A Tour of Machine Learning Tools

Dan MacLean

May 2022

Table of contents

Motivation

Open the pod bay doors, please, HAL.

Many of us nowadays own a wide range of genuinely powerful computers. Smart Speakers, Mobile Phones, Gaming Consoles and Laptops and Desktops are extremely capable machines that can run software and algorithms that were in the realms of science fiction not very long ago. The increase in computing power has facilitated an expansion in the class of algorithms known as machine learners. These algorithms are part of the general field of Artificial Intelligence (AI). AI is roughly the ability of machines to make demonstrate intelligence, such as making decisions or learning skills or solving problems for which no explicit protocol is given. We are often introduced to AI by films and science fiction and it is generally seen to be a thing to be feared or exploited.

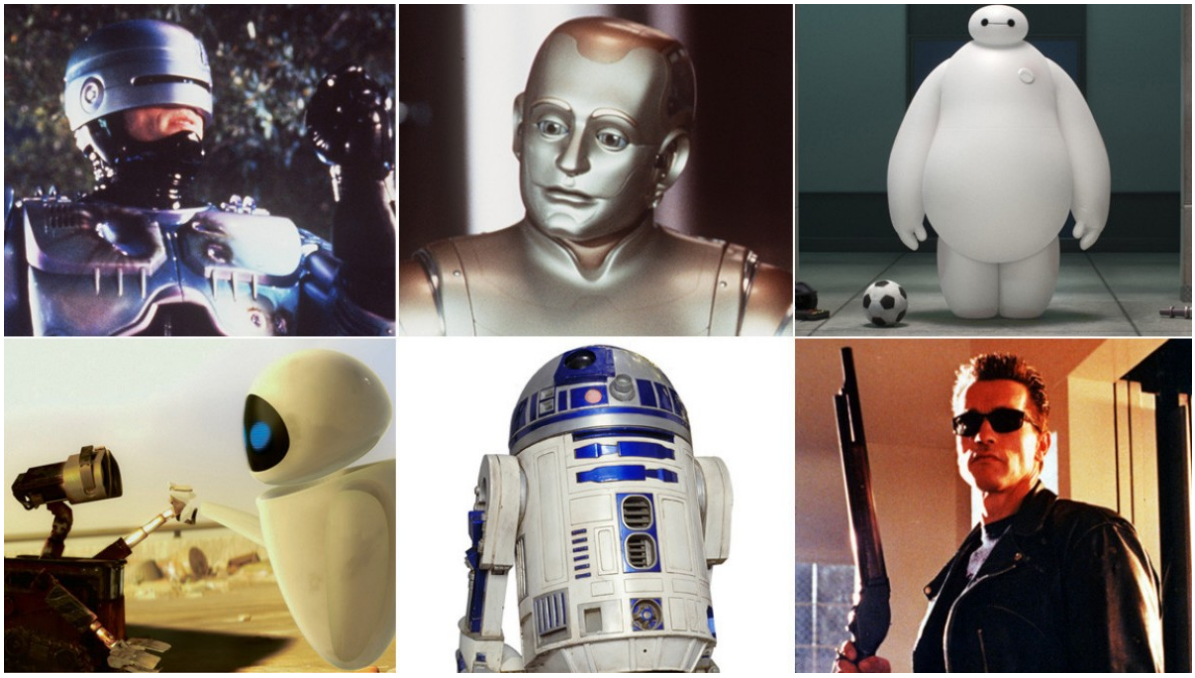


Figure 1: Some AI, yesterday.

But the aspects of AI in machine learning algorithms that we have today are extremely useful in helping us to make sense of data. Because machine learning is concerned with algorithms

that improve through experience and by learning from data, they can help us to see patterns and connections in data that we would not be able to see ourselves. Essentially by performing many calculations machine learning algorithms can help us to find structure or divisions in data, perform classifications, create simplifications of complex structures and make data based decisions. These are the algorithms that help us to make sense of the vast swathes of data that we generate in our daily lives and we can make use of them too to help us make sense of complex experimental data.

In this course we'll tour the main classes of machine learning algorithm, with a focus on ones that are common in biological data analysis. This will not be a deep dive, though we will detail the workings of some at a high level. The objective is to tour the galaxy of ML and finish with an appreciation of how they can help and how you can start to make use of such mathematically complex and powerful tools in your research.

1 Unsupervised Learning

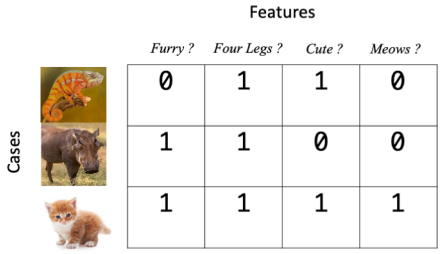
1.1 About this chapter

1. Questions
    - How can I find groups of similar things in data?
2. Objectives
    - Understand features and cases
    - Understand hierarchical and k-means clustering
3. Keypoints
    - Unsupervised learning is finding groups in a data set without known examples
    - The number of cases we have should be greater than the number of features each case has

In this chapter we'll take a look at unsupervised learning tools. This is a great place to start with ML as a biologist because, whether you know it or not, you're actually already familiar with a good number of the principles in this field. Unsupervised learning is a form of data-driven ML. In these approaches we start off with a mish-mash of things that we have information about but we don't know what any of them are. Are aim is to try and group the similar things together, and the different things apart.

1.2 p Features and n Cases

When we talk about information that we know about things, we really mean the things we noted in the experiment. This 'information' can be diverse things including but not limited to a biological sequence, a set of physical measurements, some category values, or gene expression values. There are different types of ML tool for dealing with them all. In a very general sense, what ML tools work with is an $n \times p$ matrix of n cases and p features, the features p are the things we change and the cases n are the different items or individuals we measured the features on, here's a basic example in which the presence of a feature is indicated by a 1 and it's absence by 0.



Here's a gene expression based example, gene counts from an RNAseq experiment. The treatments are p, the different genes are the ns

geneid	trt1	trt2
gene1	9.313663	5.683347
gene2	16.824577	12.921582
gene3	11.134654	18.386286
gene4	18.245261	13.271525
gene5	19.107009	11.849221

Finally, here's a sequence based example

id	seq
gene1	TESVI
gene2	NESCI
gene3	TESNI
gene4	LEDVT
gene5	ANDVI

In general ML tools need this to be true

$n >> p$

n must be much greater than p. We must have many more cases than features we measured. Most ML tools will fail, or at least have reduced power when this isn't true. This can be a limiting factor in our ability to use ML. Conversely, the power will generally go up when it is true and very large data sets can give extraordinary ML power.

1.3 Clustering

The first class of ML tools we will look it is unsupervised clustering, this will be familiar to many biologists from heatmaps of gene expression data, but also more fundamentally from phylogenetic tree analysis. We'll look at a general overview before we look at some specific tools.

1.3.1 The distance measure and matrix

The first step of clustering is to get a measure of 'distance' between all the pairs of cases *n*s that we have gathered. In this case when we say 'distance', we mean a numeric measure of how similar or dissimilar our cases are. There are lots of different metrics of distance, e.g the correlation coefficient r is a measure of how similar two sets of numbers are. With this measure the higher the value, the more similar the cases are. Different types of data will need different distance metrics. For sequence based data we typically have the substitution or edit distance (the number of changes needed to make the two sequences identical).

The distance measure is a crucial step in clustering, but all tools have a sensible default and we don't need to worry about what it is *exactly* at this stage beyond what we've discussed, but we do need to see what we do with the pairs of distances to understand the basics of the algorithm. Once we have the distances we form a distance matrix, which is always square, symmetrical across the diagonal and looks like this:

```
      gene1 gene2 gene3 gene4 gene5
gene1  0.00  1.00  0.87  1.00  0.96
gene2  1.00  0.00  0.77  1.00  0.99
gene3  0.87  0.77  0.00  0.97  1.00
gene4  1.00  1.00  0.97  0.00  0.99
gene5  0.96  0.99  1.00  0.99  0.00
```

As we can see, the further apart the genes expression across the treatments the greater the distance measure. Once we have this matrix the clustering can begin.

1.4 Hierarchical (single linkage) Clustering

Hierarchical clustering is the most common and straightforward clustering algorithm. The elements (cases or *n*s) are formed into the distance matrix and the aim is to group the pair of elements with the smallest distance into one, then repeat, continuing until we run out of elements. We then move onto grouping the pairs and so on until there's nothing left to group.

