Genomalicious tutorial 3: Population structure

Joshua A. Thia

25 Februrary 2024

Preamble

Once you have your genetic data in hand, it is time to start analysing! An essential part of any population genetic study is an analysis of population structure. Population structure describes how genetic variation is partitioned among populations. In the simplest sense, a population is a group of individuals that interbreed freely with each other at the exclusion of other individuals outside this group. As a result, discrete populations are genetically differentiated; we want to measure and characterise this genetic differentiation.

In this tutorial, you will:

1. Use *genomalicious* to calculate F_{ST}.
2. Use *genomalicious* to perform a principal components a analysis (PCA).
3. Use *genomalicious* to perform a discriminant analysis of principal components (DAPC).
4. Use *genomalicious* to visualise patterns of population structure.

Load *genomalicious*

```r
library(genomalicious)
```

Calculating F_{ST}

The classic measure of genetic differentiation among populations is the statistic, F_{ST}. There are many different formulations of this statistic. The original was derived by Seawell Wright (Wright 1951) and was referred to as a fixation index: the probability that populations are fixed for different alleles. Regardless of the underlying calculation, as F_{ST} increases, populations are increasingly genetically differentiated. In its originally formulation as a probability, F_{ST} is bounded between 0 and 1, but some formulations may also produce negative values (<0), which are basically equivalent to $F_{\mathrm{ST}} = 0$.

In *genomalicious*, F_{ST} is calculated using the Weir and Cockerham formulation, which you may see in the literature as represented as θ_{WC} (Weir & Cockerham 1984). This formulation is effectively the ratio of among population variance in allele frequencies relative to the total variance in allele frequencies. In this case, F_{ST} is bounded between -1 and 1.

Let us prepare our data for analysis. We will start with a dataset of 4 populations genotyped at 200 SNP loci. Not all of these SNPs occur in different genomic regions, RADseq "contigs" in this case. We will therefore need to filter our SNPs to obtain just one SNP per contig before proceeding further.

```
# Load in the genotype data
data(data_Genos)

# Number of samples per population
data_Genos[, .(NUM.SAMPLES=length(unique(SAMPLE))), by=POP]
```

```
##     POP NUM.SAMPLES
## 1: Pop1          25
## 2: Pop2          25
## 3: Pop3          25
## 4: Pop4          25
```

```
# Number of SNP loci
data_Genos[, length(unique(LOCUS))]
```

```
## [1] 200
```

```
# Number of loci per contig
data_Genos[, .(NUM.LOCI=length(unique(LOCUS))), by=CHROM] %>%
  .[NUM.LOCI>1]
```

```
##         CHROM NUM.LOCI
## 1: Contig2074        2
## 2: Contig2914        2
## 3: Contig3113        2
## 4: Contig3658        2
## 5: Contig4953        2
```

```
# Filter for "unlinked" loci
loci.unlink <- filter_unlink(data_Genos)

genoData <- data_Genos[LOCUS %in% loci.unlink]
```

Now that we have our filtered SNP loci, we can calcualte F_{ST}. We can calculate a **global** F_{ST}, which is a measure of genetic differentiation across all 4 populations, or a **pairwise** F_{ST} between each pair of populations. The *genomalicious* function for calculating F_{ST} is `fstat_calc`. This function takes a long-format `data.table` object with information on individually sequenced genotypes or population allele frequencies at biallelic SNP loci.

There are quite a few arguments for `fstat_calc` that enable this function to be very flexible with respect to the outputs it can produce and the data structure of the data table it can receive. We will walk through these argument before executing the function.

The first argument, `dat`, takes our data table of genotypes of allele frequencies. The argument `type` is specified with either `"genos"` of `"freqs"` to tell the function that we want F_{ST} to be estimated from genotype or allele frequency data, respectively.

We need to tell the function whether we want a global or pairwise F_{ST} using the `global` and `pairwise` arguments in tandem. These are logical arguments and they must be directly contrasting. If you want a global F_{ST}, you parameterisation must be `global=TRUE, pairwise=FALSE`; conversely, pairwise F_{ST} requires `global=FALSE, pairwise=TRUE`.

For genotype data we can also test the significance of our F_{ST} estimate relative to null expectations using a permutation test. To activate this permutation test, we use the argument `permute` which expects a logical value of `TRUE` to perform the permutation test, or `FALSE` if you do not want to perform the permutation test. The permutation test requires randomly shuffling individuals among populations for some desired number of iterations. The number of iterations is specified with `numPerms`. You can specify the number of cores to use for permutations using the argument `numCores`.

Now, we need to make sure that we specify the columns in the data table that contain the relevant information on samples, populations, loci, and genotypes or allele frequencies. These are `sampCol`, `popCol`, `locusCol`, `genoCol`, `indsCol`, and `freqCol`, respectively. Note that if you are working with genotypes, you do not need to specify `indsCol` and `freqCol`. If you are working with allele frequencies, you do not need to specify `sampCol` and `genoCol`.

The `fstat_calc` function can actually calculates a suite of F-statistics from genotype data. Aside from F_{ST}, you can also calculate F_{IS} and F_{IT}, the inbreeding coefficients within populations and in the total sample, respectively, following Weir & Cockerham (1984). You can specify which of these you want with the `fstat` argument. By default, this argument is `NULL`, so you must specify at least one of `FST`, `FIS`, or `FIT` in a character vector. If working with allele frequencies, leave this argument as `NULL`. For this tutorial, we will just focus on F_{ST} and calculate it for both genotype and allele frequency data.

