Introduction to Computational BioStatistics with R: classification I

Erik Spence

SciNet HPC Consortium

31 October 2024

Today's slides

To find today's slides, go to the "Introduction to Computational BioStatistics with R" page, under Lectures, "Classification I".

https://scinet.courses/1353

Today's class

Today we will visit the following topics:

- Classification algorithms, in general.
- Decision trees.
- kNN.

Classification

Classification is similar to regression, in a sense:

- You fit a model to data with known answers ($y = f(x_1, x_2, x_3, ...)$).
- You use the model to make predictions about new data.

But what do you do if the labels (y) are discrete? How do you deal with that?

- Data point y is either in category 1 or 2.
- You don't get points for putting y in category 1.5.

Classification algorithms are used to create models for separating data into known categories.

Classification problems

Some classic classification problems:

- Bioinformatics - classifying proteins according to function.
- Medical diagnosis.
- Image processing:
  - what objects exist in an image?
  - hand-written text analysis.
- Text categorization:
  - Spam filtering
  - Sentiment analysis: is this tweet positive or negative?
- Language recognition.
- Fraud detection.

Input variables can be continuous, discrete, or both.

Classification approaches

There are lots of classification approaches which one might use.

- Decision trees: analyze the features of the data and make 'decisions' about how to 'split' the data into uniform groups.
- Logistic regression: like linear regression, but now we fit a "yes/no" function to the data.
- Naive Bayes: a type of probabilistic analysis.
- kNN: k Nearest Neighbours; use the k nearest neighbours to a data point to predict the category of a new data point.
- Support Vector Machines: essentially a linear model of the data, used to separate groups.
- Neural networks: a weird algorithmic approach to using functions to categorize data.

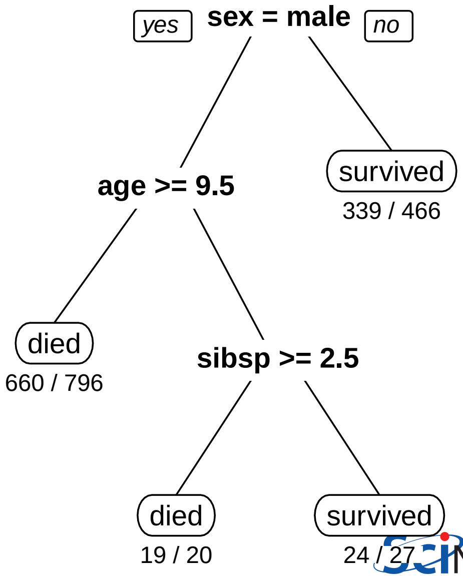
There isn't time to cover all of these. Today we'll cover Decision Trees and kNN.

Decision Trees

A Decision Tree is a structure which classifies an input based on a number of binary decisions.

It splits the data set based on one of the p "features" of the data.

"Features" are the independent variables associated with the data $(x_1, x_2, ..., x_p)$.



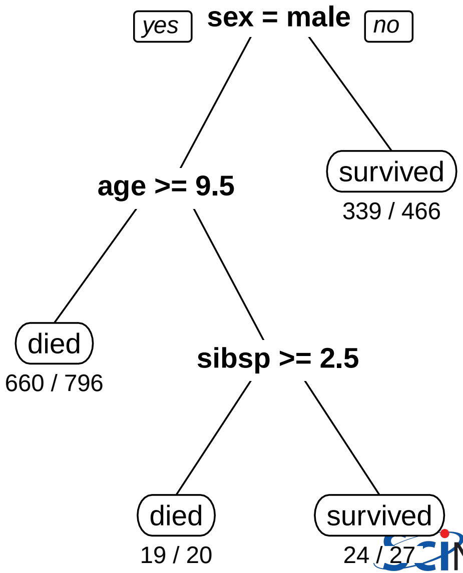
Decision Trees, continued

Data can be split based on

- discrete data (”if category == A”) or,
- continuous data (”if height < 1.5m”)

The goal of developing a decision tree is to determine when and where and how to split the data, so as to maximize the ’purity’ of the resulting sub-data set.

A good decision tree will have ”leaves” (ends of the tree) which are as pure as possible.



Muppet data set

Consider this data set. The goal is to create a decision tree algorithm which classifies Muppet characters as Sesame Street (SS) or not.