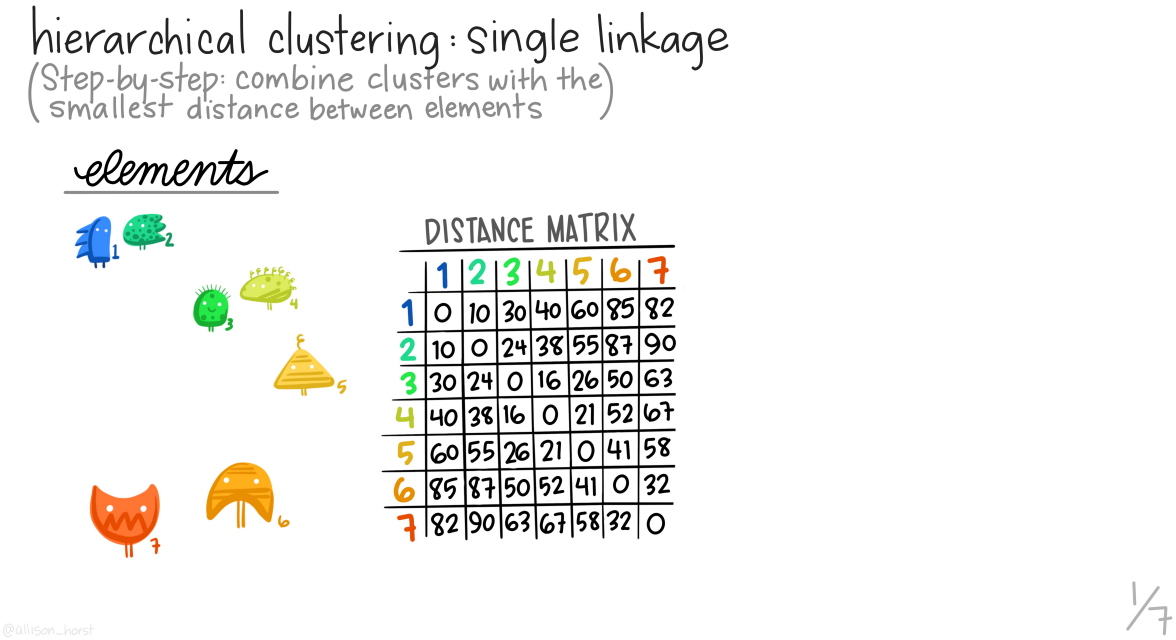


Figure 1.1: Artwork by @allison_horst

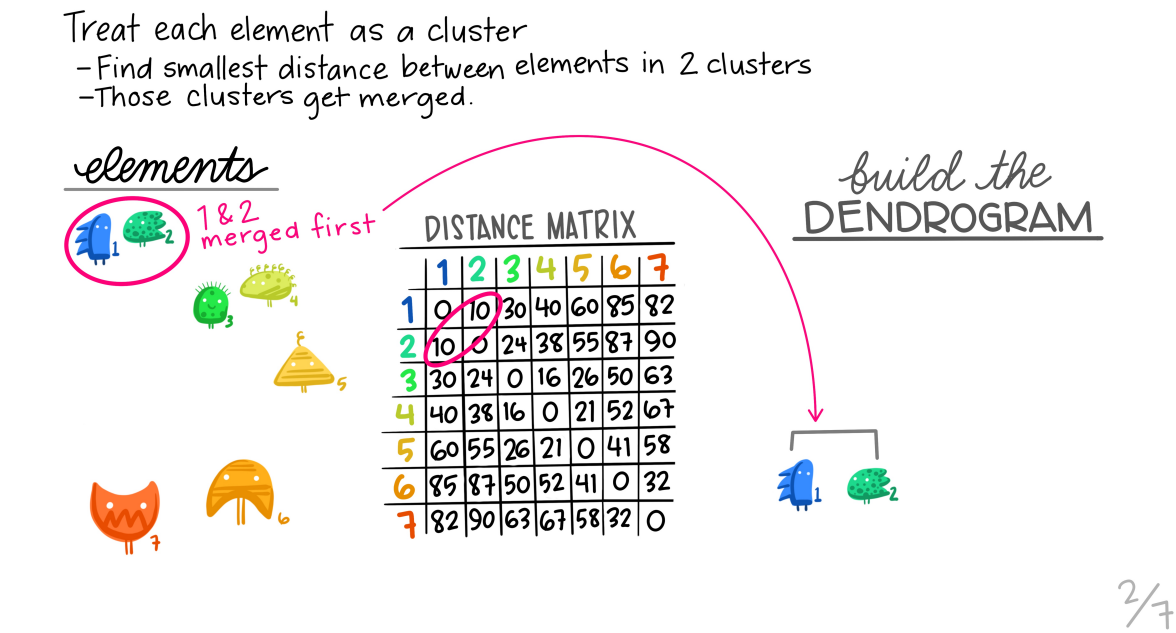


Figure 1.2: Artwork by @allison_horst

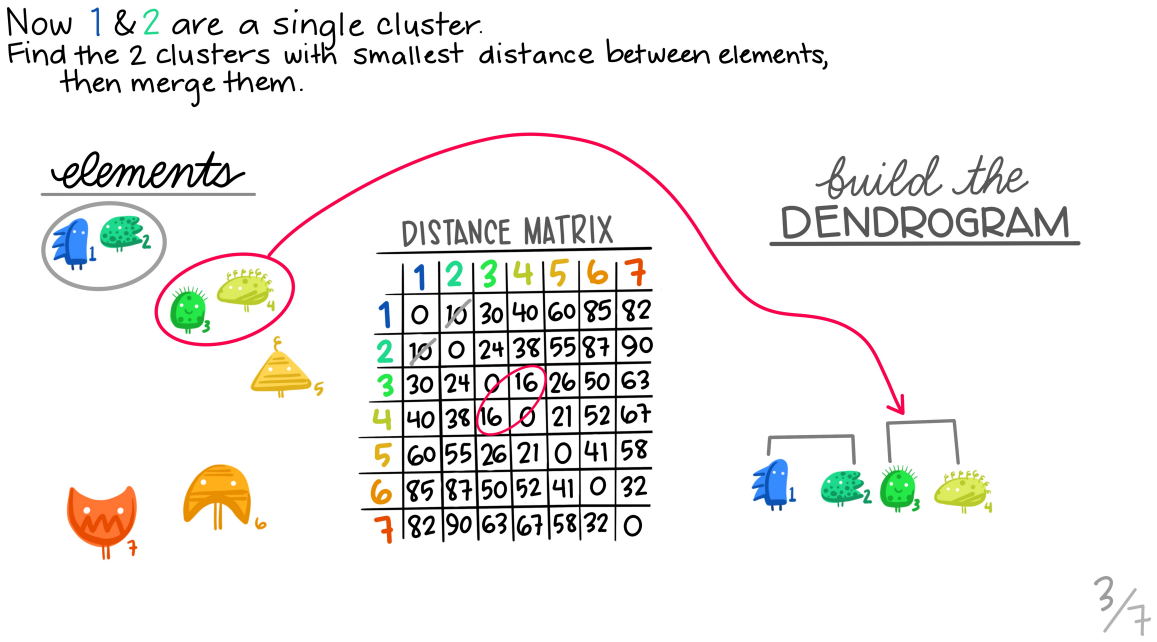


Figure 1.3: Artwork by @allison_horst

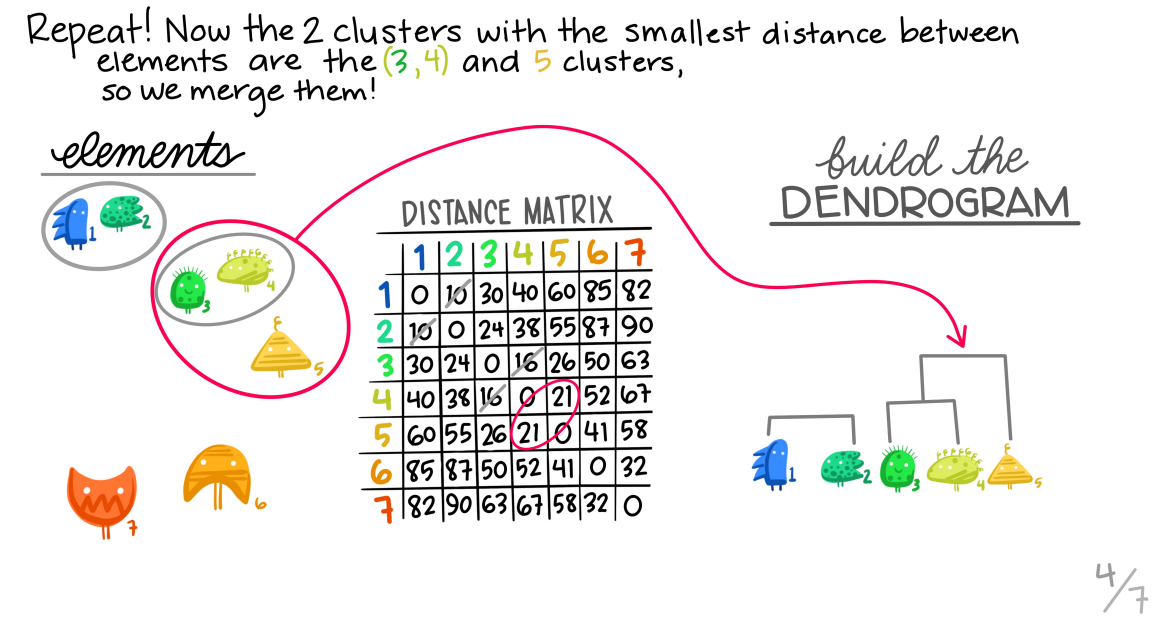


Figure 1.4: Artwork by @allison_horst

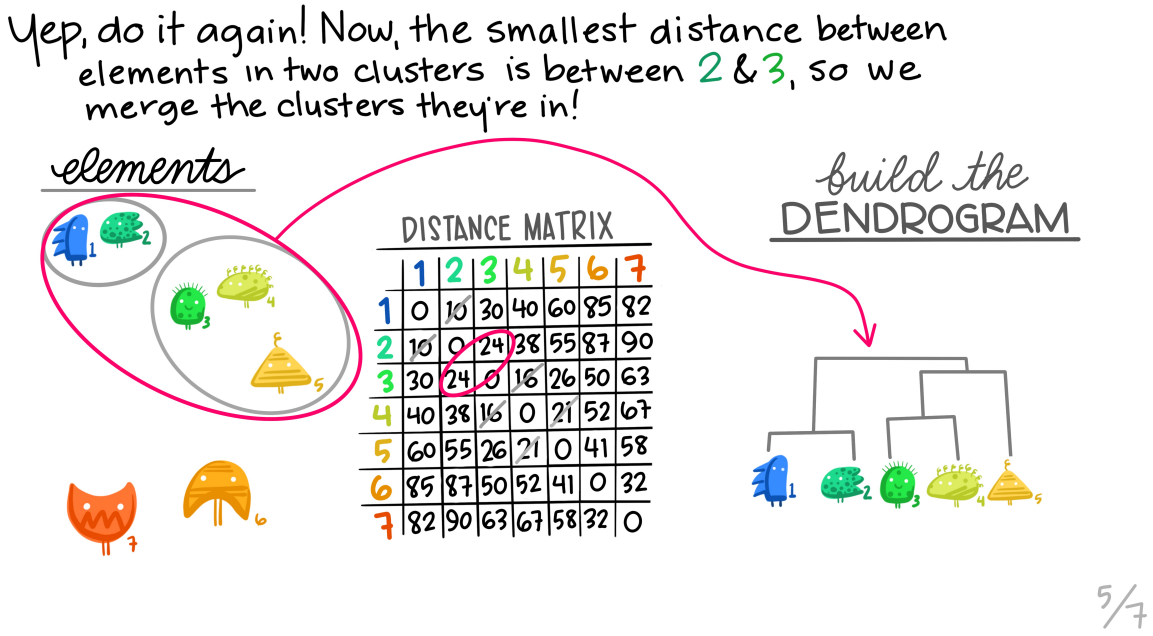


Figure 1.5: Artwork by @allison_horst

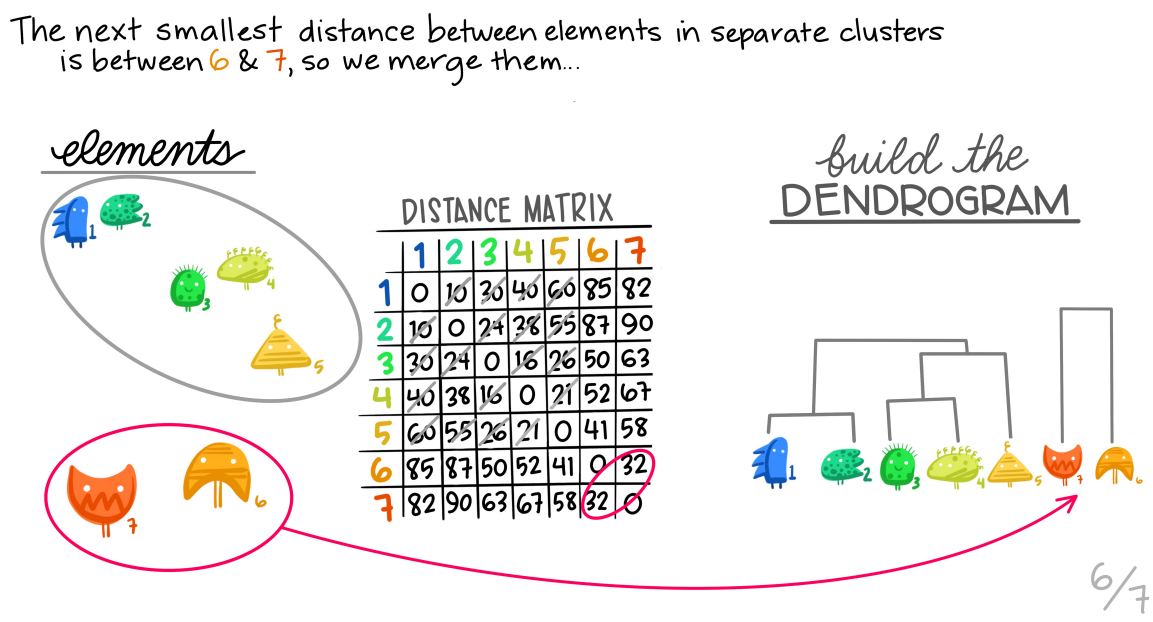


Figure 1.6: Artwork by @allison_horst

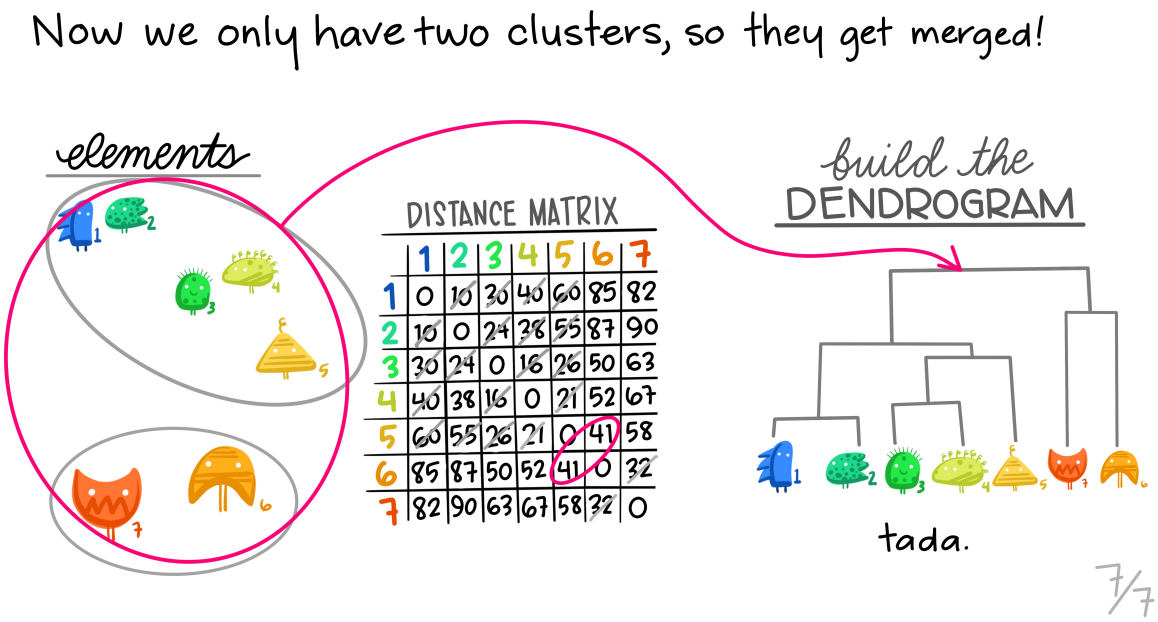


Figure 1.7: Artwork by @allison_horst

Allison Horst demonstrates it better than I, so here's the excellent illustrations she made that run through the process.

Hopefully this montage has clarified the overall process of grouping elements based on distance metrics calculated between all pairs. The question remains though, how can we do this in R?

1.4.1 Hierarchical clustering in Base R

There is an `hclust()` function built into R we can use. Being part of the base distribution and not using any packages means that this function is a bit general and needs data in particular format. Specifically it needs a numbers only matrix or data frame of information - you'd need to remove all text information from the object - getting this in to shape is left as an exercise for the reader. You would end up with a matrix object looking something like this

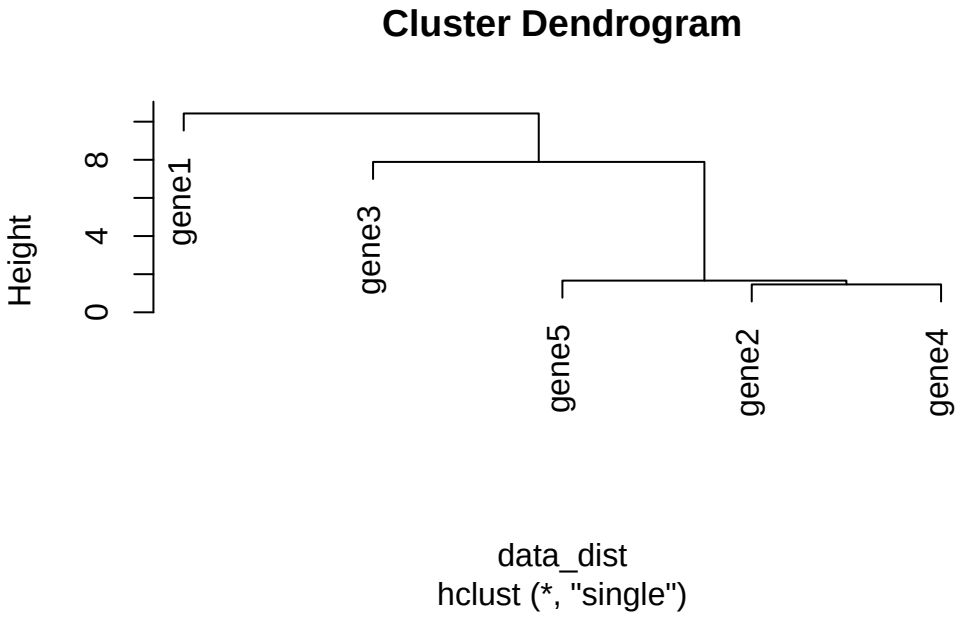
```
data_mat
```

```
           trt1      trt2
gene1  9.313663  5.683347
gene2 16.824577 12.921582
gene3 11.134654 18.386286
```

```
gene4 18.245261 13.271525
gene5 19.107009 11.849221
```

You can the get clusters by creating the `dist` object with the `dist` function and the clusters with `hclust()` using the method `single` to apply the single linkage clustering we learned above. Then we can directly plot the dendrogram.

```
data_dist <- dist(data_mat, diag=TRUE)
clusters <- hclust(data_dist, method="single")
plot(clusters)
```



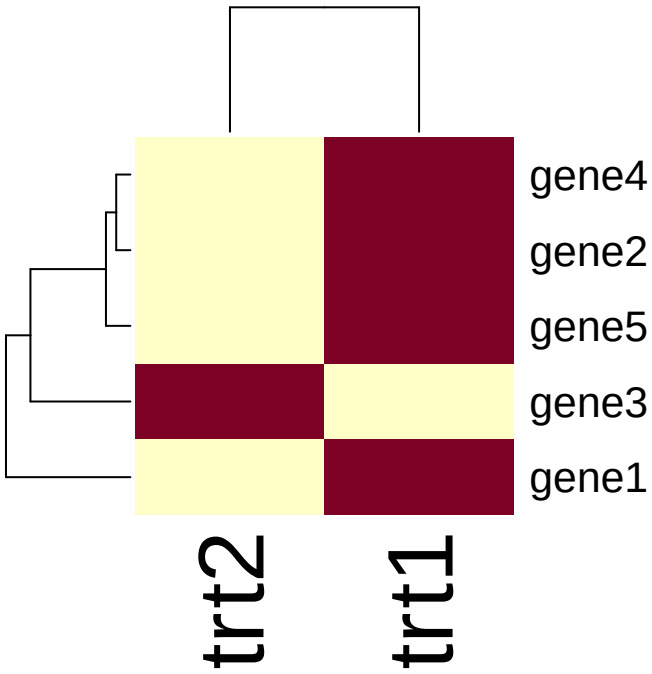
Note that the distance measure by default is `euclidean` which is a different way of computing distances than the r correlation coefficient we discussed earlier. `euclidean` is more commonly used, but it's beyond the scope of this course to discuss distance measures in detail. More information on distance measures is freely available on Wikipedia.

1.4.2 Clustered Heatmaps

Typically you'll want to make some sort of heatmap and have a tree or dendrogram of the clusters stuck on the side, rather than just have a cluster tree on its own. Again, base R has a

helpful if general function, `heatmap()`, simply pass the matrix object e.g `data_mat` and it can do the rest.

```
heatmap(data_mat)
```



The function has a lot of customisation options, which you can investigate using `?heatmap` and Google!