OK, after all that, let us run `fstat_calc` and estimate a global F_{ST}. The function call will produce a list with three indexes: `$genome`, which contains a single value, the genome-wide F_{ST} across all loci; `$locus`, which contains a data table of per locus F_{ST} and their respective numerator and denominator variance components; and `$permute`, a list itself with `$fst` with the permuted F_{ST} estimates, and `$pval` for the permuted p-value. The `$permute` index will not be present if permutations are not requested.

```
# Global FST with 100 permutations
fstGenoGlobal <- fstat_calc(
  dat=genoData, type='genos', fstat=c('FST'), popCol='POP', sampCol='SAMPLE',
  locusCol='LOCUS', genoCol='GT', global=TRUE, permute=TRUE, numPerms=100
)
```

```
## F-statistic calculation on genotype data, global estimate
## Performing permutations
```

```
# The genome-wide estimate of the global FST.
fstGenoGlobal$genome
```

```
##          FST
## 1: 0.1166258
```

```
# The per locus estimate of global FST
fstGenoGlobal$locus
```

```
##                LOCUS        FST
##   1:    Contig1_437 0.19383044
##   2:   Contig34_213 0.16117960
##   3:   Contig38_427 0.02777778
##   4:  Contig152_389 0.09945465
##   5:   Contig170_36 0.17529210
##  ---
## 191: Contig7274_197 0.24547441
## 192:  Contig7287_10 0.13183891
## 193:  Contig7291_92 0.16754504
## 194: Contig7293_124 0.01827372
## 195: Contig7299_279 0.14750710
```

```r
# Take a look at the indexes within the `permute` index.
fstGenoGlobal$permute %>% names()
```

```
## [1] "fstat" "pval"
```

```r
# Permuted FST
fstGenoGlobal$permute$fst
```

```
##      PERM           FST
##   1:    1 -4.076935e-04
##   2:    2 -2.074337e-03
##   3:    3 -1.791915e-03
##   4:    4 -7.445567e-04
##   5:    5  1.548666e-03
##   6:    6 -4.259010e-04
##   7:    7  1.184833e-03
##   8:    8  1.103253e-05
##   9:    9  6.564271e-05
##  10:   10  4.674985e-03
##  11:   11  4.660732e-04
##  12:   12  8.300334e-04
##  13:   13 -4.499293e-03
##  14:   14 -6.717167e-04
##  15:   15  2.840690e-04
##  16:   16  3.848426e-03
##  17:   17 -2.548165e-03
##  18:   18 -4.167972e-04
##  19:   19 -4.298625e-03
##  20:   20  6.844570e-04
##  21:   21 -9.448797e-04
##  22:   22 -8.173992e-04
##  23:   23  2.931696e-04
##  24:   24  3.659729e-04
##  25:   25 -2.274789e-03
##  26:   26 -9.357737e-04
##  27:   27  1.212123e-03
##  28:   28  3.113706e-04
##  29:   29  8.391316e-04
##  30:   30 -4.714202e-04
##  31:   31 -8.356102e-04
##  32:   32  3.085154e-03
##  33:   33 -8.265046e-04
##  34:   34  5.437669e-03
##  35:   35 -1.646164e-03
##  36:   36  8.573279e-04
##  37:   37  6.227293e-03
##  38:   38 -3.441443e-03
##  39:   39 -1.518640e-03
##  40:   40 -1.318261e-03
##  41:   41 -1.346017e-04
##  42:   42  4.193623e-03
##  43:   43  1.066573e-03
##  44:   44  2.757951e-03
##  45:   45 -2.165450e-03
##  46:   46 -1.382016e-03
##  47:   47 -1.955897e-03
##  48:   48 -3.450560e-03
##  49:   49 -9.539858e-04
##  50:   50  2.021618e-04
##  51:   51  7.026546e-04
##  52:   52 -2.493487e-03
##  53:   53 -9.904104e-04
##  54:   54 -1.482206e-03
##  55:   55  1.020487e-04
##  56:   56 -4.987322e-04
##  57:   57  3.285086e-03
##  58:   58 -1.363800e-03
##  59:   59  3.230561e-03
##  60:   60 -3.614675e-03
##  61:   61  1.439523e-03
##  62:   62 -2.684866e-03
##  63:   63  8.573279e-04
##  64:   64  3.750731e-04
##  65:   65  5.283339e-03
##  66:   66 -9.813042e-04
##  67:   67 -7.172414e-04
##  68:   68  4.811200e-03
##  69:   69  5.654112e-05
##  70:   70 -1.090581e-03
##  71:   71  1.184833e-03
##  72:   72 -6.178329e-05
##  73:   73  5.934666e-04
##  74:   74 -1.208972e-03
##  75:   75 -9.813042e-04
##  76:   76 -9.630919e-04
##  77:   77  1.657804e-03
##  78:   78  4.933724e-04
##  79:   79  3.021535e-03
##  80:   80  1.212123e-03
##  81:   81 -3.569086e-03
##  82:   82  9.028178e-04
##  83:   83 -3.423209e-03
##  84:   84  5.115717e-04
##  85:   85  3.185122e-03
##  86:   86 -2.511713e-03
##  87:   87  3.204711e-04
##  88:   88 -1.372908e-03
##  89:   89  2.903381e-03
##  90:   90  1.657576e-04
##  91:   91 -9.539858e-04
##  92:   92 -3.447705e-05
##  93:   93 -1.682601e-03
##  94:   94  9.028178e-04
##  95:   95 -1.573292e-03
##  96:   96  3.295715e-04
##  97:   97  2.294646e-04
##  98:   98 -2.739549e-03
##  99:   99  4.229956e-03
## 100:  100  1.202514e-04
##      PERM           FST
```

```
# Permuted p-value
fstGenoGlobal$permute$pval
```

```
##    STAT PVAL
## 1:  FST    0
```

And here is an example with pairwise F_{ST}. We will use a smaller number of permutations so that the function call runs faster.

```
# Pairwise FST with 30 permutations.
fstGenoPairs <- fstat_calc(
  dat=genoData, type='genos', fstat=c('FST'), popCol='POP', sampCol='SAMPLE',
  locusCol='LOCUS', genoCol='GT', global=FALSE, pairwise=TRUE,
  permute=TRUE, numPerms=30
)
```

```
## F-statistic calculation on genotype data, pairwise estimates
## Estimates for pair: 1 / 6
## Estimates for pair: 2 / 6
## Estimates for pair: 3 / 6
## Estimates for pair: 4 / 6
## Estimates for pair: 5 / 6
## Estimates for pair: 6 / 6
```

```
# The genome-wide estimate of pairwise FST
fstGenoPairs$genome
```

```
##    POP1 POP2       FST
## 1: Pop1 Pop2 0.1246213
## 2: Pop1 Pop3 0.1155931
## 3: Pop1 Pop4 0.1220902
## 4: Pop2 Pop3 0.1063992
## 5: Pop2 Pop4 0.1208514
## 6: Pop3 Pop4 0.1096819
```

```
# The per locus estiamte of pairwise FST
fstGenoPairs$locus
```

```
##       POP1 POP2          LOCUS          FST
##    1: Pop1 Pop2    Contig1_437  0.243221131
##    2: Pop1 Pop2   Contig34_213  0.345601538
##    3: Pop1 Pop2   Contig38_427 -0.009192646
##    4: Pop1 Pop2  Contig152_389  0.079928664
##    5: Pop1 Pop2   Contig170_36 -0.021528998
##   ---
## 1166: Pop3 Pop4 Contig7274_197  0.546001927
## 1167: Pop3 Pop4  Contig7287_10 -0.013127854
## 1168: Pop3 Pop4  Contig7291_92  0.030172414
## 1169: Pop3 Pop4 Contig7293_124 -0.003983635
## 1170: Pop3 Pop4 Contig7299_279  0.074427481
```

```r
# Permuted FST
fstGenoPairs$permute$fst
```

```
##      POP1 POP2 PERM           FST
##   1: Pop1 Pop2    1  0.0006837157
##   2: Pop1 Pop2    2  0.0013364560
##   3: Pop1 Pop2    3  0.0018257373
##   4: Pop1 Pop2    4  0.0024234292
##   5: Pop1 Pop2    5 -0.0012497554
##  ---
## 176: Pop3 Pop4   26 -0.0050867890
## 177: Pop3 Pop4   27  0.0020755782
## 178: Pop3 Pop4   28  0.0013644436
## 179: Pop3 Pop4   29  0.0067000510
## 180: Pop3 Pop4   30 -0.0006863977
```

```r
# Permuted p-value
fstGenoPairs$permute$pval
```

```
##    POP1 POP2 STAT PVAL
## 1: Pop1 Pop2  FST    0
## 2: Pop1 Pop3  FST    0
## 3: Pop1 Pop4  FST    0
## 4: Pop2 Pop3  FST    0
## 5: Pop2 Pop4  FST    0
## 6: Pop3 Pop4  FST    0
```