Name	colour	cloths	eye brows	ball nose	SS
Kermit	green	FALSE	FALSE	FALSE	FALSE
Grover	blue	FALSE	FALSE	TRUE	TRUE
Bert	yellow	TRUE	TRUE	TRUE	TRUE
Sam the Eagle	blue	FALSE	TRUE	FALSE	FALSE
Oscar	green	FALSE	TRUE	FALSE	TRUE
Miss Piggy	tan	TRUE	FALSE	FALSE	FALSE

Given this data, how does your algorithm do on this data?

Name	colour	cloths	eye brows	ball nose	SS
Gonzo	blue	TRUE	FALSE	FALSE	??
Big Bird	yellow	FALSE	FALSE	FALSE	??

Splitting algorithms

Consider the following two possible first splits:

- Split based on 'ball nose'.
    - ball nose == TRUE: we get Grover, Bert (SS).
    - ball nose == FALSE: we get Kermit, Sam the Eagle, Miss Piggy (not SS), Oscar (SS).
- Split based on 'cloths'.
    - cloths == TRUE: we get Bert (SS), Miss Piggy (not SS).
    - cloths == FALSE: we get Kermit, Sam the Eagle (not SS), Grover, Oscar (SS).

There's a sense in which the ball nose split is clearly better. It leads to two groups, one of which is totally Sesame Street, and the other which is mostly not.

The other choice gives you two groups which are just as heterogeneous as the original data.

Splitting algorithms, continued

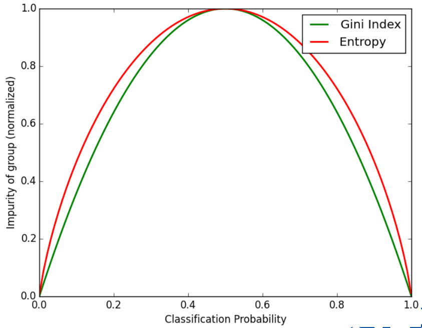
Algorithms which split the data rank possible splits based on increasing 'purity' of the two subgroups it generates.

Consider the probability p that a member of one of the labels is in a given target category. Two common measures for the 'impurity' of the generated groups are given by

Gini index: $\sum p(1-p)$

Entropy: $-\sum[p \ln p + (1-p)\ln(1-p)]$

Where the sum is over all labels and possible values in the given target category. A perfect Gini index is an impurity of 0, or a probability of 0 or 1.



Splitting algorithms, continued more

So how do these algorithms proceed?

- While every data point is not in a pure sub-tree:
  - For each feature which we haven't yet split upon, for the data remaining in the sub-tree, consider a split:
    - If the feature is categorical, consider all values, split by value and measure the impurity of the resulting subgroups.
    - If the feature is continuous, use line optimization to choose the best point at which to split, keeping track of the impurity at that point.
  - Choose the split which maximizes the change in the impurity (smallest impurity value), and split the data.

Decision trees in R

Let's use an R package to build a decision tree. We'll use the Iris data set.

- The data consists as four measurements of 150 wild irises of 3 species.
- It's a classic classification problem.
- It's one of the data sets which comes with R.
- We first randomly split the iris data set, 70/30, into training and test data sets.

```
>
> str(iris)
'data.frame':  150 obs.  of 5 variables:
 $Sepal.Length: num 5.1 4.9 4.7 4.6 5 ...
 $Sepal.Width : num 3.5 3 3.2 3.1 3.6 ...
 $Petal.Length: num 1.4 1.4 1.3 1.5 1.4 ...
 $Petal.Width : num 0.2 0.2 0.2 0.2 0.2 ...
 $Species     : Factor w/ 3 levels ...
>
> ind <- sample(c(TRUE, FALSE), nrow(iris),
+   replace = T, prob = c(0.7, 0.3))
> train.d <- iris[ind,]
> test.d <- iris[!ind,]
>
```

http://en.wikipedia.org/wiki/Iris_flower_data_set

Training versus Testing

In general, we get our data, and that's it. We don't have the luxury of generating more data on a whim.

We would like to do out-of-sample testing of whatever model we generate, to see how it does against new data. But we don't have any new data.

The solution is to hold out some of the original data. Most of the data is used for training the model, the rest is used for testing it. These data should be chosen randomly, as in the previous slide.