1.4.3 Extra Credit: ggplot and clusters

Heatmaps can be drawn in `ggplot` using the `geom_tile()` geom. If our data are in 'tidy' format like this

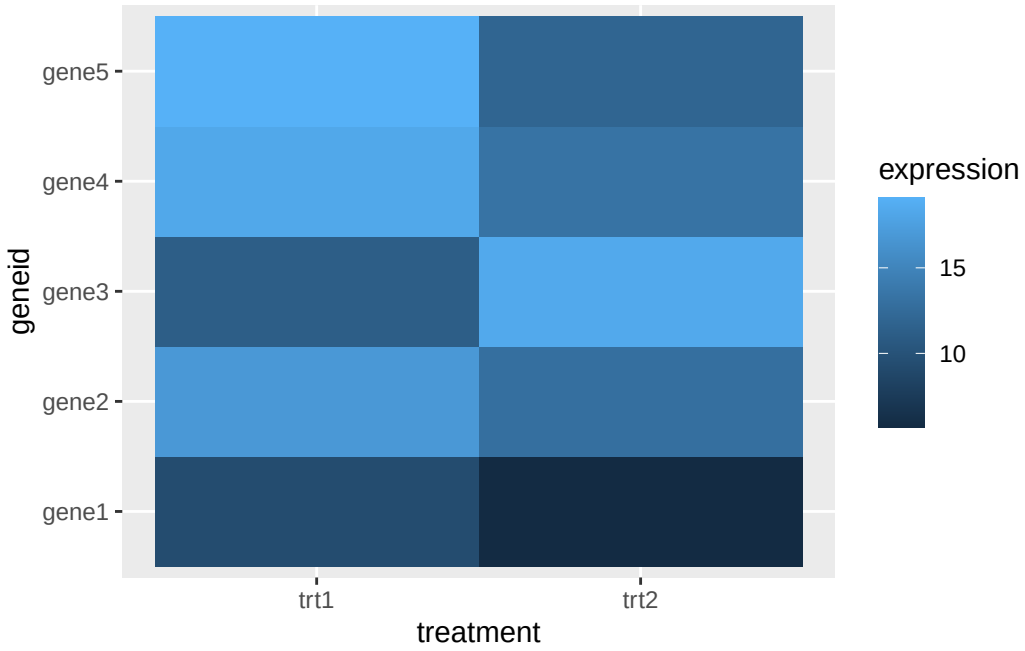
```
tidy_gdf
```

```
# A tibble: 10 x 3
   geneid treatment expression
   <chr>  <chr>          <dbl>
 1 gene1  trt1            9.31
 2 gene1  trt2            5.68
 3 gene2  trt1           16.8
```

```
 4 gene2  trt2           12.9
 5 gene3  trt1           11.1
 6 gene3  trt2           18.4
 7 gene4  trt1           18.2
 8 gene4  trt2           13.3
 9 gene5  trt1           19.1
10 gene5  trt2           11.8
```

We can make a heatmap quite simply, like this

```
library(ggplot2)
hmap <- ggplot(tidy_gdf) + aes(treatment, geneid) + geom_tile(aes(fill=expression))
hmap
```



But this has no dendrogram and is not clustered! It's going to take a little fiddling to add this on - we can get the clusters out of the `hclust()` result and apply those. To do that we must solve another problem first - turning our tidy data into a matrix! That can be done with `pivot_wider()` from `tidyr`, which gets us most of the way there.

```
library(tidyr)
wide_gdf <- tidy_gdf %>% pivot_wider(
```

```
    id_cols = "geneid",
    names_from="treatment",
    values_from="expression")
  wide_gdf
```

```
# A tibble: 5 x 3
  geneid  trt1  trt2
  <chr>  <dbl> <dbl>
1 gene1   9.31  5.68
2 gene2  16.8  12.9
3 gene3  11.1  18.4
4 gene4  18.2  13.3
5 gene5  19.1  11.8
```

And we can now remove the non-numeric columns and do the cluster

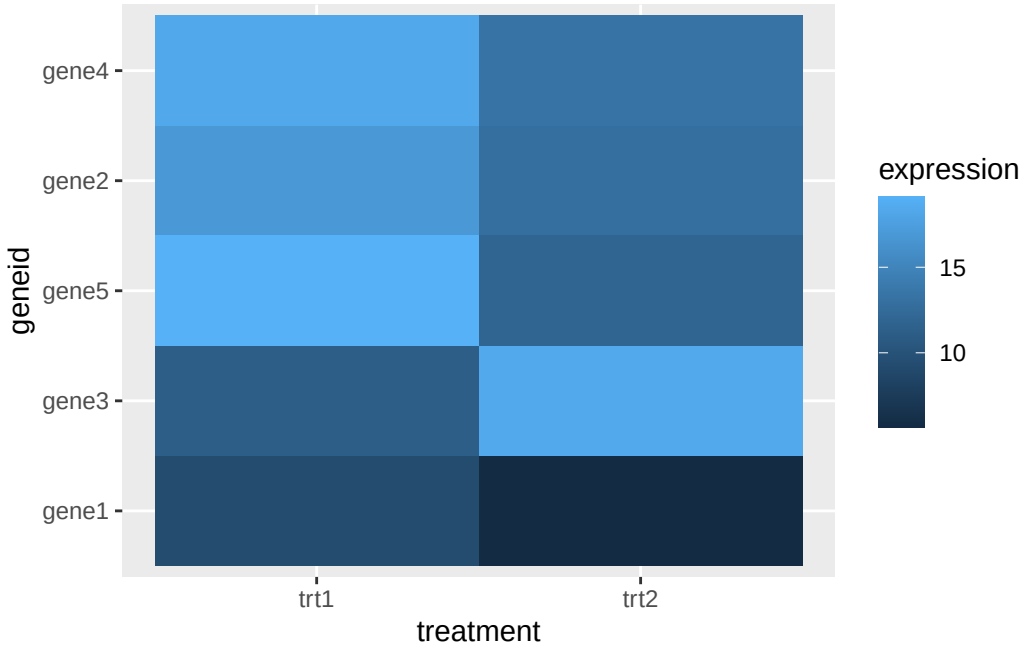
```
data_mat <- wide_gdf %>%
  select(-geneid) %>%
  as.matrix()

clusters <- hclust(dist(data_mat, diag=TRUE))
```

We can reorder the axis in our `ggplot` heatmap using the `order` from the `clusters` object to put the `geneid`s into the right order

```
clusters$order
```

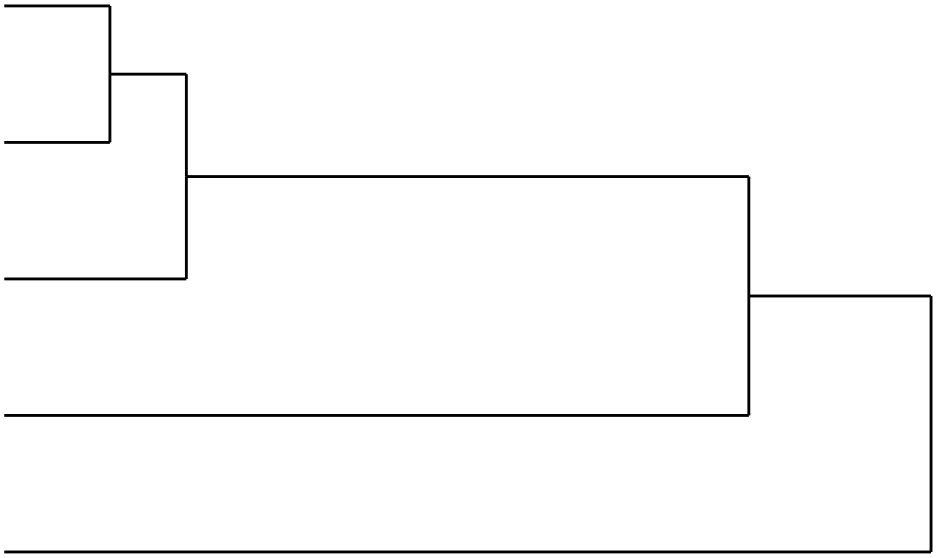
```
[1] 1 3 5 2 4
```

```
hmap + scale_y_discrete(limits= wide_gdf$geneid[clusters$order] )
```



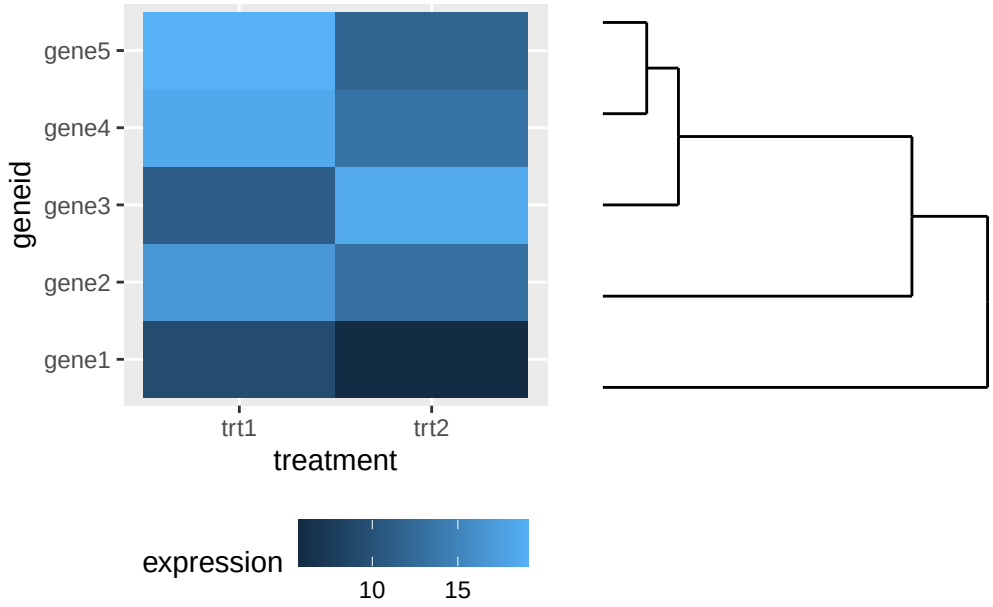
The ggdendro package allows us to create a dendrogram from a clustering

```
library(ggdendro)
dendro <- ggdendrogram(clusters) + coord_flip() + theme_dendro()
dendro
```



We can compose the two plots with `patchwork` (cheekily moving the legend out of the way first).

```
library(patchwork)
hmap + theme(legend.position="bottom") +  dendro
```



1.5 K-Means clustering

A limitation of hierarchical clustering is that we as the operator have to guess what elements are in which cluster and that can be a bit arbitrary. An alternative algorithm, the K-means cluster gets around this problem by allowing us to specify the number of clusters up-front and works from there. It starts with the assumption that there are `k` clusters and makes `k` random cluster start points (centroids) then tries to assign cases (elements/observations) to one of each centroid based on the distance from the start points. The assignment to clusters is improved iteratively by starting again with the centroid at the mean point in each cluster and continues until no improvements are made. Again, Allison Horst has drawn some great guides

1.5.1 Figure of Merit

A limitation of this approach and of hierarchical clustering is that we may not know how many `k` clusters there are. The Figure of Merit (FOM) technique can help us work out the `k` that we need. Briefly, this works by trying a k-means clustering at 1, then 2, then 3 up to a stopping number of clusters and at the end of each clustering we check the distance variability between the centroids and the elements/cases/observations. The value of `k` that minimises the distance is the value that we want to use as most points are near to a cluster centroid.

Let's walk through the process of doing FOM and then applying a k-means clustering.

