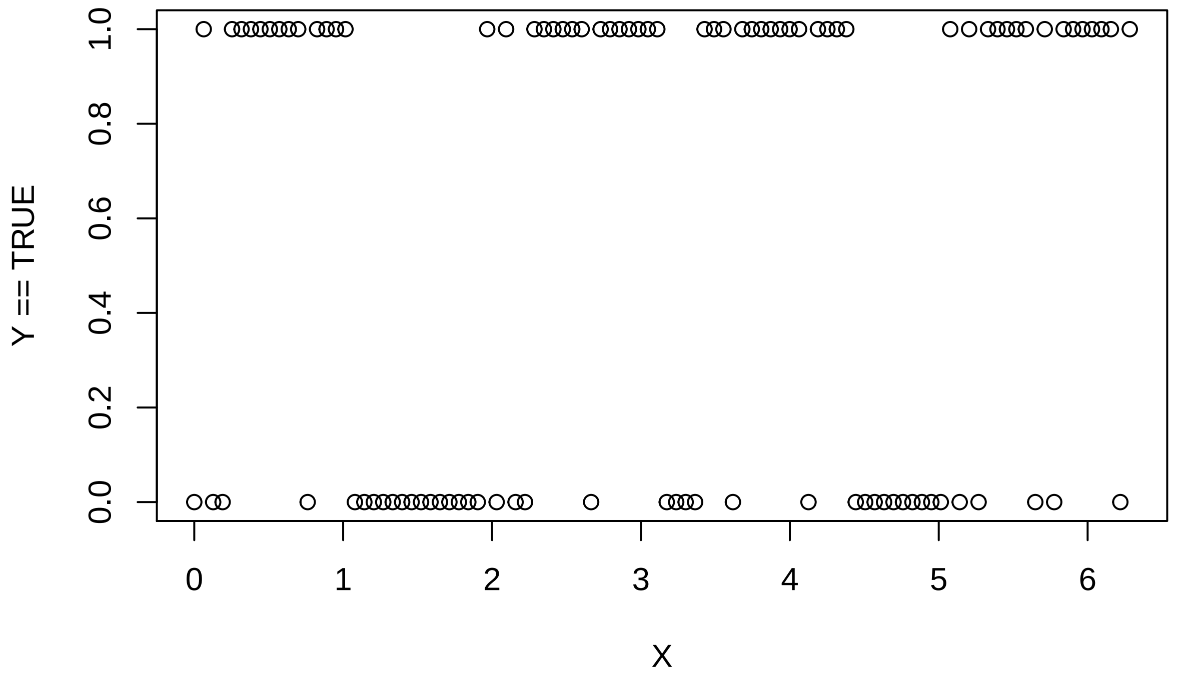
```
plot(test_data$X, test_data$Y)
lines(test_data$X, prediction, col='blue', type='p', pch=18)
```



An almost perfect fit.

Now this is a very simple example, and clearly we've seen that Random Forests are a powerful model, so let's try something a bit more interesting. Lets see how well they do at fitting a complicated function - where the probability for class '1' is a product a multiple sine waves:

```
X = seq(0, 2*pi, length.out=100)
P = 0.5*(sin(X)*sin(3*X) + 1)
Y = as.factor(runif(100) < P)
wavedata = data.frame(Y, X) #Make Y a factor to ensure random forest performs classification
plot(X, Y==TRUE)
```