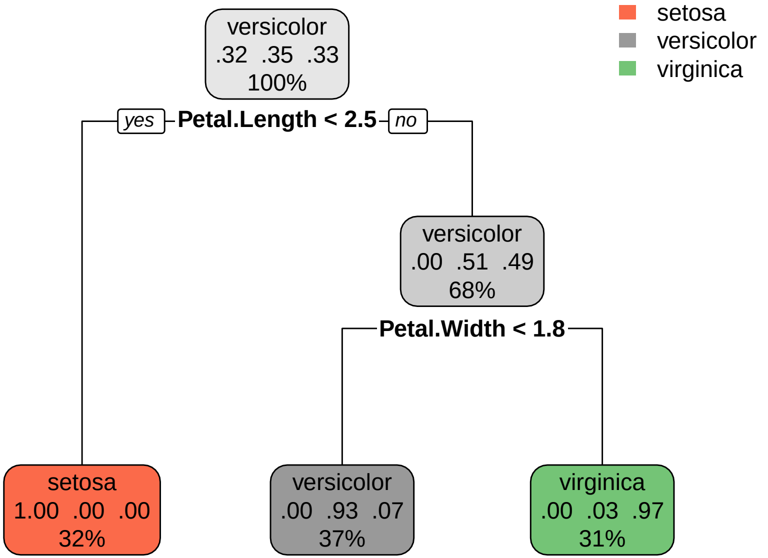
R analysis, an example, continued

Now that the data's split up, we're ready to generate the tree.

- Load the 'rpart' and 'rpart.plot' (non-standard) libraries.
- Create our formula (Note the simpler syntax which you can use).
- Generate the decision tree.
- Plot the result.

```
>
> library(rpart)
> library(rpart.plot)
>
> f <- Species ~ Sepal.Length + Sepal.Width +
+   Petal.Length + Petal.Width
>
> f <- Species ~ .     # same as above
>
> iris.tree <- rpart(f, data = train.d)
>
> rpart.plot(iris.tree)
>
```

Our decision tree



Confusion matrix

How do you determine the effectiveness of a classifier? You can count the number incorrectly classified, but this doesn't give you much information you can use to improve the result.

The 'Confusion Matrix', tells you which misclassifications happened. Traditionally, 'true' classifications are on the rows, and predictions are on the columns.

```
> pred <- predict(iris.tree, type = 'class')
>
> sum(train.d$Species == pred) / nrow(train.d)
[1] 0.9611650
>
> table(train.d$Species, pred)
            setosa versicolor virginica
 setosa         36          0         0
 versicolor      0         35         1
 virginica       0          3        28
```

R analysis, an example, continued more

Ok, but how does the decision tree do on the test data?

- Test the built tree against the test data.
- Print out the table of results.
- Not bad!

```
>
> testPred <- predict(iris.tree,
+   newdata = test.d, type = 'class')
>
> sum(test.d$Species == testPred) / nrow(test.d)
[1] 0.957447
>
> table(test.d$Species, testPred)
            setosa versicolor virginica
 setosa         14          0         0
 versicolor      0         14         0
 virginica       0          2        17
>
```

Trees and over-fitting

As with polynomials and regression, we can easily produce overly-complex decision trees which do great on the training data, but don't generalize.

This happens when the number of free (trainable) parameters in the model is similar to the number of data.

In fact, this is guaranteed to happen with decision trees, since given enough splits, it will always perfectly classify the data.

How do we deal with this? The usual approach is to prune the tree at some level, where the results are "good enough", and the model is not "too complex".

Random forests

You may have heard of "random forests". What are those?

- Random forests fall under the category of "ensemble methods". This means an averaging over several machine-learning models.
- In this case, a random forest is an average over a collection of decision trees.
- To do this,
  - bootstrapping is applied to the data set in question, and decision trees are fit to each sample;
  - however, during the training of the trees, at each split only a subset of possible features are chosen as split candidates;
  - predictions on out-of-sample data are then generated, and an average over all trees is made.
- This results in a lowering of the overfitting, which is inherently large with decision trees.

If you end up using decision trees in your research, random forests are worth considering.