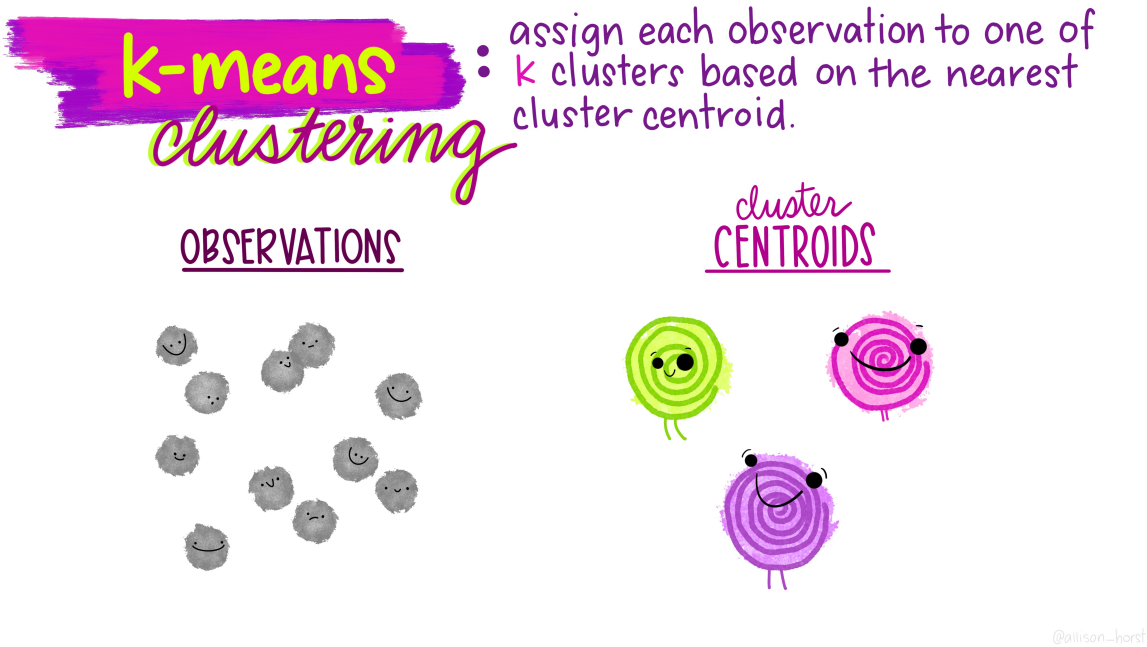


Figure 1.8: Artwork by @allison_horst

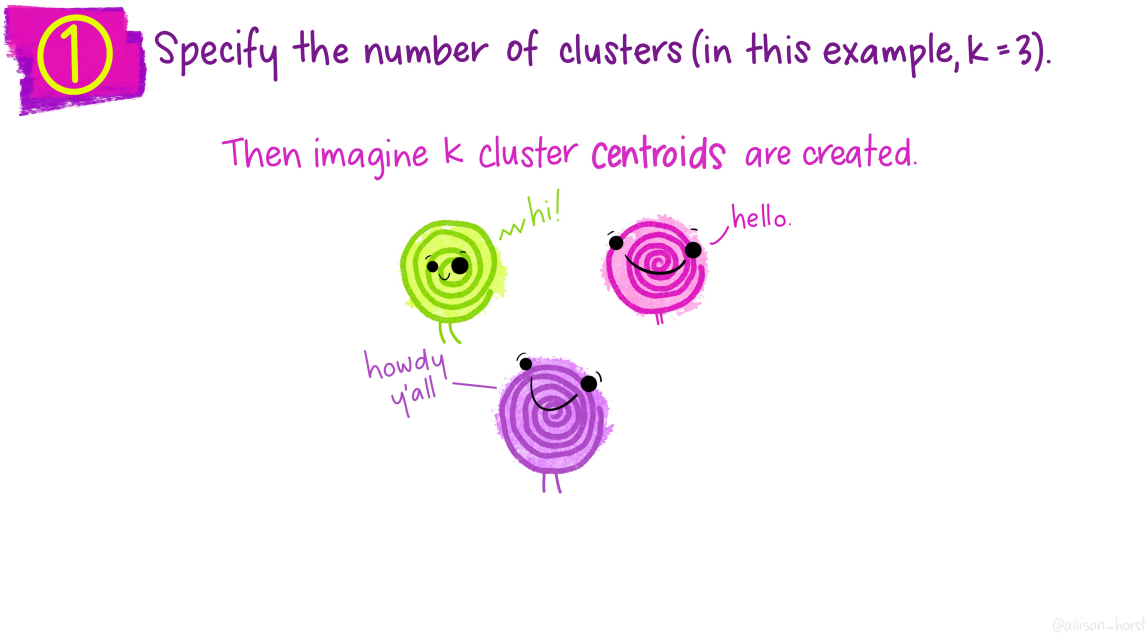


Figure 1.9: Artwork by @allison_horst

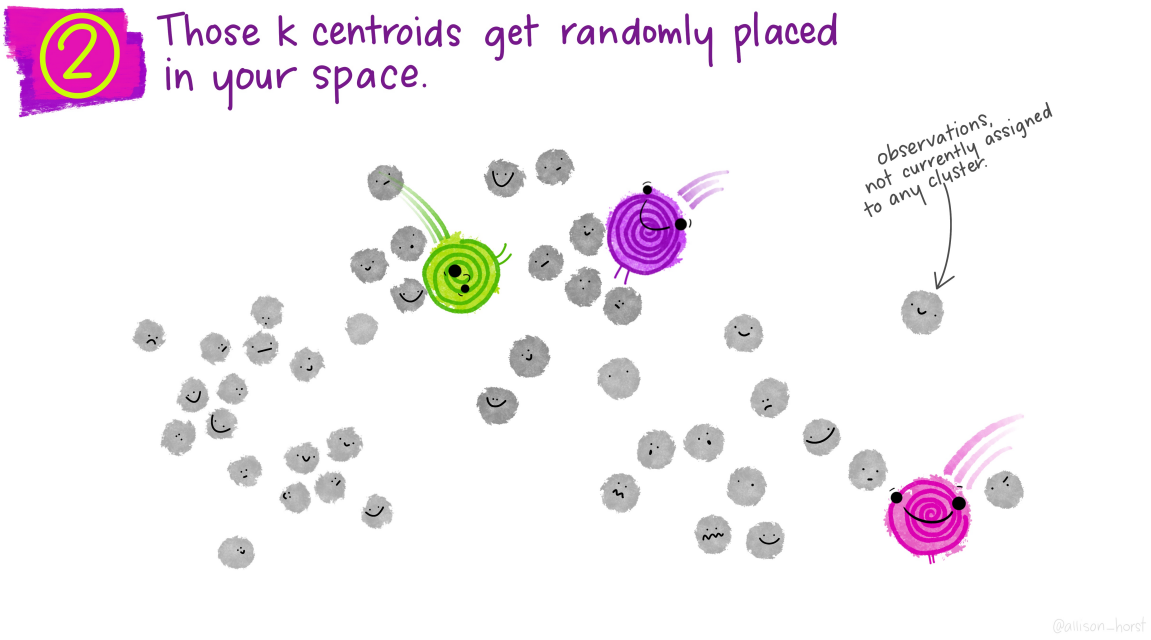


Figure 1.10: Artwork by @allison_horst

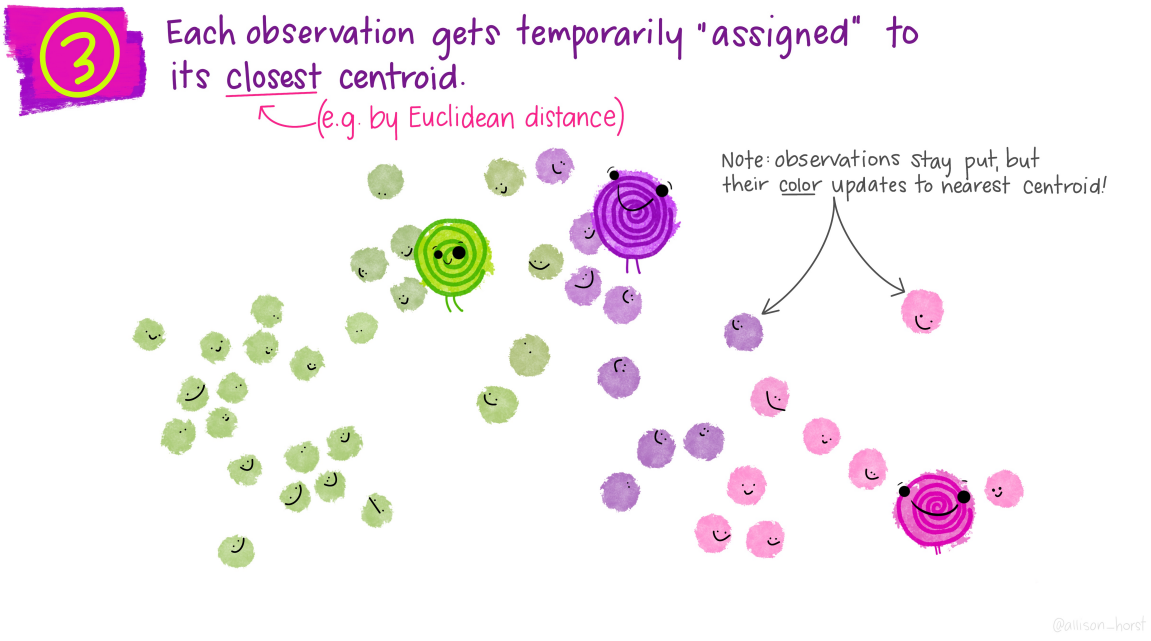


Figure 1.11: Artwork by @allison_horst

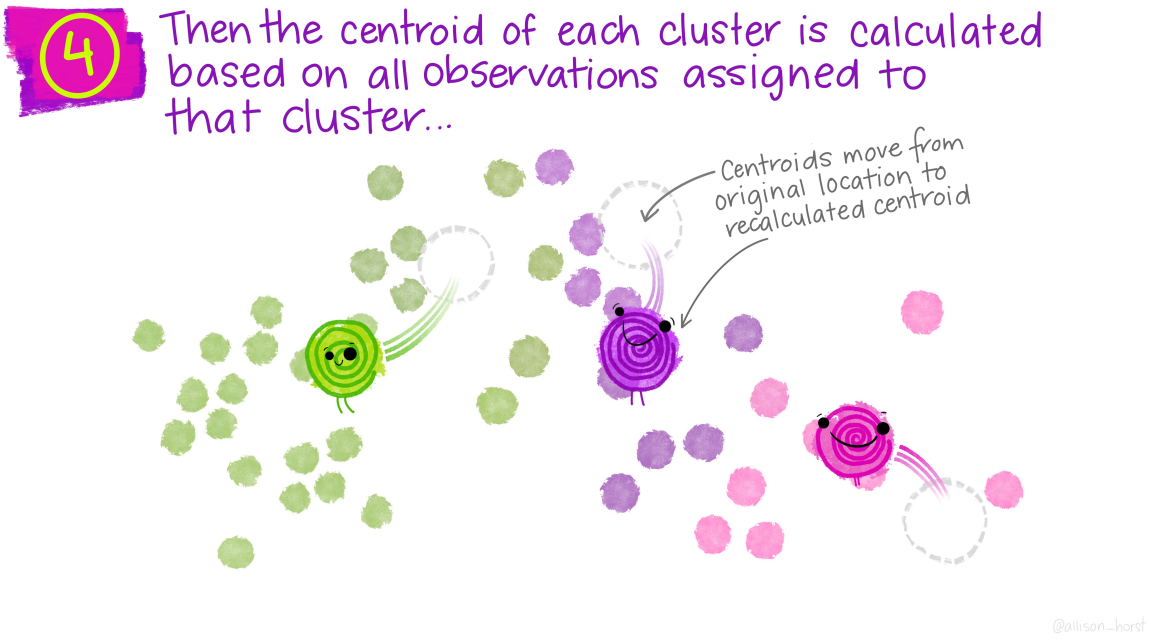


Figure 1.12: Artwork by @allison_horst

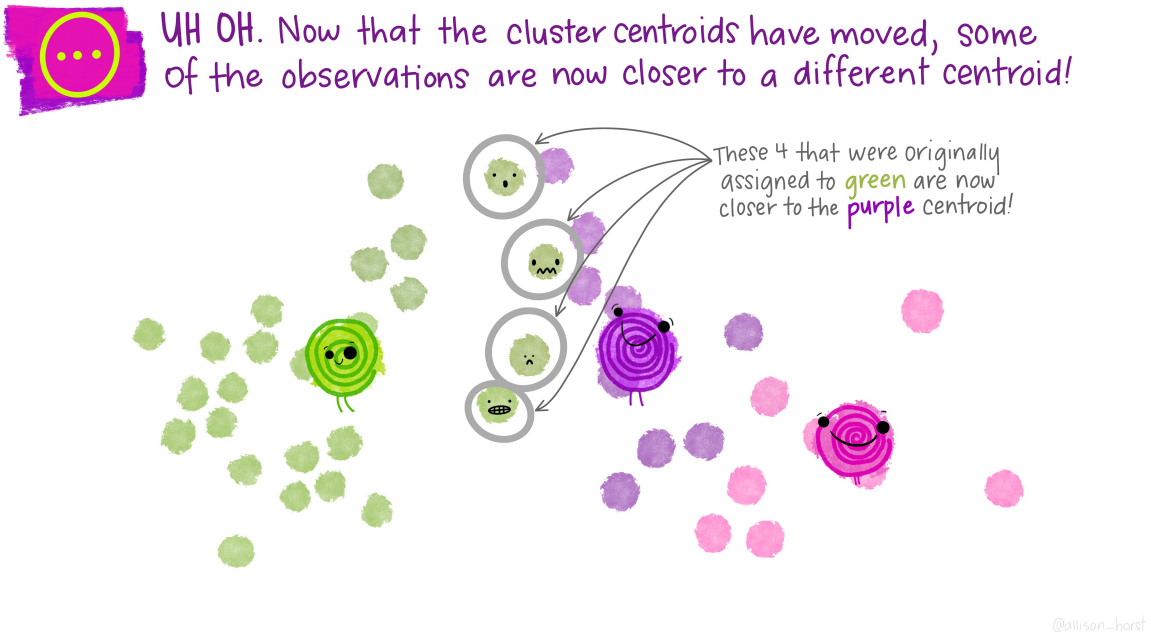


Figure 1.13: Artwork by @allison_horst

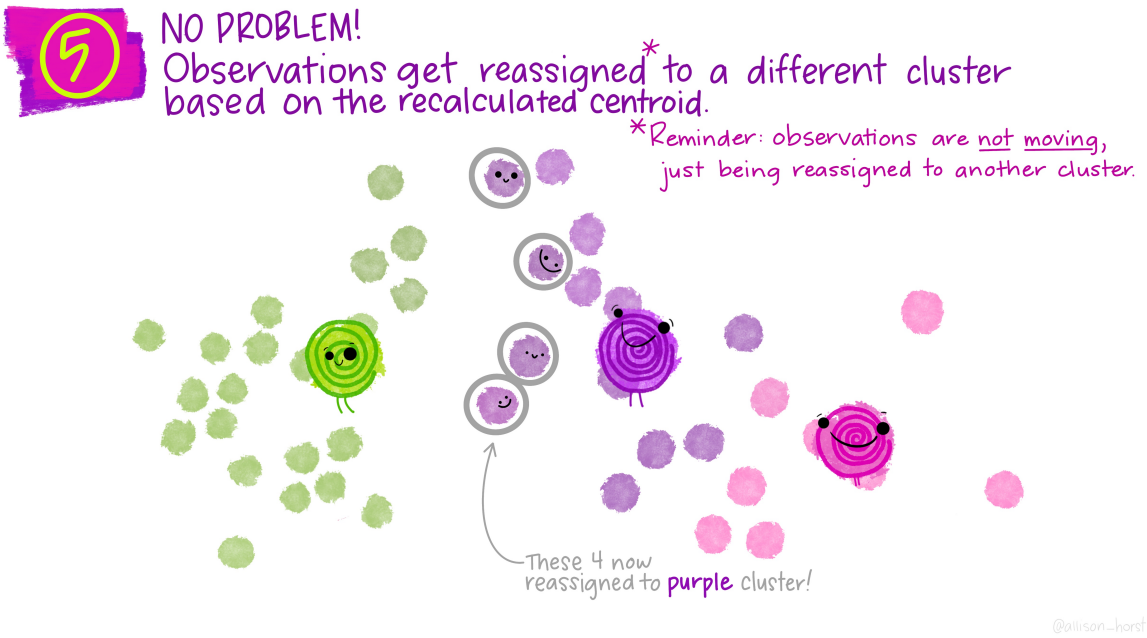


Figure 1.14: Artwork by @allison_horst

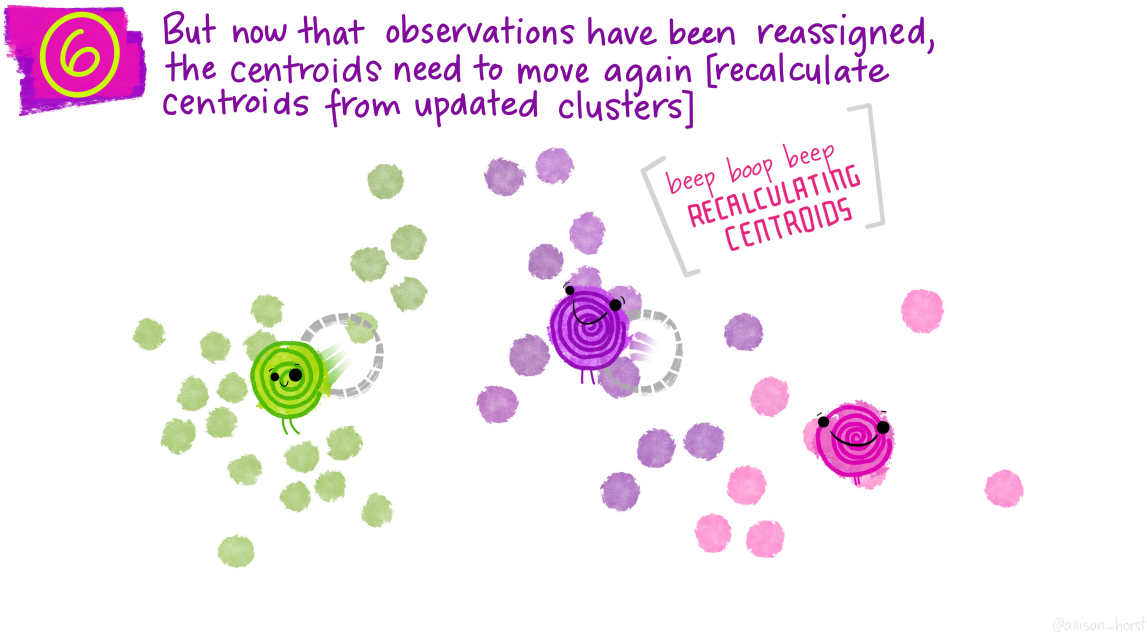


Figure 1.15: Artwork by @allison_horst

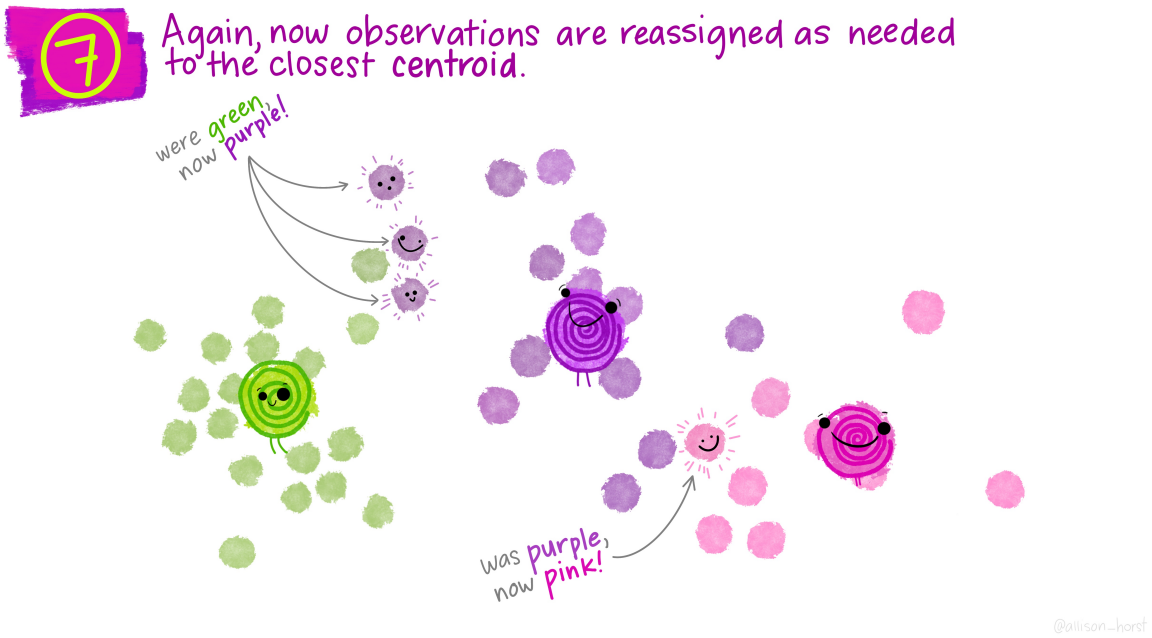


Figure 1.16: Artwork by @allison_horst

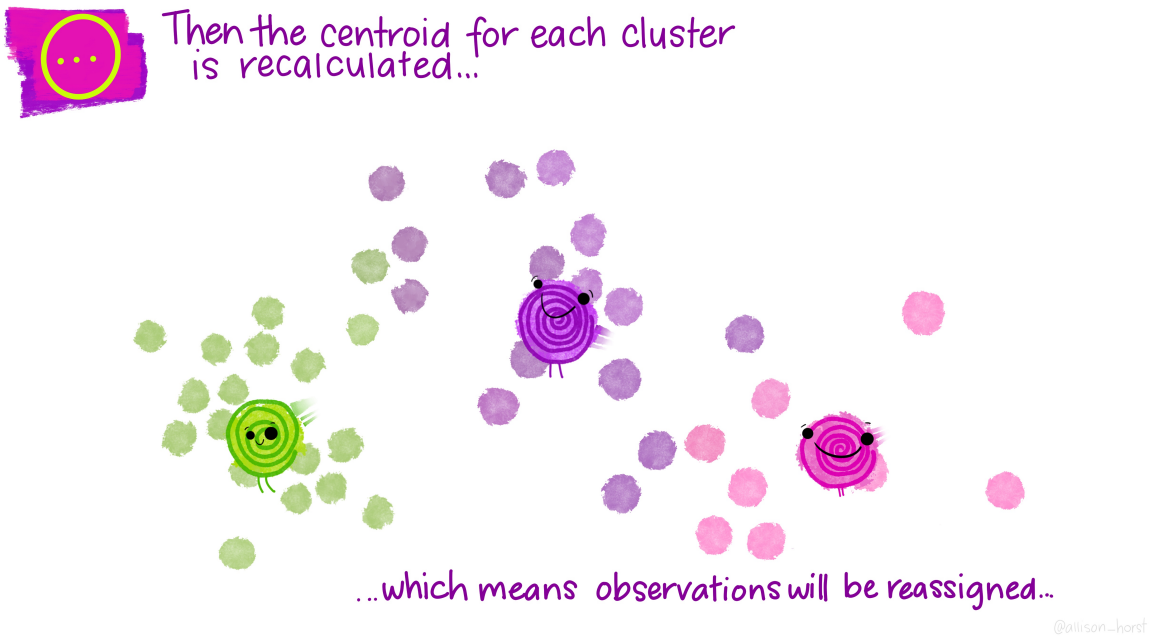


Figure 1.17: Artwork by @allison_horst

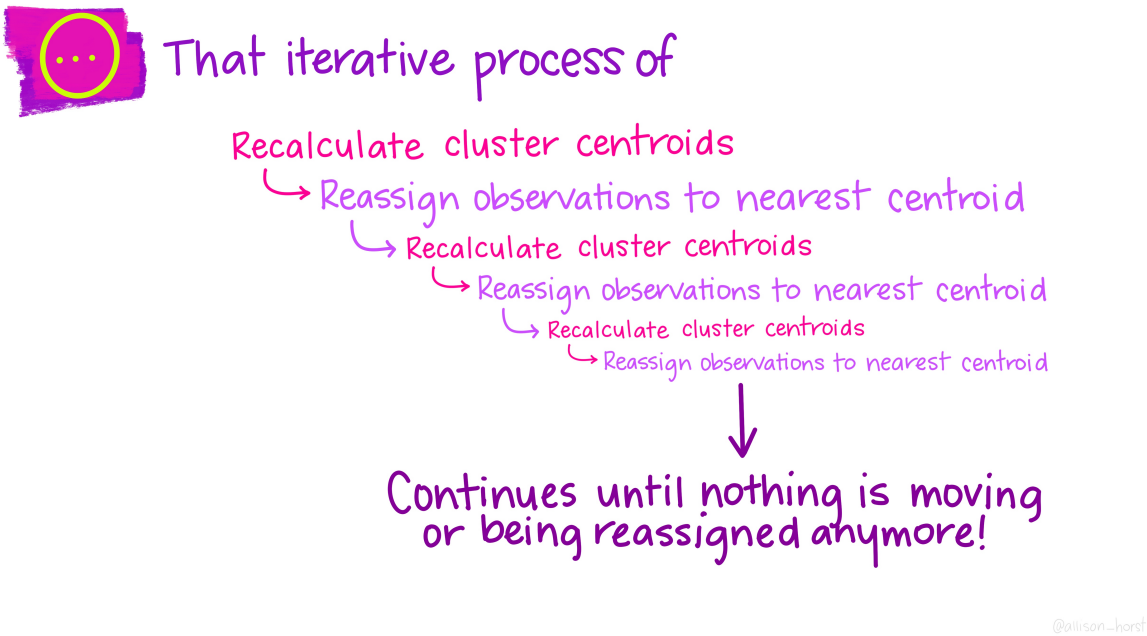


Figure 1.18: Artwork by @allison_horst

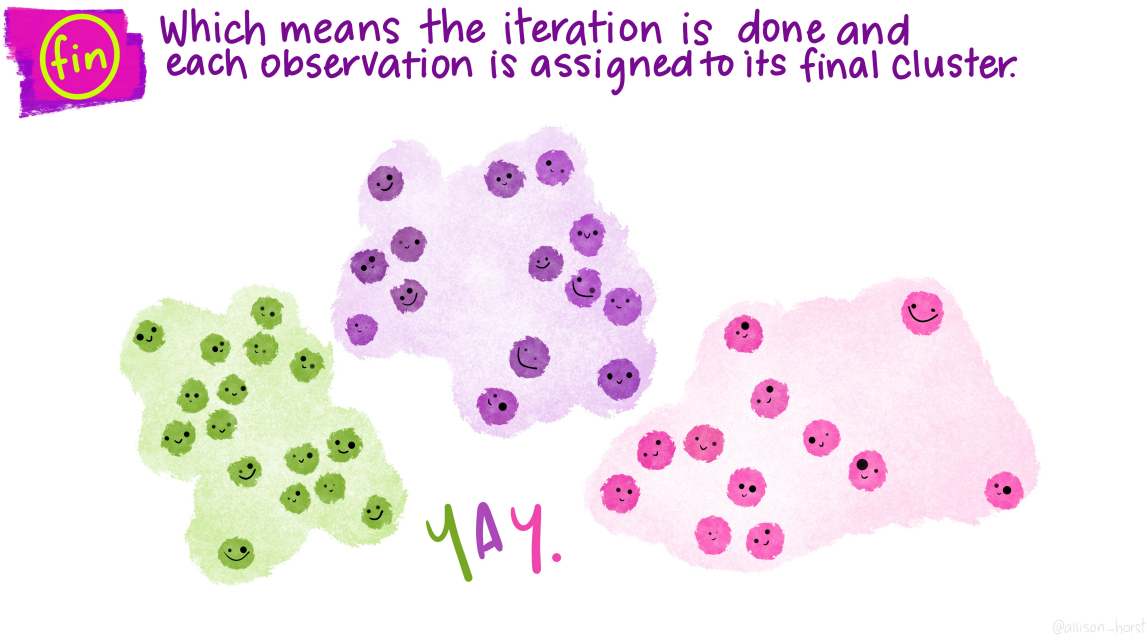


Figure 1.19: Artwork by @allison_horst

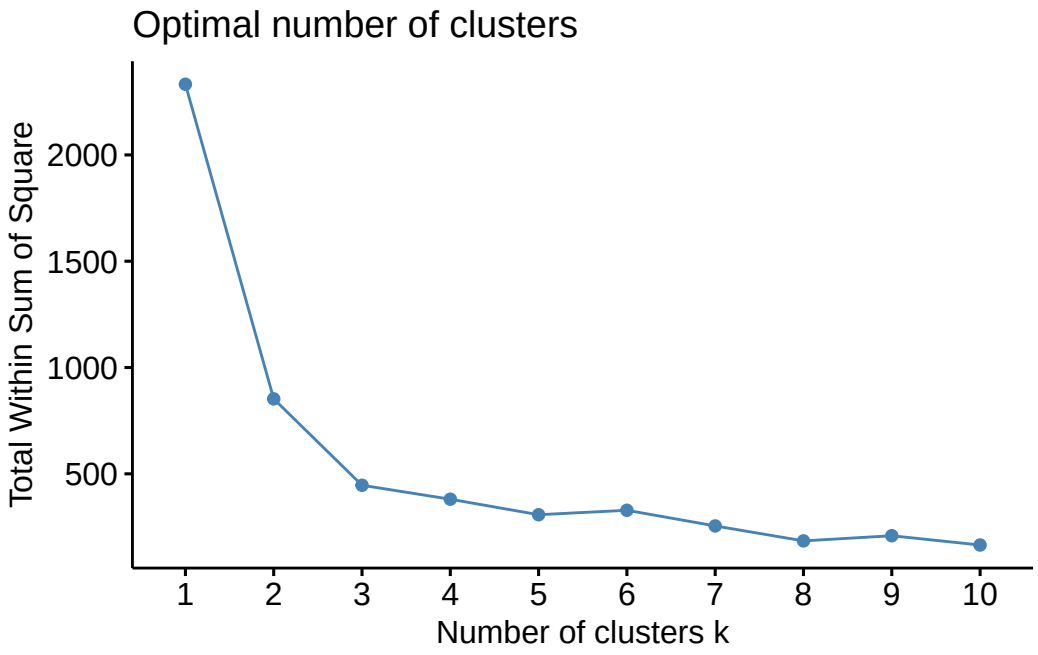
Here's a sample data set to try and cluster, we're going to cluster the rows. Note how it resembles a gene expression matrix with the gene names as the matrix row names, not in the data itself.

```
head(gene_exprs, n=3)
```

```
     sample1  sample2  sample3
gn1 4.516471 3.722275 15.52375