For pool-seq projects, you will not have individually sequenced genotypes, but instead, allele frequencies per populations. The parameterisation is basically the same. We will take a look using a pool-seq dataset comprising 4 populations and 200 RADseq SNP loci.

```r
# The dataset of allele frequencies
data("data_PoolFreqs")
head(data_PoolFreqs)
```

```
##        CHROM POS         LOCUS ALT REF  POP       FREQ  DP AO  RO POOL
## 1:   Contig1 437   Contig1_437   A   T Pop1 0.24193548  62 15  47    1
## 2:  Contig34 213  Contig34_213   G   A Pop1 0.15853659  82 13  69    1
## 3:  Contig38 427  Contig38_427   A   G Pop1 0.16296296 135 22 113    1
## 4: Contig152 389 Contig152_389   T   A Pop1 0.03809524 105  4 101    1
## 5: Contig170  36  Contig170_36   G   A Pop1 0.69879518  83 58  25    1
## 6: Contig183 250 Contig183_250   C   A Pop1 0.93023256  43 40   3    1
```

```r
# The metadata of pooled samples
data('data_PoolInfo')
data_PoolInfo
```

```
##    POOL INDS
## 1:    1   30
## 2:    2   30
## 3:    3   30
## 4:    4   30
```

```r
# You need to add in the information of number of pooled individuals into the
# data table of allele frequencies. Let's do this with `left_join`.
data_PoolFreqs <- left_join(data_PoolFreqs, data_PoolInfo, by=c('POOL'))
head(data_PoolFreqs)
```

```
##        CHROM POS         LOCUS ALT REF  POP       FREQ  DP AO  RO POOL INDS
## 1:   Contig1 437   Contig1_437   A   T Pop1 0.24193548  62 15  47    1   30
## 2:  Contig34 213  Contig34_213   G   A Pop1 0.15853659  82 13  69    1   30
## 3:  Contig38 427  Contig38_427   A   G Pop1 0.16296296 135 22 113    1   30
## 4: Contig152 389 Contig152_389   T   A Pop1 0.03809524 105  4 101    1   30
## 5: Contig170  36  Contig170_36   G   A Pop1 0.69879518  83 58  25    1   30
## 6: Contig183 250 Contig183_250   C   A Pop1 0.93023256  43 40   3    1   30
```

```r
# There are some RAD contigs with more than one SNP locus
data_PoolFreqs[, .(NUM.LOCI=length(unique(LOCUS))), by=CHROM] %>%
  .[NUM.LOCI>1]
```

```
##         CHROM NUM.LOCI
## 1: Contig2074        2
## 2: Contig2914        2
## 3: Contig3113        2
## 4: Contig3658        2
## 5: Contig4953        2
```

```r
# Get the "unlinked" loci.
loci.unlink.freq <- filter_unlink(dat=data_PoolFreqs)

# Subset for one SNP per RAD contig.
freqData <- data_PoolFreqs[LOCUS %in% loci.unlink.freq]

# Calculate pairwise FST with 30 iterations.
fstFreqsPairs <- fstat_calc(
  dat=freqData, type='freqs', fstat=NULL, popCol='POOL',
  locusCol='LOCUS', freqCol='FREQ', indsCol='INDS',
  global=FALSE, pairwise=TRUE, numPerms=30
)
```

```
## F-statistic calculation on frequency data, global estimate
```

```r
# The genome-wide pairwise FST.
fstFreqsPairs$genome
```

```
##    POP1 POP2       FST
## 1: Pop1 Pop2 0.1222501
## 2: Pop1 Pop3 0.1207414
## 3: Pop1 Pop4 0.1184308
## 4: Pop2 Pop3 0.0935783
## 5: Pop2 Pop4 0.1171423
## 6: Pop3 Pop4 0.1047949
```

```r
# The per locus pairiwse FST.
fstFreqsPairs$locus
```

```
##       POP1 POP2          LOCUS         FST
##    1: Pop1 Pop2    Contig1_437  0.34352035
##    2: Pop1 Pop2   Contig34_213  0.39573054
##    3: Pop1 Pop2   Contig38_427 -0.01572640
##    4: Pop1 Pop2  Contig152_389  0.09308798
##    5: Pop1 Pop2   Contig170_36  0.01598986
##   ---
## 1166: Pop3 Pop4 Contig7274_197  0.60614422
## 1167: Pop3 Pop4  Contig7287_10 -0.03389232
## 1168: Pop3 Pop4  Contig7291_92 -0.03305589
## 1169: Pop3 Pop4 Contig7293_124 -0.03034464
## 1170: Pop3 Pop4 Contig7299_279  0.16584577
```

Principal components analysis (PCA)

PCA is a multivariate statistical technique for summarising (co)variation among a set of measured variables. Datasets with many measured variables often suffer from collinearity, that is, when different variables covary (are correlated). The goal of PCA is to reduce the dimensionality across these measured variables. PCA creates new "synthetic" axis from combinations of the original variables, based on their covariance (or correlation).

The number of PC axes created equal to the number of measured variables, or the sample size, whichever is smallest. These PC axes are constrained to be orthogonal (at right angles) and describe decreasingly less variation. The first PC axis will describe the greatest amount of variation, the second PC axes will describe the second most variation, etc. The new set of PC axes generated is often referred to as the **PC space**.

PCA provides a useful way for summarising genetic differences among individuals and populations. We expect that across multiple loci, individuals from the sample population have more similar genotypes than individuals from different populations. These genotypic covariances across multiple loci can be summarised. Individuals can be plotted in the new PC space to visualise genetic relationships. From a population genomics perspective, PCA is very useful because the number of loci can range from hundreds, to thousands, to millions.

In *genomalicious*, we use the `pca_genos` function for performing a PCA on genotype data. Our main data input is `dat`, a long-format `data.table` object of genotypes for multiple individuals and loci. We use the `scaling` argument to define how we want genotypes to be scaled. Use `"covar"` for covariance, `"corr"` for correlation, or `"patterson"` for the scaling described in Patterson et al. (2006). The sample, locus, and genotype columns need to be specified with `sampCol`, `locusCol`, and `genoCol`, respectively. The population column, `popCol`, is an optional argument; you should specify this column if you want to do downstream analyses with *genomalicious*.

The function `pca_genos` is in fact a wrapper for R's internal function, `prcomp`. You will be returned a `prcomp` object from `pca_genos`. If you have specified the `popCol`, then the output will have an additional `$pops` index with the population designation.

```r
# Fit the PCA
PCA <- pca_genos(
  dat=genoData, scaling='covar',
  sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT', popCol='POP'
)
```

```r
# Take a look at the class
class(PCA)
```

```
## [1] "prcomp"
```

```r
# See the indexes in this object
names(PCA)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"        "pops"
```

```r
# Note the $pops index. Here are the populations of the first 40 individuals.
PCA$pops %>% head(., 40)
```

```
##  [1] "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1"
## [11] "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop1"
## [21] "Pop1" "Pop1" "Pop1" "Pop1" "Pop1" "Pop2" "Pop2" "Pop2" "Pop2" "Pop2"
## [31] "Pop2" "Pop2" "Pop2" "Pop2" "Pop2" "Pop2" "Pop2" "Pop2" "Pop2" "Pop2"
```

```r
# The combinations of loci contributing to each PC axes are stored
# in the $rotation index. Here are the first 3 PC axes and the
# first 8 loci.
PCA$rotation[1:8, 1:3]
```

```
##                         PC1          PC2           PC3
## Contig1014_62   0.02457683  0.003777642 -0.1521021169
## Contig1047_30  -0.05487043 -0.054785237  0.1153941504
## Contig1074_118  0.01584607  0.135445469  0.0111219021
## Contig1078_331  0.01461413 -0.040778239 -0.0006191517
## Contig1109_489  0.01770112 -0.047703448 -0.0031718832
## Contig1132_452  0.02906339 -0.012224774 -0.1453458010
## Contig1219_313  0.06860691  0.152075910  0.1481717705
## Contig1323_488 -0.11498077  0.041542671 -0.2240768171
```

```r
# The scores for individuals in the new PC space are stored in
# the $x index. Here are the first 3 PC axes and first 8 individuals.
PCA$x[1:8,1:3]
```

```
##                PC1      PC2         PC3
## Ind1_1  -2.697820 2.081174 -0.14044399
## Ind1_10 -2.980405 1.128620  0.63009655
## Ind1_11 -3.134540 1.256936 -0.78987718
## Ind1_12 -2.755598 2.117070  0.26755118
## Ind1_13 -4.546516 1.633624  1.64320869
## Ind1_14 -2.779081 1.565420  0.03160392
## Ind1_15 -2.762871 0.487206  0.28571537
## Ind1_16 -3.582775 1.943842  1.62717998
```

```r
# You can convert the PCA result into a long-format data table
# with the genomalicious function, `pca2DT`.
PCA.tab <- pca2DT(PCA, subAxes=1:5)

head(PCA.tab)
```