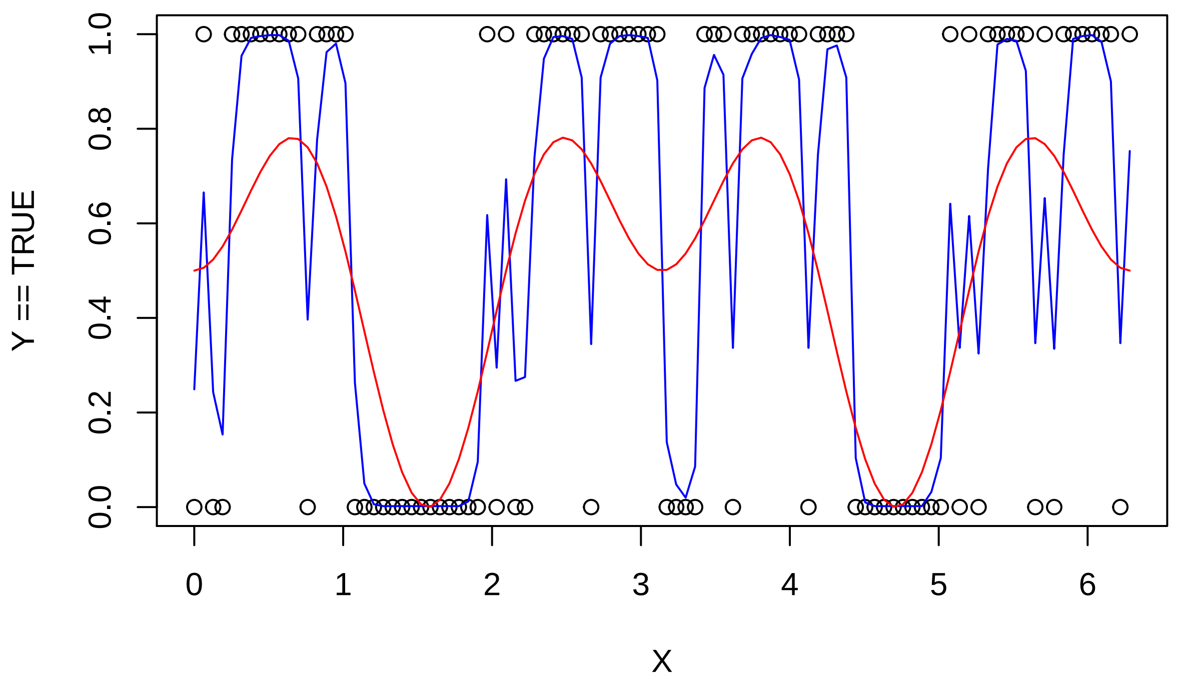


We'll train the model on this data, and see what it thinks the underlying probability function looks like, by making a prediction on a new set of inputs that are actually the same as the training inputs. Then we'll compare this to what we know is the real underlying function: $y = \sin(x)\sin(3x)$:

```
waveForest = randomForest(Y~X, data = wavedata)
prediction = predict(waveForest, newdata = wavedata, type='prob')

#Extract second colum as P(TRUE) and adjust using numbe rof trees
prediction = prediction[,2]
number_of_trees=waveForest$ntree
prediction = (number_of_trees*prediction + 1)/(number_of_trees + 2)

real_f = 0.5*(sin(wavedata$X)*sin(3*wavedata$X)+1)
plot(X, Y==TRUE)
lines(wavedata$X, prediction, col='blue')
lines(wavedata$X, real_f, col='red')
legend(x=2.5, y = -0.5, legend=c('Data', 'Inferred', 'Real'),
       col= c('black', 'blue', 'red'), pch=c(1, NA, NA), lty=c(NA, 1, 1))
```



You can see from the comparison that the inferred function in blue follows the general pattern of the real probability in red, which would be very difficult to model with a technique like logistic regression unless you had a very good idea what the red line was likely to be *in advance*. However, because decision trees do not fit explicit functions, the inferred function is not as smooth as you would get using a polynomial regression or a similar technique; small changes in x sometimes produce sharp changes in y as a few decision trees pass their critical splitting values. Because the Random Forest is not constrained to follow a functional form it does not have to be smooth at all points, and therefore tends to fit a little more of the noisy variation of the training data than some other methods. In addition, the Random Forest seems to overshoot and is overconfident, making predictions that are closer to zero and one than the red line. To test whether this effect of the noise makes the Random Forest model a worse predictor than the alternatives you should use the *model selection techniques* we learnt about previously to compare and select between alternative models. Once you are able to get the prediction from a model in terms of the *probability* you can fairly compare not just different possible input factors within a type of model, but also completely different types of models as well, via cross-validation.

We've just seen that Random Forests can fit very complex functions with minimal effort on the part of the analyst. If they're so powerful, why don't we use them for everything?

The main reason is the constant tension between a model that *fits* well and one that is *interpretable*. Remember when we did linear and logistic regression, we could get summaries of the model telling us exactly how important each input was (the regression coefficients), what direction of effect these inputs produced (the sign of the coefficient) and how uncertain these effects were (confidence intervals). By contrast, with a Random Forest we have a model that does very well numerically but provides no easily read information about the model's internal function. We may find that some set of inputs is able to predict an output very well, but we don't know how or why. (**NB: I strongly advise you not to trust the Variable Importance metrics that the Random Forest package provides - these are not a statistically sound way to infer which variables are genuinely useful, and are calculated in some complex ways that are not immediately apparent to the user.**)

For this reason we should generally use the simplest model we can that still fits the data well. How well? That is something you need to judge for yourself as an analyst, with whoever asks you to do the analysis! For example, if you see data with a very clear monotonic trend you should use linear or logistic regression (for regression or classification problems). If you see something that looks a bit more complicated you might use polynomial regression. For very complex problems, or ones where prediction accuracy matters more than interpretation, you can consider a Random Forest.

Remember the quote, often attributed to Albert Einstein: 'Everything should be made as simple as possible, but not simpler'.