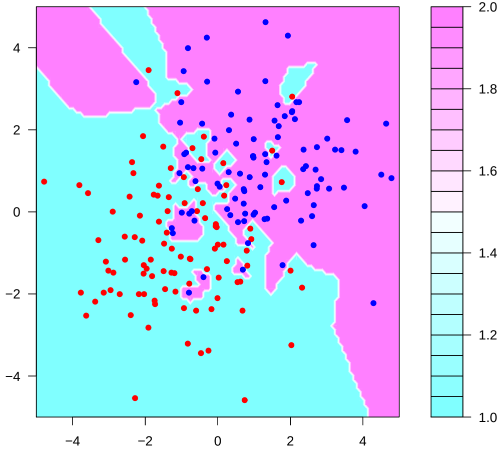
Nearest neighbours - kNN

Consider a more-geometric approach to classification: given an input data point, find the nearest point in the training set, and choose that classification for your input data point.

This is a type of regression.

A generalization is to choose the k Nearest Neighbours (kNN), and choose the classification that the majority of those k points has.

This case: two 2D Gaussians, centred on (-1,-1) (red), and (1,1) (blue) with $\sigma = 1.5$, $k = 1$.



Nearest neighbours - kNN, continued

```
library(class); n <- 100

# Generate Gaussian data.
x1 <- rnorm(n, -1, 1.5);  y1 <- rnorm(n, -1, 1.5)
x2 <- rnorm(n, 1, 1.5);  y2 <- rnorm(n, 1, 1.5)

# Generate mesh for plotting.
x.range <- seq(-5, 5, length = n)
x3 <- rep(x.range, n)
y3 <- rep(x.range, each = n)

# Put the data in data frames.
data <- data.frame(x = c(x1, x2), y = c(y1, y2))
grid.data <- data.frame(x = x3, y = y3)

# Create labels for the data.
labels <- rep(c(1, 2), each = n)
```

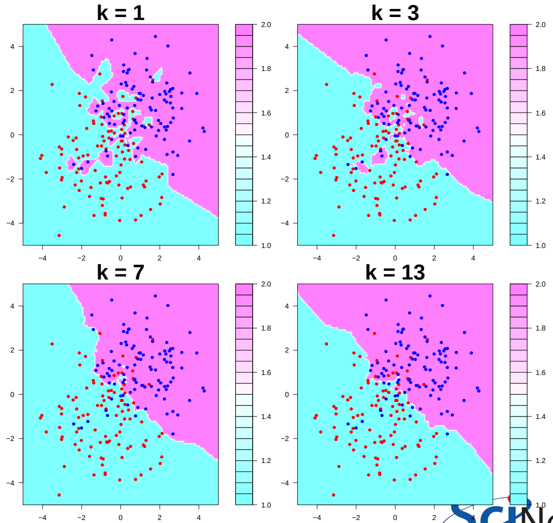
```
# Make predictions, based on
# the training data.
p <- as.integer(knn(train = data,
  test = grid.data, cl = labels))

# Make the result 2D.
dim(p) <- c(n, n)

# Make the plot.
filled.contour(x.range, x.range, p
  plot.axes = {axis(1); axis(2);
    points(x1, y1, pch = 16,
    col = "red", xlim = c(-5, 5),
    ylim = c(-5, 5), ann = F);
    points(x2, y2, pch = 16,
    col = "blue", xlim = c(-5, 5),
    ylim = c(-5, 5), ann = F)}
  )
```

Bias-variance in kNN

There's a bias-variance-like trade-off in kNN, as can be seen by varying k on the same data.

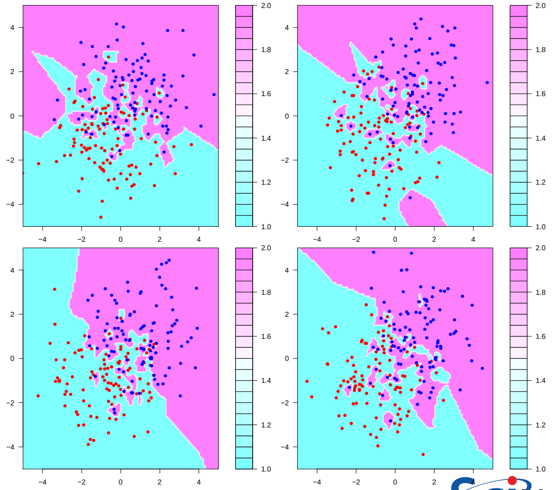
At low k, the variance is very large. The model is trying to fit to every single point.