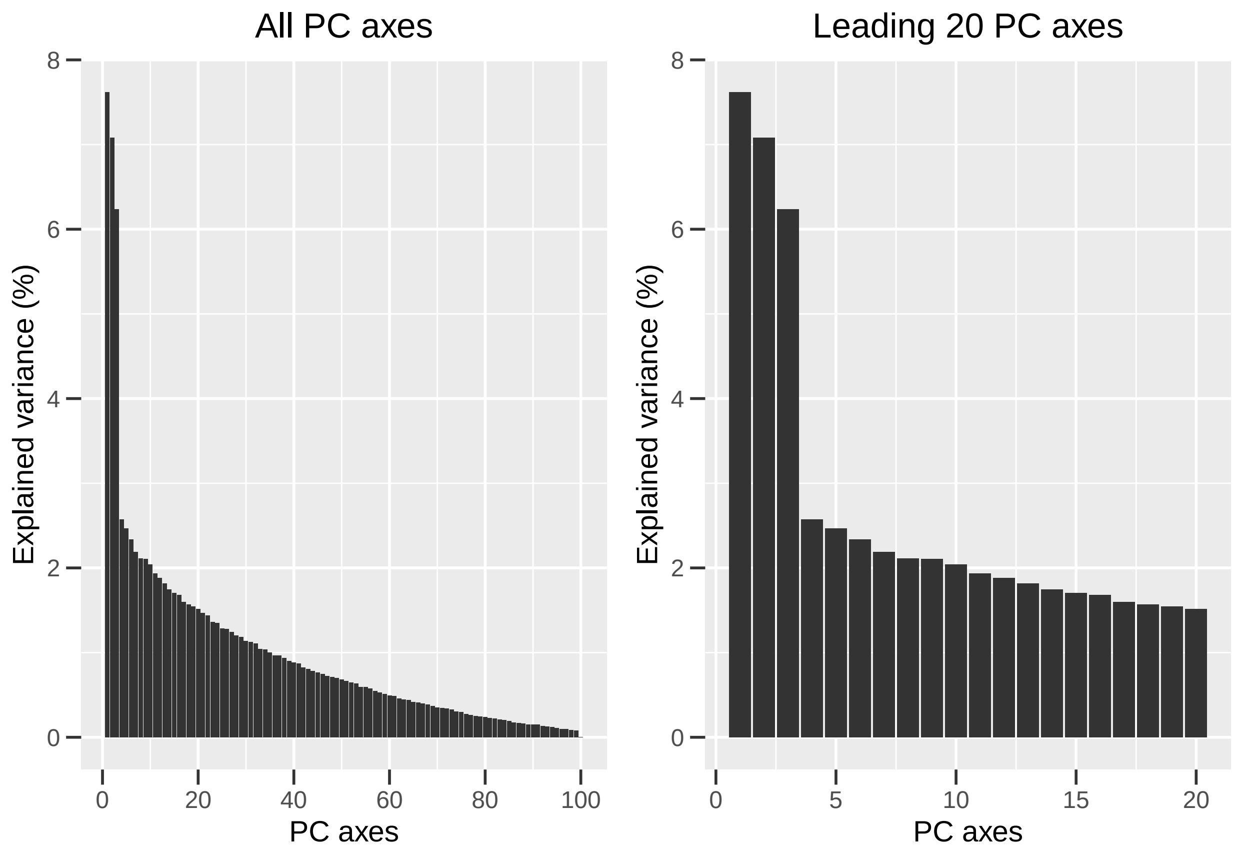
```
##     SAMPLE       PC1      PC2         PC3        PC4        PC5  POP
## 1:  Ind1_1 -2.697820 2.081174 -0.14044399 -0.7598846  0.2616145 Pop1
## 2: Ind1_10 -2.980405 1.128620  0.63009655  1.8208115  0.5346243 Pop1
## 3: Ind1_11 -3.134540 1.256936 -0.78987718  2.0467099 -1.7396232 Pop1
## 4: Ind1_12 -2.755598 2.117070  0.26755118 -0.5469413  2.1247598 Pop1
## 5: Ind1_13 -4.546516 1.633624  1.64320869  1.9161048 -0.5305748 Pop1
## 6: Ind1_14 -2.779081 1.565420  0.03160392  0.7022996 -0.5418015 Pop1
```

OK, so we have performed a PCA and summarised the genotypic covariances and we have scores for individuals projected into the new PC space. We can use the *genomalicious* function `pca_plot` to visualise our PCA results. This function takes in a `prcomp` object as its input using the `pcaObj` argument.

Additional arguments for `pca_plot` allow us to customise the output. There are three different types of plots that can be obtained using the `type` argument: `"scatter"` returns a scatterplot of individuals projected into PC space; `"scree"` returns a screeplot of explained variances for each PC axis; and `"cumvar"` returns a barplot of cumulative variance for each PC axis. The `axisIndex` argument takes an integer vector, the PC axes to plot. The `pops` argument is optional, and is only used if you want to colour samples by population in the scatterplot. The function will search for a `$pops` index in the `pcaObj` input, but you can manually specify this if you prefer. The `plotColours` argument is a vector of colours for manually specifying the colour palette of your plot. The `look` argument is used to specify whether you want your plot to have a ggplot2 stype, `"ggplot"`, or a classic R style, `"classic"`.

Let us first start by looking at how our PC axes describe the variaiton in our genotype dataset. We will first make a screeplot, which illustrates how much variation is captured by each PC axes.

```r
# Two screeplots: one using the default settings and another zooming in on the
# leading 20 PC axes.
plot.pca.scree.1 <- pca_plot(PCA, type='scree')
plot.pca.scree.2 <- pca_plot(PCA, type='scree', axisIndex=1:20)

ggarrange(
  plot.pca.scree.1 +
    ggtitle('All PC axes') +
    theme(plot.title=element_text(hjust=0.5)),
  plot.pca.scree.2 +
    ggtitle('Leading 20 PC axes') +
    theme(plot.title=element_text(hjust=0.5))
  )
```