gn2 1.451247 3.225752 15.21140
gn3 3.584614 3.020616 15.40538
```

We'll use the `factoextra` package to do the FOM and the subsequent k-means. First the FOM using `fviz_nbclust()`

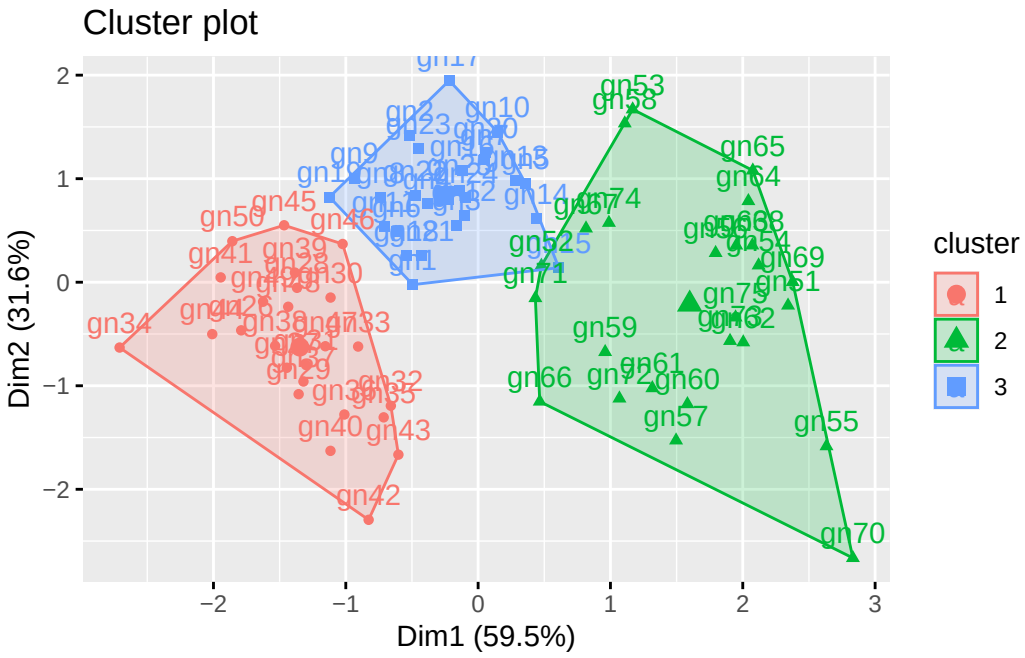
```
library(factoextra)
fviz_nbclust(gene_exprs, kmeans, method="wss")
```



The option `method` lets us specify which method we want to use to estimate the variability, here we use `wss` for within sum of squares, which is a reasonable one. The resulting plot shows that `wss` improves lots until we get to 3 clusters, at which point there is only minimal improvement. We interpret this as meaning that there are 3 clusters within our data. We can

use that to make our k-means cluster. The `kmeans()` function does this easily and we can plot the result using the `fviz_cluster()` function.

```
km_clus <- kmeans(gene_exprs, 3, nstart=25, iter.max = 1000)
fviz_cluster(km_clus, data=gene_exprs)
```



The plot shows clearly the elements of the data are clustered into 3 groups. The `km_clus` object contains information about the elements cluster membership if you wish to extract that for any reason.

> **Roundup**
>
> - Unsupervised learning algorithms group things based on distances computed between them.
> - Hierarchical and k-means are two common and useful methods.