At higher k, we average over a large area, and we start to lose features.



Bias-variance in kNN, continued

On the right we see 4 instances of the previous data set. The model has been built with $k = 1$ for all 4. It's clear that the decision boundary varies widely from one run to the next.



Scaling continuous features

In the iris data set, petal length varies over a much greater range than sepal width. If we just use Euclidean distance for $\boldsymbol{k}$NN, sepal width will provide very little information: all points are close to each other in that dimension.

We want the information in all variables to contribute to the solution. To this end, we should scale the variables to that they all get to play. A common technique is to centre the variables by subtracting off their means, and then scaling them by their standard deviations.

$$\boldsymbol{x' = \frac{x - \mu}{\sigma_x}}$$

Many libraries will do this for you, for methods where it matters. But not all will; check the documentation!

The caret package

The caret package has many many Machine Learning algorithms built into it, including kNN. It can be used to determine the most-important features of the data.

It will also select the best value for k.

The 'tuneLength' is the number of values of k caret will consider.

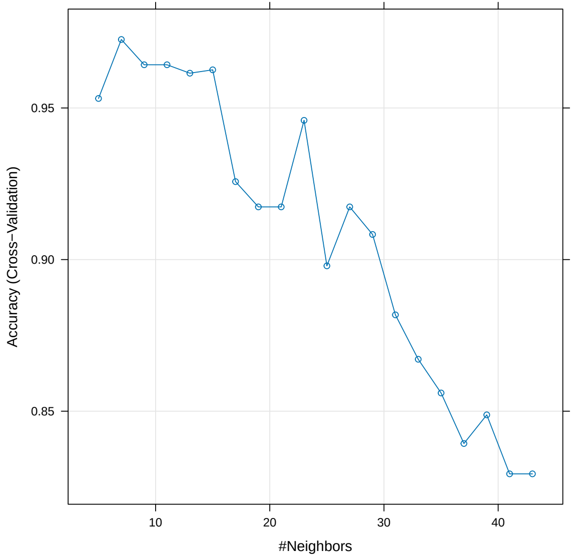
```
> library(caret)
>
> ind <- sample(c(T,F), nrow(iris),
+   replace = T, prob = c(0.7, 0.3))
>
> train.d <- iris[ind,]
> test.d <- iris[!ind,]
>
> fit.control <- trainControl(method = 'cv',
+                               number = 10)
>
> knnFit <- train(Species ~ .,
+   data = train.d, method = 'knn',
+   preProcess = c('center', 'scale'),
+   trControl = fit.control,
+   tuneLength = 20)
>
```

The caret package, continued

```
> knnFit
⋮

Pre-processing:  centered (4), scaled (4)
Resampling:  Cross-Validated (10 fold)
Summary of sample sizes:  104, 104, 104, 104, 104, 104, ...
Resampling results across tuning parameters:

k Accuracy Kappa
5 0.9531313 0.9292578
7 0.9725758 0.9584244
⋮
41 0.8293434 0.7437357
43 0.8293434 0.7437357
The final value used for the model was k = 7.
```

```
> plot(knnFit)
```

The caret analysis



The caret package, continued

```
> knnPredict <- predict(knnFit, newdata = test.d)
>
> table(knnPredict, test.d$Species)
            setosa versicolor virginica
 setosa         10          0         0
 versicolor      0         16         1
 virginica       0          2        17
>
> varImp(knnFit)
ROC curve variable importance

variables are sorted by maximum importance across the classes
              setosa versicolor virginica
 Petal.Length 100.00     100.00    100.00
 Petal.Width  100.00     100.00    100.00
 Sepal.Length  97.21      93.79     97.21
 Sepal.Width    0.00      51.31     51.31
>
```

Summary

Things to remember from today:

- Decision tree strength: can sensibly deal with categorical data.
- Decision tree strength: preform implicit feature selection.
- Decision tree strength: easy to understand (and explain) the results.
- Decision tree weakness: prone to over-fitting.
- kNN strength: completely non-parametric (data can take any form).
- kNN strength: works in as many dimensions as you like.
- kNN weakness: slow if there are too many data points.
- kNN weakness: doesn't handle categorical data.
- Note that there are other classification algorithms out there: logistic regression, naive Bayes, support vector machines, *etc.*