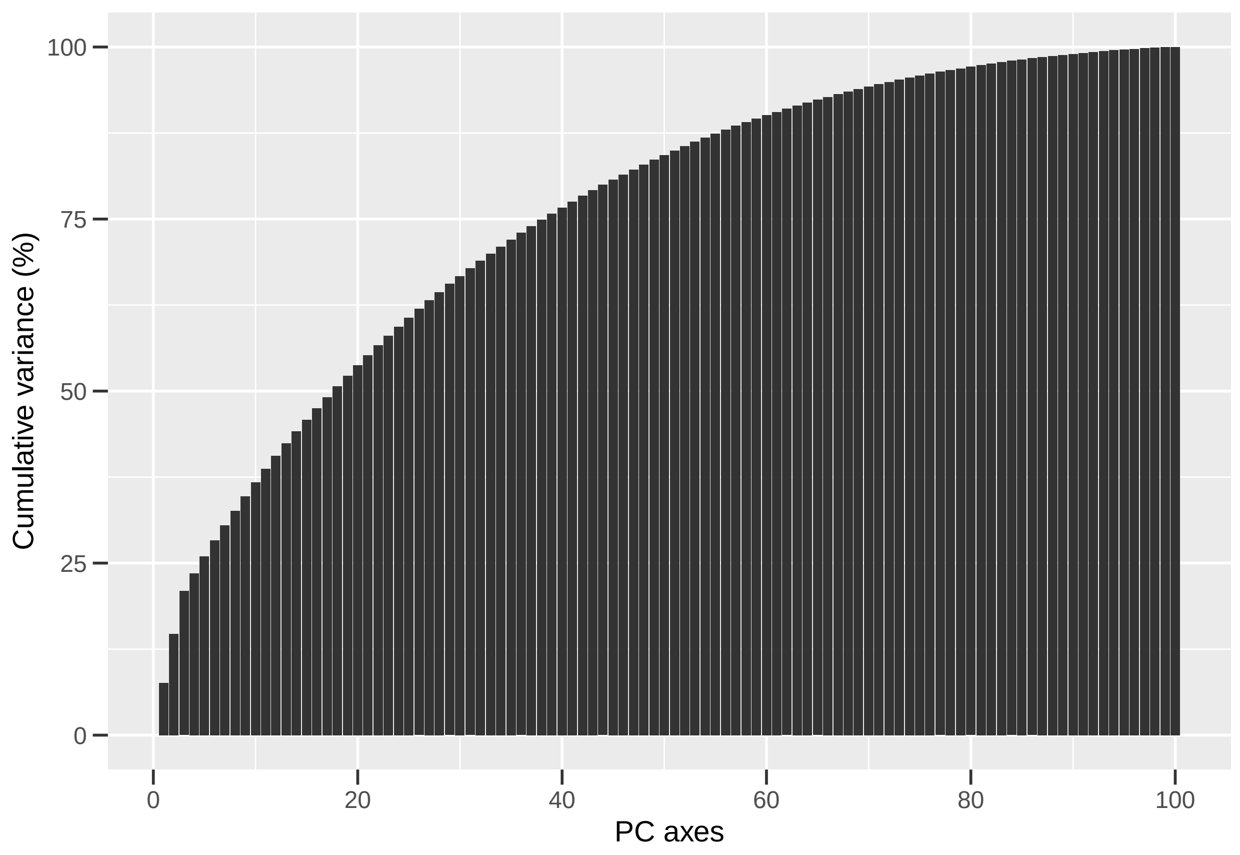
The screeplot places the PC axes on the x-axis and the percent of total explained variance on the y-axis. You can see that the explained variance decreases with each successive PC axis. However, this decline is not necessarily even. If we zoom in on the leading 20 PC axes, we can see that the first 3 PC axes describe the most variation, then there is quite a large jump in explained varaince from PC axes 3 to 4. From PC axis 4 and beyond, there is a more gradual decline in explained variance.

The break in the pattern of explained variance arises from a unique property of PCA to capture the major axes of population structure for k populations on the first $k - 1$ PC axes. Our dataset comprises $k = 4$ populations, therefore, all important genetic covariances describing the population structure among these populations is summarised by PC axes 1, 2, and 3. Beyond PC axis 3, the PC axes describe noise and random covariances in our genotype dataset. Whereas the leading $k - 1$ PC axes are **biologically informative**, the PC axes $>= k$ are **biologically uninformative**. (for more reading, see Patterson et al. 2006 *PLOS Genetics*, and Thia 2022 *Mol. Ecol.*)

Understanding which PC axes describe biologically informative variation is important for our interpretation of PCA results. Typically, you will at least want to visualise the PC1 and 2, but if other PC axes are important, you may also want to look at those too. Knowledge of the biologically informative PC axes is also essential for modelling populations structure, but we will get to that a bit later in this lesson.

Another way to visualise the explained variance is through a barplot of the cumulative variance. This explains the total explained variance with each successive PC axis:

```r
# A barplot of cumulative variance using default settings
plot.pca.cumvar <- pca_plot(PCA, type='cumvar')
plot.pca.cumvar
```