2 Supervised Learning

2.1 About this chapter

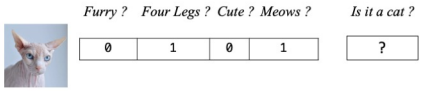
1. Questions
    - How can I find items in data that are like things I already know about?
2. Objectives
    - Understand labelled data and classification
    - Understand training and test data
    - Understand K nearest neighbours and Random Forest
3. Key Points
    - Supervised learning is classifying cases or elements based on examples that we already know
    - Good training data is key
    - Don't mix test and training data

In this chapter we'll take a look at supervised learning tools. It's called supervised learning because we have a set of data that we have already classified into one or more groups and the algorithms use that as guide and try to fit some other unknown data into the groups we've specified, so the classification is supervised in the sense that there are known examples of the groups. Again the input data is usually a data matrix of some features, like measurements or gene expression values.

2.2 Labelled Data

For supervised learning algorithms we need to give examples of our categories. This is called labelling the data. And in most cases we can achieve this just by extending our np features/cases data matrix by one column and add a label in there, usually as a number. For our animal matrix example that would look like this if we wanted to label our data as a cat or not.

The object for the learning algorithm is then to guess labels for data that we don't know beforehand. So in our animal matrix example, that looks like this



2.3 Training and Testing

2.3.1 The Training Phase

Most supervised learning algorithms have an initial training phase. Training is a part of the procedure where the algorithm creates a model - an internal representation of the data and the associated categories or groups - that it can later use to tell which of our categories a new observation or case belong to. Each type of supervised learner has a different approach to training.

2.3.2 The Testing Phase

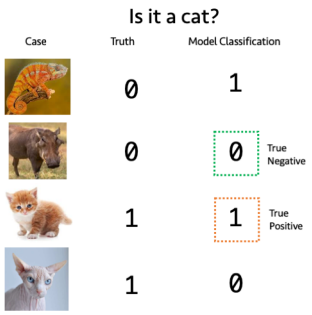
Once we have a trained model we must evaluate its accuracy. If we can't tell how accurate the model is, then we can't trust it's predictions and there is not point proceeding. We can test the model on data that *we* know the labels of but that the model *hasn't seen before*. The testing phase is crucial and it is imperative that we don't use the same data for testing that we used for training, doing so would be like giving a student the answers before the test. The accuracy would be artificially high as they'd already seen the right answers. Once we have a good test of the model done we can use it. Ideally, we'd want the model to give high accuracy, but that can be subjective. For some applications we might need 99% or greater accuracy, in others just getting an answer better than random would do.

2.4 Measuring accuracy

Measuring accuracy of a model in the testing phase is less straightforward than we might first think We might assume that all we have to do is count the number of test cases that we got correct, but that is only one quarter of the story at best!. In fact, for a binary classification (a model that knows only two groups, e.g in our animal example a model that can say whether it thinks something is or isn't a cat) there are two ways to be right and two ways to be wrong and we must calculate as many of these we can in order to get a good accuracy estimate. For a model with more than two groups or for models trying to predict a quantity rather than a group the question is more complicated and we'll look at those later.

2.4.1 Two ways to be right: True Positives and True Negatives

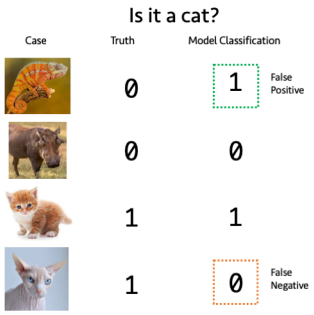
The two ways to be right are to get a correct positive classification - a True Positive and a correct negative classification - a True Negative. These are easier to understand graphically. In the figure below we have a set of trained model generated answers and their true classes.



A True Positive occurs when the model classifies a case positively (is a cat) and is correct, similarly a True Negative occurs when the model classifies a case negatively (is not a cat) and is correct.

2.4.2 Two ways to be wrong: False Postive and False Negatives

The two ways to be wrong are False Positive and false negative classifications False Positives and False Negatives. A False Positive occurs when the model classifies a non-cat as a cat and a False Negative occurs when the model classifies a cat as not a cat.



2.4.3 Sensitivity and Specificity

For a given set of test data for which we know the true labels, we run the model and get it's classifications. We can count up the True/False Positive/Negatives and calculate two quantities Sensitivity and Specificity. Sensitivity tells us roughly what proportion of True Positives we got, given the errors and Specificity tells how few wrong calls we made. The two measures are therefore complementary and are used together to get a picture of how well the model performs. A good model is high in both. The quantities are calculated as follows

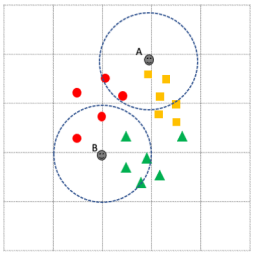
$Sensitivity = \frac{TP}{TP+FN}$ $Specificity = \frac{TN}{TN+FP}$

2.4.4 Other measures of accuracy

There are in fact, many other measures of accuracy in use beyond sensitivity and specificity. These include things called F scores, precision and recall, FDRs and (confusingly) one actually called accuracy. It's important to know that they are all a bit different and give different measures but they all try to capture the 'rightness' or 'accuracy' of our classifiers. As we try out different tools we will see other measures.

2.5 k-Nearest Neighbours

The k-Nearest Neighbour algorithm is a multi-class capable classification algorithm. Like the unsupervised methods this relies on distance measures between cases/elements and tries to apply a class to an unknown element by looking at the number of nearest neighbours classes. Roughly, the unknown case gets the class of the majority of the k nearest neighbours. We can see an example in the figure below



If we set k to be 5 then Unknown case A has 3 orange squares and 2 red circles in its 5 nearest neighbours, so unknown case A would be classified as an orange square. Similarly, unknown case B has more green triangles in its k nearest neighbours so it gets classified as a green triangle. Note how the known class labels are crucial in putting the unknown cases into classes. This approach only works because we have some known examples. Also note how much harder the algorithm would find the task if there were too few examples of each class. For this and many other types of supervised learning algorithm, the more training data we have, the better.

2.5.1 Training and evaluating kNN

Let's run through using the algorithm with the data below. The first phase is training and evaluation. There are 3 sets we will use, a training set of 55 points, which is labelled in a separate vector (`train_data` and `train_labels`), a test set of 20 points that is labelled (`test_data` and `test_labels`) and an unlabelled, unknown data set of 75 points that we wish to label using k Nearest Neighbours.

```
dplyr::glimpse(train_data)
```

```
Rows: 55
Columns: 4
$ measure1 <dbl> 6.7, 5.4, 6.4, 5.0, 5.3, 5.4, 5.4, 5.7, 6.0, 5.1, 6.0, 5.8, 4~
$ measure2 <dbl> 3.1, 3.4, 3.2, 3.2, 3.7, 3.9, 3.0, 4.4, 2.9, 3.8, 2.2, 2.6, 3~
$ measure3 <dbl> 5.6, 1.7, 5.3, 1.2, 1.5, 1.7, 4.5, 1.5, 4.5, 1.9, 4.0, 4.0, 1~
$ measure4 <dbl> 2.4, 0.2, 2.3, 0.2, 0.2, 0.4, 1.5, 0.4, 1.5, 0.4, 1.0, 1.2, 0~
```

```
train_labels
```

```
 [1] C A C A A A B A B A B B A A B C C A A B A B B C A A A A C C B C B A B C C B
[39] A A B C B C C A C B A A A B B C B
```

```
Levels: A B C
```

```
  dplyr::glimpse(test_data)
```

```
Rows: 20
Columns: 4
$ measure1 <dbl> 5.6, 6.7, 6.3, 6.3, 5.0, 7.2, 6.2, 6.7, 4.6, 5.1, 6.0, 6.7, 7~
$ measure2 <dbl> 3.0, 2.5, 3.3, 2.7, 2.3, 3.6, 2.2, 3.1, 3.4, 3.8, 2.2, 3.3, 3~
$ measure3 <dbl> 4.1, 5.8, 6.0, 4.9, 3.3, 6.1, 4.5, 4.7, 1.4, 1.6, 5.0, 5.7, 6~
$ measure4 <dbl> 1.3, 1.8, 2.5, 1.8, 1.0, 2.5, 1.5, 1.5, 0.3, 0.2, 1.5, 2.1, 2~
```

```
  test_labels
```

```
 [1] B C C C B C B B A A C C C A A B A A B A
Levels: A B C
```

```
  dplyr::glimpse(unknown_data)
```

```
Rows: 75
Columns: 4
$ measure1 <dbl> 5.6, 5.0, 6.3, 6.1, 5.8, 5.5, 5.1, 5.1, 7.2, 5.0, 4.7, 7.7, 5~
$ measure2 <dbl> 2.9, 3.6, 2.8, 2.8, 2.7, 2.6, 3.8, 3.7, 3.0, 3.0, 3.2, 2.8, 3~
$ measure3 <dbl> 3.6, 1.4, 5.1, 4.7, 5.1, 4.4, 1.5, 1.5, 5.8, 1.6, 1.6, 6.7, 1~
$ measure4 <dbl> 1.3, 0.2, 1.5, 1.2, 1.9, 1.2, 0.3, 0.4, 1.6, 0.2, 0.2, 2.0, 0~
```

The first step is to train and test a model. As we are going to go through the process twice (one evaluating, one with unknown data), we must remember to control the random element of the algorithm. `set.seed()` with a consistent argument (`123`) puts the random number generator back to the same place each time allowing reproducibility.

The `knn()` function is in the `class` package so we load that and pass it the `train_data` to learn from and the known `test_data` to predict groups on. The `cl` parameter gets the vector of `train_labels`. Finally the k nearest neighbours is passed as `k`, here `9`.

```
  set.seed(123)
  library(class)
  test_set_predictions <- knn(train_data, test=test_data, cl = train_labels, k=9)
```

```
test_set_predictions
```

```
 [1] B C C C B C B B A A B C C A A B A A B A
Levels: A B C
```

As we can see, the predictions are returned as vector whose elements correspond to the rows of `test_data`. We can check the accuracy of the predictions by comparing the predictions with the known labels. The `caret` package function `confusionMatrix()` returns an object with lots of useful information.

```
library(caret)
confusionMatrix(test_set_predictions, test_labels)
```

```
Confusion Matrix and Statistics

          Reference
Prediction A B C
         A 7 0 0
         B 0 6 1
         C 0 0 6

Overall Statistics

               Accuracy : 0.95
                 95% CI : (0.7513, 0.9987)
    No Information Rate : 0.35
    P-Value [Acc > NIR] : 2.903e-08

                  Kappa : 0.9251

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: A Class: B Class: C
Sensitivity             1.00   1.0000   0.8571
Specificity             1.00   0.9286   1.0000
Pos Pred Value          1.00   0.8571   1.0000
Neg Pred Value          1.00   1.0000   0.9286
Prevalence              0.35   0.3000   0.3500
```

```
Detection Rate           0.35   0.3000   0.3000
Detection Prevalence     0.35   0.3500   0.3000
Balanced Accuracy        1.00   0.9643   0.9286
```

At the top of the output, the confusion matrix shows how 'mixed' up the model got. Read it down the columns, so that for the 7 real group `A` the algorithm predicted 7 `A`, 0 `B` and 0 `C`; for the 6 real group `B` the algorithm predicted 0 `A`, 6 `B` and 0 `C` and for the 7 real group `C` the predictions were 0 `A`, 1 `B` and 6 `C`, so a `C` was misclassified as a `B`. This error rate and pattern is reflected in the overall accuracy, stated as 95 % and the more useful per group Sensitivity and Specificity, the lower Specificity for group `B` is due to the `C` miscalled as a `B` (so a false positive `B`). The same error causes the lower Sensitivity for group `C`.

2.5.2 Using a trained model

Now that we have evaluated the model and know how accurate it is - and that it is accurate enough to be useful, we can run on our unknown data. This is virtually identical to before, replacing the `test_data` with the `unknown_data`. We must remember to reset the random number generator again, and we can go ahead and add the predictions straight to the data frame if we wish. We now have predicted groups for the unknown data and an estimate of the accuracy of our predictions.

```
set.seed(123)
unknown_predictions <- knn(train_data, test=unknown_data, cl = train_labels, k=9)
unknown_data$predicted_group <- unknown_predictions
dplyr::glimpse(unknown_data)
```

```
Rows: 75
Columns: 5
$ measure1       <dbl> 5.6, 5.0, 6.3, 6.1, 5.8, 5.5, 5.1, 5.1, 7.2, 5.0, 4.7,~
$ measure2       <dbl> 2.9, 3.6, 2.8, 2.8, 2.7, 2.6, 3.8, 3.7, 3.0, 3.0, 3.2,~
$ measure3       <dbl> 3.6, 1.4, 5.1, 4.7, 5.1, 4.4, 1.5, 1.5, 5.8, 1.6, 1.6,~
$ measure4       <dbl> 1.3, 0.2, 1.5, 1.2, 1.9, 1.2, 0.3, 0.4, 1.6, 0.2, 0.2,~
$ predicted_group <fct> B, A, C, B, C, B, A, A, C, A, A, C, A, C, B, B, C, A, ~
```

2.6 Random Forest

Random Forest is another supervised learning algorithm that is based on ensembles of decision trees. A decision tree is a model that resembles a question flowchart that has a 'question' at

each branch point and continues until enough have been 'asked' to differentiate the item in hand. Here is one potential decision tree for the animal classification we've been using.

In a Random Forest classifier trees are made using the training data and the ones that are best at classifying the data are retained. There are a whole set of possible good trees so the ensemble of trees is used, hence Random Forest. The many trees make up one model that are used with unseen data.