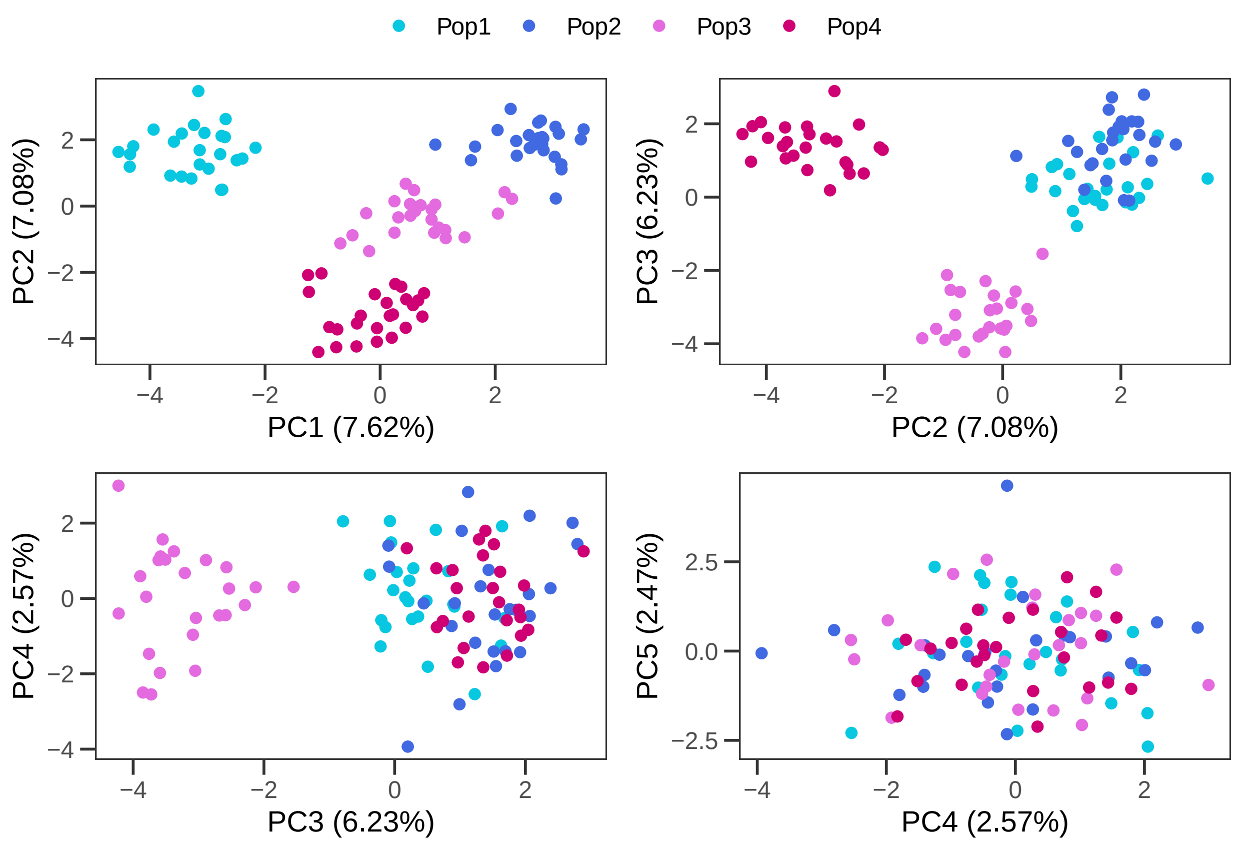


Now that we understand how variance is captured on our PC axes, let us now visualise the projections of samples into PC space using scatterplots. We will look at projections from PC axes 1 to 5, but remember, only the leading 3 PC axes describe biologically informative variation.

```
# Create an empty list to hold the plots
scatterPlotList <- list()

# Create a list of axis combinations
axis.combos <- list(1:2, 2:3, 3:4, 4:5)

# Iterate through each ith axis combination, make a plot, and store in the list.
# These plots illustrate the subsetting of axes, custom colours for populations,
# and specifying a classic plot look.
for(i in 1:length(axis.combos)){
  scatterPlotList[[i]] <- pca_plot(
    PCA, type='scatter', axisIndex=axis.combos[[i]],
    plotColours=c(Pop1='#08c7e0', Pop2='#4169e1', Pop3='#e46adf', Pop4='#ce0073'),
    look='classic'
  )
}

# Check out the length of the list.
length(scatterPlotList)
```

```
## [1] 4
```

```
# Combine into a single figure.
ggarrange(plotlist=scatterPlotList, common.legend=TRUE)
```



Our scatterplots highlight the unique combinations genotypic covariances described by our PCA on PC axes 1, 2, and 3. We can see that on each of these leading 3 PC axes, there is separation of populations into discrete regions of PC space; we often describe this separation as "clustering". However, once we get to PC axis 4, this clustering disappears. On PC axes 4 and 5, all samples collapse into a single "cloud" of points. This illustrates the lack of population structure on PC axis 4 and beyond.

Discriminant analysis of principal components (DAPC)

DAPC has become a popular method for studying population structure following its introduction by Jombart et al.'s (2010). This method begins with a PCA to reduce dimensionality in a genotype dataset. Some number of leading PC axes are then chosen as **predictors** of population structure. These are then used to fit a model of among population differences using discriminatory analysis (DA). DA then produces a set of linear discriminant (LD) axes that best separate populations using linear combinations of scores on the predictor PC axes. The new multidimensional space that describes the maximal among population differences can be described as the **LD space**.

It is important to keep in mind that populations are not always clearly defined units. Where population structure exists, there is a value of k that describes the effective number of populations. This effective number of biological populations may or may not be the same as our pre-conceived expectations of what the populations are. It is therefore important to clearly communicate how you parameterise a DAPC to ensure that biological patterns are not confounded by statistical artefacts and misinterpretations of the results (Miller et al. 2020; Thia 2022)

There are two important parameters in a DA. The first is k_{DA}, the number of populations a researcher wants to discriminate. The second is p_{axes}, the number of PC axes to use as predictors of the among-population differences.

The choice of k_{DA} depends on whether a researcher is interested in building a model that describes differences among a set of *a priori* defined populations, or a set of *de novo* defined populations. It is important to decide this in advance **before** analysing your data because you must allocate indiviudals to these defined populations before fitting your DA model. *A priori* defined populations might constitute sampling sites that you want to test for population structure using a DA, this may or may not be equivalent to k. *De novo* defined populations are inferred from clustering in the data and a DA is used to model the differences among these groups, and typically this is chosen to match the inferred k.

The choice of p_{axes} is dependent on the number of biologically informative PC axes. As described above, only a limited number of PC axes describe population structure in a genotype dataset. For k effective populations, no more than $k - 1$ leading PC axes should be used as predictors of among-population differences in a DA.

Thia (2022 *Mol. Ecol*) provides recommendations on best practise guidelines for parameterising and interpreting DAPC on genotype datasets. We will go over these recommendations in this lesson using *genomalicious*.

Inferring the *k* effective populations

As we saw earlier, an examination of the PCA screeplot of explained variances and the scatterplot of projections can help us visually assess the number of biologically informative PC axes. Aside from these visual examinations, we can also perform statistical tests to infer k. One of the most widely implemented methods (in the context of DAPC) is K-means clustering (Jombart et al. 2010 *BMC Genetics*).

K-means clustering involves dividing samples into some specified number of groups based on a set of measured variables. We can use our PC axes of genotypic variation as predictors of different numbers of potential groups (populations) in our dataset. We can then calculate a test statistic for these different fits to identify the most likely k. Note, that K-means is a modelling approach, so parameterisation matters; this was highlighted in Thia (2022). It is therefore important to try different parameterisations to see: (1) whether different parameterisation converge on the same solution; and (2) whether these parameterisations produce consistent results to those in our visual inspection of PCA results.

In *genomalicious*, the `dapc_infer` function can be used to help summarise PCA results and different parameterisations of K-means clustering on a genotype dataset. As usual, the input is a long-format genotype `data.table` object. We specify the scaling for PCA using the `scaling` argument, and the arguments `sampCol`, `locusCol`, and `genoCol` to specify where to find the sample, locus, and genotype information, respectively.

The parameterisations for the K-means clustering are set with `kTest`, an integer vector of values of k to test, and `pTest`, an integer vector of the number of PC axes to use as predictors. The argument `screeMax` is a single integer, specifying the number of PC axes to show in the returned screeplot. And the `look` argument is used to control whether a `"ggplot"` or `"classic"` stype plot should be returned.

The function returns a `list` object, with the indexes:

1. `$tab`, a table of K-means test statistics for each fit, the BIC score (lower the more likely).
2. `$fit` a list of `kmeans` objects, one for each parameterisation.
3. `$plot` a `gg` object, a plot summarising the PCA and K-means results.

```
# Perform a PCA using a covariance scaling. Fit K-means with k ranging from 1
# to 10. Use 4, 10, 20 and 40 PC axes as predictors. Display only the first
# 20 PC axes for the screeplot.
DAPC.infer <- dapc_infer(
  genoData, scaling='covar', sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT',
  kTest=1:10, pTest=c(4,10,20,40), screeMax=20
)

# Take a look at the indexes
names(DAPC.infer)
```

```
## [1] "tab"  "fit"  "plot"
```

```
# The table of BIC test statistics in the $tab index, the first 10 rows
DAPC.infer$tab %>% head(., 10)
```

```
##     K  P      BIC
##  1: 1  4 276.9879
##  2: 1 10 321.7058
##  3: 1 20 359.7095
##  4: 1 40 395.2312
##  5: 2  4 246.8140
##  6: 2 10 305.5008
##  7: 2 20 350.5286
##  8: 2 40 390.3383
##  9: 3  4 203.0723
## 10: 3 10 286.0405
```

```
# Take a look at the names in the $fit index
DAPC.infer$fit %>% names()
```

```
##  [1] "k=1,p=4"   "k=1,p=10"  "k=1,p=20"  "k=1,p=40"  "k=2,p=4"   "k=2,p=10"
##  [7] "k=2,p=20"  "k=2,p=40"  "k=3,p=4"   "k=3,p=10"  "k=3,p=20"  "k=3,p=40"
## [13] "k=4,p=4"   "k=4,p=10"  "k=4,p=20"  "k=4,p=40"  "k=5,p=4"   "k=5,p=10"
## [19] "k=5,p=20"  "k=5,p=40"  "k=6,p=4"   "k=6,p=10"  "k=6,p=20"  "k=6,p=40"
## [25] "k=7,p=4"   "k=7,p=10"  "k=7,p=20"  "k=7,p=40"  "k=8,p=4"   "k=8,p=10"
## [31] "k=8,p=20"  "k=8,p=40"  "k=9,p=4"   "k=9,p=10"  "k=9,p=20"  "k=9,p=40"
## [37] "k=10,p=4"  "k=10,p=10" "k=10,p=20" "k=10,p=40"
```

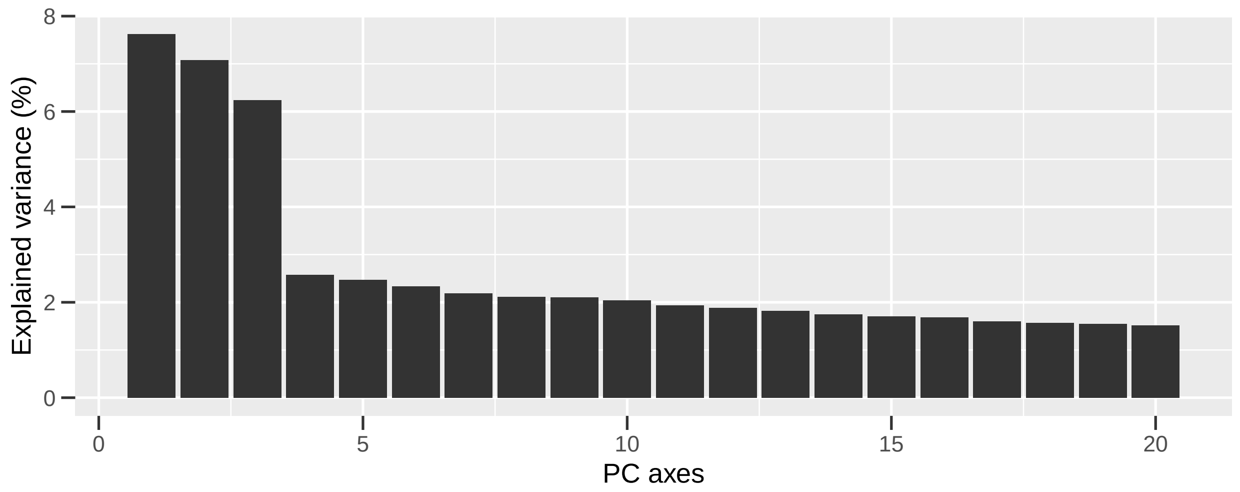
```
# The indexes within $fit are themselves kmeans objects.
DAPC.infer$fit$`k=1,p=10` %>% class
```

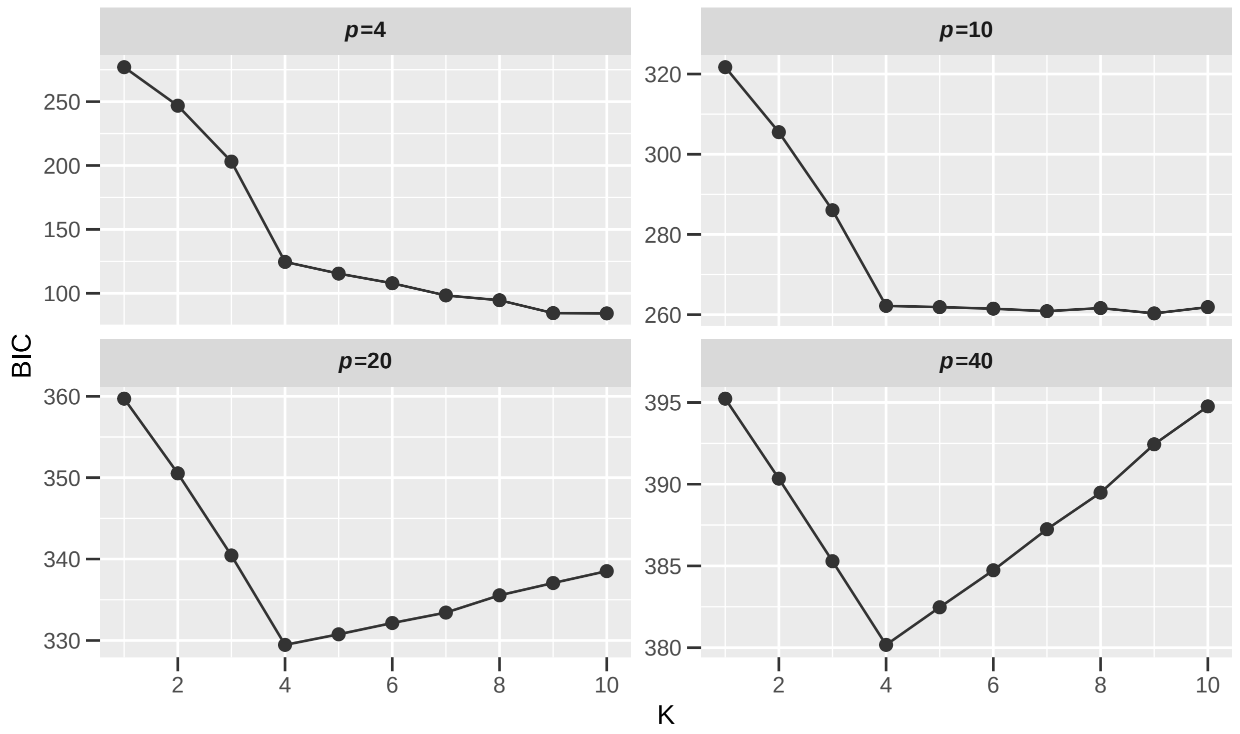
```
## [1] "kmeans"
```

```
# Note, you can find the assigned clusters for the K-means fits.
# Here are the assigned clusters for k=1 and p=10.
DAPC.infer$fit$`k=1,p=10`$cluster
```

```
##  Ind1_1 Ind1_10 Ind1_11 Ind1_12 Ind1_13 Ind1_14 Ind1_15 Ind1_16 Ind1_17 Ind1_18
##       1       1       1       1       1       1       1       1       1       1
## Ind1_19  Ind1_2 Ind1_20 Ind1_21 Ind1_22 Ind1_23 Ind1_24 Ind1_25  Ind1_3  Ind1_4
##       1       1       1       1       1       1       1       1       1       1
##  Ind1_5  Ind1_6  Ind1_7  Ind1_8  Ind1_9  Ind2_1 Ind2_10 Ind2_11 Ind2_12 Ind2_13
##       1       1       1       1       1       1       1       1       1       1
## Ind2_14 Ind2_15 Ind2_16 Ind2_17 Ind2_18 Ind2_19  Ind2_2 Ind2_20 Ind2_21 Ind2_22
##       1       1       1       1       1       1       1       1       1       1
## Ind2_23 Ind2_24 Ind2_25  Ind2_3  Ind2_4  Ind2_5  Ind2_6  Ind2_7  Ind2_8  Ind2_9
##       1       1       1       1       1       1       1       1       1       1
##  Ind3_1 Ind3_10 Ind3_11 Ind3_12 Ind3_13 Ind3_14 Ind3_15 Ind3_16 Ind3_17 Ind3_18
##       1       1       1       1       1       1       1       1       1       1
## Ind3_19  Ind3_2 Ind3_20 Ind3_21 Ind3_22 Ind3_23 Ind3_24 Ind3_25  Ind3_3  Ind3_4
##       1       1       1       1       1       1       1       1       1       1
##  Ind3_5  Ind3_6  Ind3_7  Ind3_8  Ind3_9  Ind4_1 Ind4_10 Ind4_11 Ind4_12 Ind4_13
##       1       1       1       1       1       1       1       1       1       1
## Ind4_14 Ind4_15 Ind4_16 Ind4_17 Ind4_18 Ind4_19  Ind4_2 Ind4_20 Ind4_21 Ind4_22
##       1       1       1       1       1       1       1       1       1       1
## Ind4_23 Ind4_24 Ind4_25  Ind4_3  Ind4_4  Ind4_5  Ind4_6  Ind4_7  Ind4_8  Ind4_9
##       1       1       1       1       1       1       1       1       1       1
```

```r
# Take a look at the visual summary in the $plot index. Illustrates the
# screeplot and the estimated BIC for each parameterisation set.
DAPC.infer$plot
```





Let us consider the returned summary plot. As expected, there is a clear break in the screeplot at PC axis 3 (the $k - 1$ PC axis), which we expect for 3 populations in this simulated dataset. The K-means clustering are all consistent with $k = 4$ populations. However you can clearly see that different parameterisations produce quite different outcomes with respect to the magnitude of the test statistic and the test statistic curves. Nonetheless, everything is pointing to using $k - 1 = 3$ PC axes to model differences among $k = 4$ populations.