2.6.1 Building a Random Forest Model

We use the `randomForest` package to do this, and we will use the training and test data as we did with k nearest neighbours above, for random forest, the labels are specified in the data, so we don't have a separate label vector and must now add them on to the training and test data. Let's do that first

```
train_data$group <- train_labels
dplyr::glimpse(train_data)
```

```
Rows: 55
Columns: 5
$ measure1 <dbl> 6.7, 5.4, 6.4, 5.0, 5.3, 5.4, 5.4, 5.7, 6.0, 5.1, 6.0, 5.8, 4~
$ measure2 <dbl> 3.1, 3.4, 3.2, 3.2, 3.7, 3.9, 3.0, 4.4, 2.9, 3.8, 2.2, 2.6, 3~
$ measure3 <dbl> 5.6, 1.7, 5.3, 1.2, 1.5, 1.7, 4.5, 1.5, 4.5, 1.9, 4.0, 4.0, 1~
$ measure4 <dbl> 2.4, 0.2, 2.3, 0.2, 0.2, 0.4, 1.5, 0.4, 1.5, 0.4, 1.0, 1.2, 0~
$ group    <fct> C, A, C, A, A, A, B, A, B, A, B, B, A, A, B, C, C, A, A, B, A~
```

We can now build the model with the `randomForest()` function. The setup uses R's formula based syntax, so is very similar to that we used for linear models. The `group` is to be predicted based on `.` which means all other columns in the data `train_data`. The `model` variable holds the trained model

```
library(randomForest)
model <- randomForest(group ~ ., data = train_data, mtry=2)
```

2.6.2 Testing a Random Forest model

With the model built we can use the generic `predict()` function to get the model to predict groups for the unlabelled `test_data` then compare it to the real groups with `confusionMatrix()`. Setting the value of `type` to `class` tells the `predict()` we want group classifications

```
test_set_predictions <- predict(model, test_data, type="class")
confusionMatrix(test_set_predictions, test_labels)
```

```
Confusion Matrix and Statistics

          Reference
Prediction A B C
         A 7 0 0
         B 0 6 1
         C 0 0 6

Overall Statistics

               Accuracy : 0.95
                 95% CI : (0.7513, 0.9987)
    No Information Rate : 0.35
    P-Value [Acc > NIR] : 2.903e-08

                  Kappa : 0.9251

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: A Class: B Class: C
Sensitivity              1.00   1.0000   0.8571
Specificity              1.00   0.9286   1.0000
Pos Pred Value           1.00   0.8571   1.0000
Neg Pred Value           1.00   1.0000   0.9286
Prevalence               0.35   0.3000   0.3500
```

```
Detection Rate           0.35   0.3000   0.3000
Detection Prevalence     0.35   0.3500   0.3000
Balanced Accuracy        1.00   0.9643   0.9286
```

The model is again, convincing and highly accurate so we can repeat use `predict()` with `model` to get predictions for `unknown_data`, and again add it to the data

```
unknown_predictions <- predict(model, unknown_data, type="class")
unknown_data$predicted_group <- unknown_predictions
dplyr::glimpse(unknown_data)
```

```
Rows: 75
Columns: 5
$ measure1       <dbl> 5.6, 5.0, 6.3, 6.1, 5.8, 5.5, 5.1, 5.1, 7.2, 5.0, 4.7,~
$ measure2       <dbl> 2.9, 3.6, 2.8, 2.8, 2.7, 2.6, 3.8, 3.7, 3.0, 3.0, 3.2,~
$ measure3       <dbl> 3.6, 1.4, 5.1, 4.7, 5.1, 4.4, 1.5, 1.5, 5.8, 1.6, 1.6,~
$ measure4       <dbl> 1.3, 0.2, 1.5, 1.2, 1.9, 1.2, 0.3, 0.4, 1.6, 0.2, 0.2,~
$ predicted_group <fct> B, A, C, B, C, B, A, A, C, A, A, C, A, C, B, B, C, A, ~
```

2.6.3 Random Forest with categorical predictors

In our kNN example and the previous Random Forest predictor, the input data features were solely numeric. Random Forest can handle a mixture of numeric and character or categorical based features allowing us to make classifications on more than numbers. The process is similar, so let's get some appropriate data and do that

```
dplyr::glimpse(train_data_mixed)
```

```
Rows: 55
Columns: 6
$ measure1 <dbl> 6.7, 5.4, 6.4, 5.0, 5.3, 5.4, 5.4, 5.7, 6.0, 5.1, 6.0, 5.8, 4~
$ measure2 <dbl> 3.1, 3.4, 3.2, 3.2, 3.7, 3.9, 3.0, 4.4, 2.9, 3.8, 2.2, 2.6, 3~
$ measure3 <dbl> 5.6, 1.7, 5.3, 1.2, 1.5, 1.7, 4.5, 1.5, 4.5, 1.9, 4.0, 4.0, 1~
$ measure4 <dbl> 2.4, 0.2, 2.3, 0.2, 0.2, 0.4, 1.5, 0.4, 1.5, 0.4, 1.0, 1.2, 0~
$ group    <fct> C, A, C, A, A, A, B, A, B, A, B, B, A, A, B, C, C, A, A, B, A~
$ colour   <fct> White, Green, White, Green, Green, Green, Blue, Green, Blue, ~
```

```
dplyr::glimpse(test_data_mixed)
```

```
Rows: 20
Columns: 6
$ measure1 <dbl> 5.6, 6.7, 6.3, 6.3, 5.0, 7.2, 6.2, 6.7, 4.6, 5.1, 6.0, 6.7, 7~
$ measure2 <dbl> 3.0, 2.5, 3.3, 2.7, 2.3, 3.6, 2.2, 3.1, 3.4, 3.8, 2.2, 3.3, 3~
$ measure3 <dbl> 4.1, 5.8, 6.0, 4.9, 3.3, 6.1, 4.5, 4.7, 1.4, 1.6, 5.0, 5.7, 6~
$ measure4 <dbl> 1.3, 1.8, 2.5, 1.8, 1.0, 2.5, 1.5, 1.5, 0.3, 0.2, 1.5, 2.1, 2~
$ group    <fct> B, C, C, C, B, C, B, B, A, A, C, C, C, A, A, B, A, A, B, A
$ colour   <fct> Blue, White, White, White, Blue, White, Blue, Blue, Green, Gr~
```

```
dplyr::glimpse(unknown_data_mixed)
```

```
Rows: 75
Columns: 6
$ measure1 <dbl> 5.6, 5.0, 6.3, 6.1, 5.8, 5.5, 5.1, 5.1, 7.2, 5.0, 4.7, 7.7, 5~
$ measure2 <dbl> 2.9, 3.6, 2.8, 2.8, 2.7, 2.6, 3.8, 3.7, 3.0, 3.0, 3.2, 2.8, 3~
$ measure3 <dbl> 3.6, 1.4, 5.1, 4.7, 5.1, 4.4, 1.5, 1.5, 5.8, 1.6, 1.6, 6.7, 1~
$ measure4 <dbl> 1.3, 0.2, 1.5, 1.2, 1.9, 1.2, 0.3, 0.4, 1.6, 0.2, 0.2, 2.0, 0~
$ colour   <fct> Blue, Green, White, Blue, White, Blue, Green, Green, White, G~
$ group    <fct> B, A, C, B, C, B, A, A, C, A, A, C, A, C, B, B, C, A, C, B, A~
```

We can see that there is a new categorical feature called `colour` in our train and test data, but not in our unknown data, so let's try to predict the colour this time.

```
model2 <- randomForest(colour ~ ., data = train_data_mixed, mtry=2)

test_set_mixed_predictions <- predict(model2, test_data_mixed, type="class")
confusionMatrix(test_set_mixed_predictions, test_labels_mixed)
```

```
Confusion Matrix and Statistics

          Reference
Prediction Blue Green White
     Blue     6     0     0
     Green    0     7     0
     White    0     0     7

Overall Statistics

               Accuracy : 1
```

```
                 95% CI : (0.8316, 1)
    No Information Rate : 0.35
    P-Value [Acc > NIR] : 7.61e-10

                  Kappa : 1

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: Blue Class: Green Class: White
Sensitivity                  1.0         1.00         1.00
Specificity                  1.0         1.00         1.00
Pos Pred Value               1.0         1.00         1.00
Neg Pred Value               1.0         1.00         1.00
Prevalence                   0.3         0.35         0.35
Detection Rate               0.3         0.35         0.35
Detection Prevalence         0.3         0.35         0.35
Balanced Accuracy            1.0         1.00         1.00
```

So we have created a model that is capable of perfectly predicting the value of the categoric value colour from a mixture of numeric and categoric features. Why is the model so accurate? It's a bit of a fix! This sample data has a direct mapping between the `group` and the `colour`: `A` is always `Green`, `B` is always `Blue` and `C` is always `White` so it is easy to predict colour if you have `group`. The data aren't typical in this sense but it does highlight the procedure.

2.6.4 Random Forest Regression

It is also possible to perform prediction of numeric values and not just classes with Random Forest. We simply set up the model with a numeric value as the predicted value in the formula as follows

```
model3 <- randomForest(measure1 ~ ., data = train_data_mixed, mtry=2)
```

Now when we use `predict()` and omit the `type` argument, we get a set of numbers, not classes back

```
test_set_mixed_predictions_numeric <- predict(model3, test_data_mixed)
test_set_mixed_predictions_numeric
```

```
      89       109       101       124        94       110        69        87
5.971972 6.240498 6.900242 6.008055 5.519931 6.887112 5.696763 6.220078
       7        47       120       125       118         1        15        96
5.036305 5.179204 5.836345 6.723473 6.890082 5.087420 5.108052 5.988062
      24        38        88        48
5.259467 5.011159 5.739434 4.832837
```

2.6.4.1 Evaluating numeric predictions

Previously we've evaluated predictions from our models for classes, counting True Positives etc, but we can't do that here because we have no classes. Instead we can calculate how far away from the real value the predictions are on average. That's a simple sum to do in R

```
mean( (test_data_mixed$measure1 - test_set_mixed_predictions_numeric) ^ 2)
```

```
[1] 0.1747463
```

The quantity is called the Mean Squared Error or MSE. The lower the better, though the actual size is dependent on context. The context here is the descriptive statistics of the known values for the test data, which we can get with `summary()`

```
summary(test_data_mixed$measure1)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4.60    5.10    5.90    5.88    6.40    7.70
```

These values range between 4.6 and 7.7, with 50% of them lying between 5.1 and 6.4. With that in mind it seems like an MSE of 0.17 is a pretty good result and we can conclude to predict accurately the values of `measure1` from our Random Forest Regression model.

> **Roundup**
>
> - Supervised Learning uses labelled data to make predictions on unseen data
> - Random Forest can predict classes and numeric values (perform regression)
> - It is imperative to evaluate the predictive model on a set of known cases

3 Deep Learning

1. Questions
    - What is Deep Learning?
    - How is Deep Learning distinct from classical Machine Learning?
2. Objectives
    - Discuss how cryptic patterns can find their own important features in arbitrary data sets
    - Study the outline of training a Neural Network
    - Build and test a simple Neural Network
3. Key Points
    - Deep Learners find important features and patterns in the data automatically
    - Neural Networks use optimised weights to learn classifications
    - Power comes at the expense of interpretability

3.1 About this chapter

In this chapter we'll look at the latest advance in Machine Learning, a special set of extremely powerful techniques called Deep Learning. Deep learning methods are the ones that have been used in headline grabbing Artificial Intelligence toold from large scale facial recognition, data mining to influence voters on social media, voice recognition like Siri and Alexa, Netflix's suggestion algorithm, Google's advertising algorithm, and in science things like AlphaFold - the protein folding prediction tool and medical image analysis.

These tools and algorithms are a wide family of their own and have properties distinct from the machine learning techniques we've looked at so far, principally these types are able to select the most important features themselves and work out what is the most reliable data. They are also much more complicated in practice and much more dense so they become a black box and we are less able to interpret how they are making the decisions they make. This is the trade off we make when using Deep Learning.

3.2 Feature Selection in Deep Learning

In the previous tools we've looked at we used a *np* feature matrix, with p features - a column of data for each thing we measured. Feature selection is really important and can make or break the usefulness of a machine learning tool. If the features we select can't differentiate between the classes, then the machine learning tool will never be able to make good predictions. Consider what it would be like if we tried to work out a person's hair colour from their height! Height is an easy thing to measure but does it ever predict a person's hair?

So we must pick our features carefully if we're to make use of ML generally, but with Deep Learning the algorithms themselves work out which are the most useful features and also patterns within the features and preferentially use them. This leads to a bit of a kitchen sink approach, we can take all the features we like, pump them into a Deep Learning algorithm and let it decide the best way to use them. A practical upside of this then is that our *np* feature matrix can become very complicated and we can start to squeeze pretty much anything into the training data. We simply have to be able to encode it as numbers somehow.

3.3 Cryptic patterns in Deep Learning

The ability to automatically select features or patterns to use means that the algorithms can find and use patterns that we don't specify explicitly and in fact don't even know about. To understand this, let's work through a protein sequence based example. Our first issue with biological sequences is the question of how to encode it as numbers. One common way of taking a categoric thing and making it numeric is to use 'One Hot Encoding'. Which looks like this:

AMINO ACID

POSITION IN SEQUENCE	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

This encoding represents the protein sequence 'ACDE', the columns represent the alphabet of amino acids (in alphabetic order), the rows represent the position of sequence. We add a `1` at the intersection of the position and amino acid to show the amino acid at each position. Each row therefore has only one row.

Once we have an encoding, patterns will start to appear that the algorithms can use. Consider a protein motif, like RHLR - that would look like this in our one hot encoding.

AMINO ACID

POSITION IN SEQUENCE	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
x+1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
x+2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
x+3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Now wherever the pattern crops up the algorithm will see it. The pattern can become associated with a particular class and used as part of the signal for classification. We didn't have

to say 'protein has RHLR', if it's an important pattern and associated with a class or group then the algorithm will use it.

3.3.1 Implications of Cryptic Patterns

Being able to select its own features and patterns means that the Deep Learning methods get a special sort of sensitivity. Considering our protein example again, then lots of properties of the proteins that are reliant on sequence at some level will be detectable and useable in some way by the Deep Learner. Things like physico-chemical sequence properties such as hydrophobicity are reliant on the actual amino acids to exist so they can be captured and used.

An important thing to note is that the patterns have only to be associated, not *over-represented*, on the whole. The Deep Learners might find a pattern that occurs only a few times in millions of example data, but if it is associated pretty uniquely with just one class or group then it can be used. This stands in contrast to typical methods of pattern finding in bioinformatics, which use majority or statistical over-representation. The patterns often have weight with other patterns and these associations increase the patterns power too.

The ability to find cryptic patterns and make associations is reliant on having a great deal of training data. Deep Learning methods do require lots more data than the ML methods we've already looked at and this can be a drawback in practice.

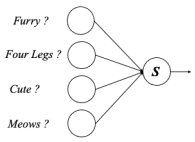
Deep Learning models internal representations become very large and hard to interpret, so that actually understanding what they're using to classify upon can become impossible. This is a significant trade off for the high power that we can get

3.4 Neural Networks

The core of most Deep Learning models and model types is the neural network, let's run through how that works to gain some insight into how it gets its power.

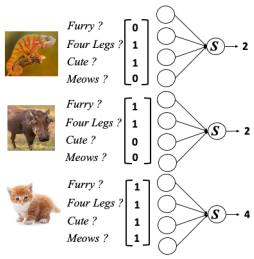
3.4.1 The Perceptron

Neural networks are made up of units called Perceptrons, these are mathematical structures inspired by biological neurons. They take multiple inputs, integrate them in some way and produce an output. One that worked on our animal matrix might look like this



3.4.2 The Network

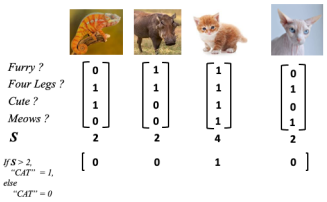
Combining lots of perceptrons results in a neural network and at a basic level might work with our animal data like this,



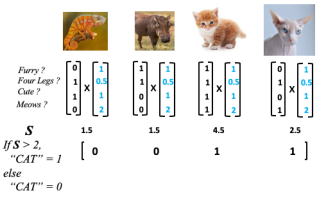
Making sense of all the integrations from the neural network, that is the calling of a class (in this case `is_a_cat`) is done by a decision function, here that may look like this

```
if S > 2,
    cat = 1
else
    cat = 0
```

Applied to the animal neural network the animal data classifications end up like this



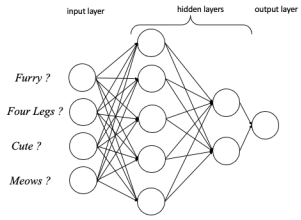
Only one of the actual cats was correctly labelled. The clever part of the neural network is to apply weights to each of the features that modify the value they add to the neural network (in the figure below as blue values).



Weights work to give the more useful features higher values (like `meows`) and less useful features lower values (like `four legs`). The network can now more accurately classify the animals in the picture.

3.4.3 Neural Network Structure

The networks needn't be restricted to simple structures in which the initial inputs go straight to the output, many layers of neurons can be made, each arbitrary numbers of neurons deep. These extra layers are called the hidden layers



The hidden layers increase the power of the neural network by allowing for further integration of information and extra weighting. But they also make the network more obscure and hard to read, again this is how the power of the neural network comes at the expense of interpretability.

3.4.4 Training to find weights

The main part of training the neural network and the place it makes itself powerful is in the weight finding phase. This is called the learning or training phase. To do this the training algorithm goes back and forth across the network methodically adjusting the weights until it sees no further improvement when classifying on the training data - it is constantly comparing its current state against the answers in the training data.

3.4.5 Neural network training phases are long and involved

As you can see there is a lot about neural networks to be specified and optimised. The number and depth of hidden layers that is optimal varies for each data set and there is no rule to follow as to what will be best. It is also not true that bigger is always better. The weights of the neural network must also be optimised for every data set, and we must be careful to use training data that is distinct from our test data to be confident in the generality of our resulting model. As a result of these considerations the training and testing phases of neural networks are particularly involved.

We won't go through that whole procedure here, though you should be aware of it as it is the key to a truly useful deep learning model. But we will try out a small neural network in R.

3.5 A simple neural network in R

3.5.1 Frog Data

In this example we shall use some data on amphibian presence at various sites. Here's a `glimpse()` of the `train_rows`, we also have a `test_rows`

```
dplyr::glimpse(train_rows)
```

```
Rows: 94
Columns: 20
$ SR                <dbl> 1000, 100, 200, 30000, 10050, 700, 50, 8000, 2500, ~
$ NR                <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 2, 1, 1, 1, ~
$ TR                <dbl> 1, 1, 5, 1, 1, 5, 1, 1, 1, 2, 1, 1, 1, 1, 14, 1, 14~
$ VR                <dbl> 3, 2, 1, 3, 2, 2, 2, 3, 3, 0, 1, 2, 3, 2, 3, 3, 1, ~
$ SUR1              <dbl> 2, 2, 10, 1, 1, 10, 2, 2, 10, 6, 2, 2, 2, 2, 7, 2, ~
$ SUR2              <dbl> 1, 7, 6, 1, 10, 6, 7, 10, 2, 9, 7, 7, 2, 10, 2, 2, ~
$ SUR3              <dbl> 9, 6, 10, 1, 6, 9, 10, 7, 6, 2, 6, 9, 1, 10, 1, 7, ~
$ UR                <dbl> 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 0, ~
$ FR                <dbl> 0, 0, 4, 0, 1, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 0, 0, ~
$ OR                <dbl> 100, 100, 75, 100, 100, 100, 100, 100, 100, 50, 100~
$ RR                <dbl> 2, 2, 1, 2, 5, 1, 5, 9, 0, 0, 0, 0, 5, 1, 5, 0, 5, ~
$ BR                <dbl> 5, 2, 1, 10, 5, 1, 5, 9, 1, 0, 0, 0, 5, 5, 5, 1, 5,~
$ MR                <dbl> 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ CR                <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ Green_frogs       <dbl> 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, ~
$ Brown_frogs       <dbl> 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, ~
$ Common_toad       <dbl> 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, ~
```

```
$ Tree_frog          <dbl> 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, ~
$ Common_newt        <dbl> 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, ~
$ Great_crested_newt <dbl> 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, ~
```

These data are from (Blachnik, Sołtysiak, and Dąbrowska 2019) originally and you can see a description of the 20 columns at https://archive.ics.uci.edu/ml/datasets/Amphibians. Briefly, they are things like the presence and size and maintenance of reservoirs and the surrounding area, whether humans use the area, whether there is fishing. All potentially pertinent measurements. The presence of different types of amphibian species are recorded as 1 for present, and 0 for not present.

We could use a neural network to predict any of the species listed, but let's work on predicting `Green_frogs`.

We'll use the straightforward `neuralnet()` function in the `neuralnet` package for this. It can take an R formula specification, which as you'll recall takes the form `y ~ feature_1 + feature_2 ...` where `y` is the thing to be predicted and `feature_x` are the features to input for prediction with. With 19 to enter, that's lots of typing, so I've squashed it into a variable called `long_formula`

```
long_formula
```

```
Green_frogs ~ SR + NR + TR + VR + SUR1 + SUR2 + SUR3 + UR + FR +
    OR + RR + BR + MR + CR + Green_frogs + Brown_frogs + Common_toad +
    Tree_frog + Common_newt + Great_crested_newt
```

3.5.2 Training a 3 hidden layer neural network

We can put the formula into the function `neuralnet()` specify the training data and the depth of the hidden layers. Here we'll have 3, with 15, 10 and 5 neurons respectively.

```
library(neuralnet)
nn <- neuralnet(long_formula, train_rows, hidden=c(15,10,5), linear.output=FALSE)
```

That single step builds the neural network, and trains it and gives it back to use so we can use it to make predictions with. Of course the first thing we want to make predictions on is our test set so we can evaluate the accuracy.

3.5.3 Testing the neural network

The `compute()` function takes a neural network model and data and creates predictions. Here we feed it our test data. However when we look at the resulting predictions (stored in the `net.result` slot in our `predictions` object) we see something odd.

```
predictions <- compute(nn, test_rows)
head(predictions$net.result)
```

```
            [,1]
[1,] 0.998154554
[2,] 0.998147802
[3,] 0.008503821
[4,] 0.998154554
[5,] 0.996971835
[6,] 0.006267082
```

The predictions are not of classes, but are actually numbers that represent the level of sureness the model has that the site has Green Frogs. This value is sometimes useful, but we need to convert it to classes to evaluate it. As the values run between 0 and 1 we can do that by simple rounding so that any prediction over 0.5 is considered a present prediction, anything below is consider an absent predictions (other algorithms and functions exist for this conversion).

```
binary_predictions <- round(predictions$net.result,digits=0)
```

We can then put those binarised predictions into the `confusionMatrix()` function we used previously alongside the true values from the `test_rows` data (remembering to convert them to `factors` as they are not already).

```
library(caret)
confusionMatrix(factor(test_rows$Green_frogs), factor(binary_predictions))
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 30  4
         1 19 42

               Accuracy : 0.7579
```

```
               95% CI : (0.6592, 0.8399)
    No Information Rate : 0.5158
    P-Value [Acc > NIR] : 1.097e-06

                  Kappa : 0.5201

 Mcnemar's Test P-Value : 0.003509

            Sensitivity : 0.6122
            Specificity : 0.9130
         Pos Pred Value : 0.8824
         Neg Pred Value : 0.6885
             Prevalence : 0.5158
         Detection Rate : 0.3158
   Detection Prevalence : 0.3579
      Balanced Accuracy : 0.7626

       'Positive' Class : 0
```

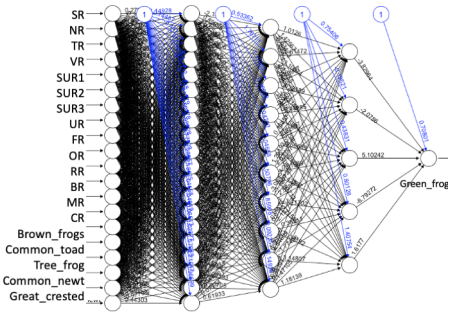
We can see the resulting network has about 60% sensitivity and 90% specificity, so missing a lot of real green frog sites.

As you can imagine the exact choice of the parameters can make a difference on final neural network performance. This is just one instance. In real analyses we would try out many different hidden layer and other parameter configurations and select the best performing at the the testing stage. That may then even move on to a further fine tuning stage, the development of machine learners is art as much as it is science.

3.5.4 Examining the structure of the neural network

The `neuralnet` package we used here was chosen not least because it is straightforward and fast, but also because it is possible to get a plot of the created network.

```
plot(nn)
```



Note that the first layer corresponds to the input columns in the data with one neuron each, these then feed into the 3 hidden layers we specified of 20,15 and 5 layers each and finally the one neuron layer intergrating everything to give us the final prediction on whether we have a Green Frog.

We can see that the neural network we made is really complex. Even with just the small number of input features and hidden layers we have the combinations of weights and their effect into the next layer is too hard to understand (even if the plot were readable). This shows us how neural network structures become black boxes, we can't be sure which of the input variables (or combinations of which) were most important in making the classifications.

Roundup

- Deep Learners choose their own features
- Deep Learners like neural networks can work on patterns we dont explicitly state
- Neural network training means finding weights that give the best classifications
- Neural networks are black boxes and hard to interpret

3.6 References

Prerequisites

Knowledge prerequisites

There are no specific knowledge prerequisites for this book but it will be very helpful if you have read and worked through the `Data Driven Visualisation`, `Data Exploration`, `Intro to Stats`, `Estimation Statistics` and `Bayesian Inference` books and are familiar with R use.

Software prerequisites

You need to install the following stuff for this book:

1. R
2. RStudio
3. Some R packages: `tidyverse`, `factoextra`, `patchwork`

Installing R

Follow this link and install the right version for your operating system https://www.stats.bris.ac.uk/R/

Installing RStudio

Follow this link and install the right version for your operating system https://www.rstudio.com/products/rstudio/download/

Installing R packages in RStudio

Standard packages

In the RStudio console, type

```
install.packages(c("tidyverse","factoextra", "patchwork", "ggdendro", "class",
"caret", "e1071","randomForest", "neuralnet"))
```

R Fundamentals

About this chapter

1. Questions:

- How do I use R?

2. Objectives:

- Become familiar with R syntax
- Understand the concepts of objects and assignment
- Get exposed to a few functions

3. Keypoints:

- R's capabilities are provided by functions
- R users call functions and get results

Working with R

In this workshop we'll use R in the extremely useful RStudio software. For the most part we'll work interactively, meaning we'll type stuff straight into the R console in RStudio (Usually this is a window on the left or lower left) and get our results there too (usually in the console or in a window on the right).

Panels like the ones below mimic the interaction with R and first show the thing to type into R, and below the calculated result from R.

Let's look at how R works by using it for it's most basic job - as a calculator:

```
3 + 5
```

```
[1] 8
```

```
12 * 2
```

```
[1] 24
```

```
1 / 3
```

```
[1] 0.3333333
```

```
12 * 2
```

```
[1] 24
```

Fairly straightforward, we type in the expression and we get a result. That's how this whole book will work, you type the stuff in, and get answers out. It'll be easiest to learn if you go ahead and copy the examples one by one. Try to resist the urge to use copy and paste. Typing longhand really encourages you to look at what you're entering.

As far as the R output itself goes, it's really straightforward - its just the answer with a `[1]` stuck on the front. This `[1]` tells us how many items through the output we are. Often R will return long lists of numbers and it can be helpful to have this extra information.

Variables

We can save the output of operations for later use by giving it a name using the assignment symbol `<-`. Read this symbol as 'gets', so `x <- 5` reads as 'x gets 5'. These names are called variables, because the value they are associated with can change.

Let's give five a name, `x` then refer to the value 5 by it's name. We can then use the name in place of the value. In the jargon of computing we say we are assigning a value to a variable.

```
x <- 5
x
```

```
[1] 5
```

```
x * 2
```

```
[1] 10
```

```
y <- 3
x * y
```

```
[1] 15
```

This is of course of limited value with just numbers but is of great value when we have large datasets, as the whole thing can be referred to by the variable.

Using objects and functions

At the top level, R is a simple language with two types of thing: functions and objects. As a user you will use functions to do stuff, and get back objects as an answer. Functions are easy to spot, they are a name followed by a pair of brackets. A function like `mean()` is the function for calculating a mean. The options (or arguments) for the function go inside the brackets:

```
sqrt(16)
```

```
[1] 4
```

Often the result from a function will be more complicated than a simple number object, often it will be a vector (simple list), like from the `rnorm()` function that returns lists of random numbers

```
rnorm(100)
```

```
  [1]  0.89122179  0.33644209  0.12738436 -1.09068085  1.28592008  0.87783702
  [7] -0.80190747 -0.32031939  0.92835701  1.17415709 -2.24815203  0.62973420
 [13]  0.09667571 -0.25977006 -0.61583704 -0.34784295 -0.61209335 -1.89338375
 [19] -1.03291003 -1.22343384  0.84989698  0.71120815 -0.39044344  0.71148555
 [25]  0.17038429  0.28986583  0.87745151  0.68179310  0.48640292  0.61270488
 [31] -1.10992469  1.03928353  1.67458293  0.08192491  0.05875668  0.69437266
 [37] -1.63467147 -0.03475879  0.63361050  1.22245180  0.01049303  2.21231496
 [43] -0.06748033  1.25851974 -0.23502419  1.19873205 -0.74351510  0.45972360
 [49]  1.34299742 -0.43509793 -0.89420213 -0.08935856 -0.96822585 -0.74948787
 [55]  0.74986468  1.79038933  0.35565569 -0.32114546 -0.89059614 -0.67362470
 [61]  0.34268813  1.29511613  0.72194351  2.09045237  1.45378209  1.67512713
```

```
 [67] -0.77756941 -1.20659631 -0.57080254 -0.75364600  0.07498261  0.53002655
 [73]  0.54224879 -1.78712109  0.08791697 -0.12603800  1.81663871  1.08807544
 [79]  0.95347826  1.26766140 -0.36662189 -0.50058462 -1.85956645 -0.08105427
 [85] -0.25105474  0.61234485 -0.71690014  0.91970570  0.01942227  1.34305636
 [91]  0.85899910  1.24778179  1.91462308 -0.21422645  0.26717435  1.02025121
 [97] -0.54744300  0.16574293  0.62589084 -0.76815763
```

We can combine objects, variables and functions to do more complex stuff in R, here's how we get the mean of 100 random numbers.

```r
numbers <- rnorm(100)
mean(numbers)
```

```
[1] -0.1671
```

Here we created a vector object with `rnorm(100)` and assigned it to the variable `numbers`. We than used the `mean()` function, passing it the variable `numbers`. The `mean()` function returned the mean of the hundred random numbers.

Dataframes

One of the more common objects that R uses is a dataframe. The dataframe is a rectangular table-like object that contains data, think of it like a spreadsheet tab. Like the spreadsheet, the dataframe has rows and columns, the columns have names and the different columns can have different types of data in. Here's a little one

```
  names age    score
1 Guido  24 13.96877
2 Marty  45 57.40436
3  Alan  11 92.06278
```

Usually we get a dataframe by loading in data from an external source or as a result from functions, occasionally we'll want to hand make one, which can be done with various functions, `data.frame` being the most common.

```r
data.frame(
  names = c("Guido", "Marty", "Alan"),
  age = c(24,45,11),
  score = runif(3) * 100
```

```
)
```

Packages

Many of the tools we use in will come in R packages, little nuggets of code that group related functions together. Installing new packages can be done using the `Packages` pane of RStudio or the `install.packages()` function. When we wish to use that code we use the `library()` function

```
library(somepackage)
```

Using R Help

R provides a command, called `?` that will display the documentation for functions. For example `?mean` will display the help for the `mean()` function.

```
?mean
```

As in all programming languages the internal documentation in R is written with some assumption that the reader is familiar with the language. This can be a pain when you are starting out as the help will seem a bit obscure at times. Don't worry about this, usually the `Examples` section will give you a good idea of how to use the function and as your experience grows then the more things will make more sense.

> **Roundup**
>
> `* R is an excellent and powerful statistical computing environment`

> **For you to do**
>
> Complete the interactive tutorial online https://danmaclean.shinyapps.io/r-start

Blachnik, Marcin, Marek Sołtysiak, and Dominika Dąbrowska. 2019. "Predicting Presence of Amphibian Species Using Features Obtained from GIS and Satellite Images." *ISPRS International Journal of Geo-Information* 8 (3). https://doi.org/10.3390/ijgi8030123.