Fitting a DAPC

Alright, we now know how we want to parameterise our DAPC, so we can turn to the function `dapc_fit` to fit our model of among-population differences. This function has a lot of similar arguments to those we have already seen. We need to provide our data as a long-format `data.table` object. We specify the scaling method for PCA with the `scaling` argument. We have the arguments `sampCol`, `locusCol`, and `genoCol` to indicate the columns with the sample, locus, and genotype information.

Importantly, this time we also have the `popCol` argument, which specifies the column with population information. This column represents the populations that we will be discriminating among in our DAPC. They could be *a priori* definitions of populations, or *de novo* inferred definitions based on some other clustering methods prior to performing DAPC.

The `dapc_fit` function has a 3 different methods that can be run using the argument `method`. The default value is `"fit"`, which simply runs the DAPC fit on the entire dataset. We will focus on the `method=="fit"` parameterisation of the `dapc_fit` function first before discussing alternate settings.

When `method=="fit"`, a DA of PC axes predictors is fit using R's internal `lda` function from the *MASS* package. The `dapc_fit` function returns a `list` object with multiple indexes.

1. `$da.fit` is an `lda` class object, the DA of among-population differences using PC axes as predictors.
2. `$da.tab` is a `data.table` object of LD scores for each sample.
3. `da.prob` is a `data.table` object of posterior probabilities of populaton identity for each sample.
4. `$pca.fit` is a `prcomp` object of the PCA fit.
5. `$pca.tab` is a `data.table` object of PC scores for each sample.
6. `$snp.contrib` is a `data.table` object of SNP contributions (proportion of variation) to each LD axis (values for each axis sums to 1).

```r
# Fit the DAPC with a PCA of genotype covariances, using 3 PC axes as
# predictors of among-population differences.
DAPC.fit <- dapc_fit(
  genoData, scaling='covar', sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT',
  popCol='POP', pcPreds=3, method='fit'
)

# The output is a list
class(DAPC.fit)
```

```
## [1] "list"
```

```r
# Take a look at the classes of all the indexes
lapply(DAPC.fit, class)
```

```
## $da.fit
## [1] "lda"
##
## $da.tab
## [1] "data.table" "data.frame"
##
## $da.prob
## [1] "data.table" "data.frame"
##
## $pca.fit
## [1] "prcomp"
##
## $pca.tab
## [1] "data.table" "data.frame"
##
## $snp.contrib
## [1] "data.table" "data.frame"
```

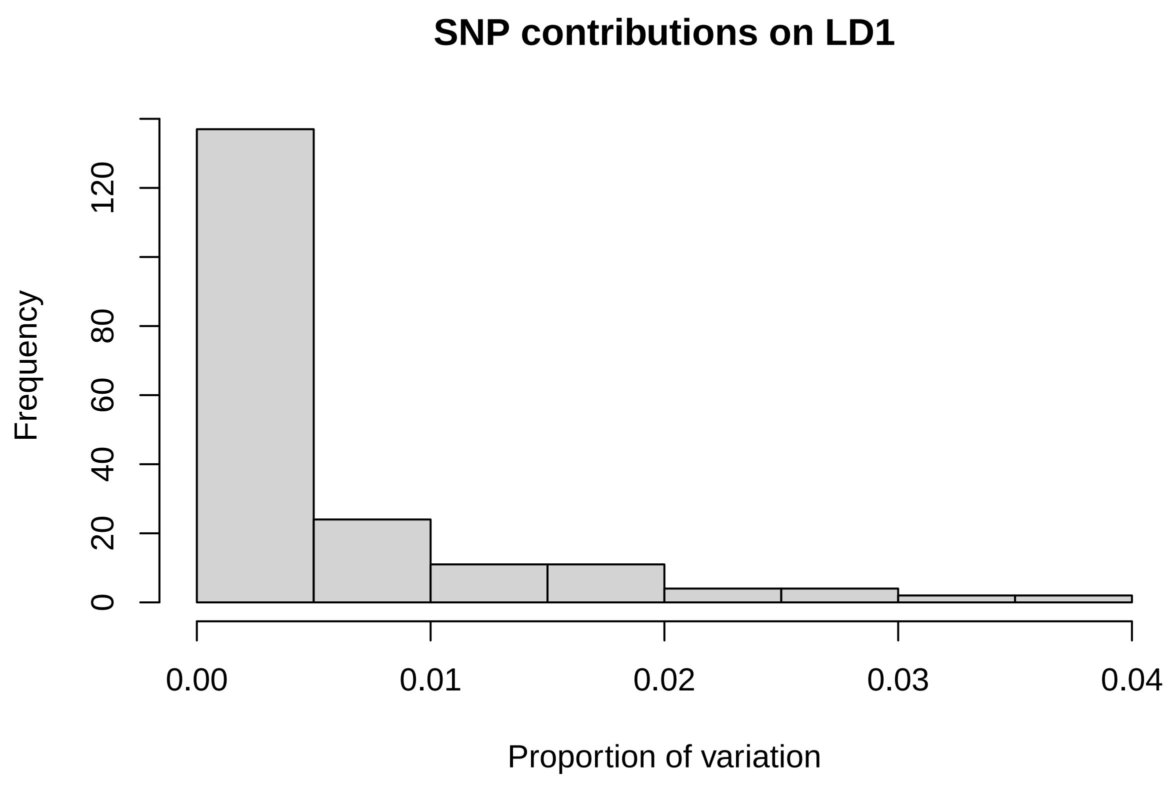
```r
# The LD scores
DAPC.fit$da.tab
```

```
##       POP  SAMPLE         LD1        LD2         LD3
##   1: Pop1  Ind1_1  5.30233792 -1.6389072 -0.18663421
##   2: Pop1 Ind1_10  4.03714256 -3.1513319  0.31302427
##   3: Pop1 Ind1_11  4.70533317 -2.3241965 -1.56006966
##   4: Pop1 Ind1_12  5.30133577 -1.9462542  0.34887052
##   5: Pop1 Ind1_13  5.86620572 -5.3114032  1.34010391
##   6: Pop1 Ind1_14  4.61520152 -2.2137307 -0.21563514
##   7: Pop1 Ind1_15  3.04096206 -3.1256873 -0.35211621
##   8: Pop1 Ind1_16  5.45106925 -3.9337962  1.78634781
##   9: Pop1 Ind1_17  6.12861684 -2.4116017 -0.47927555
##  10: Pop1 Ind1_18  4.13021301 -1.9483200 -0.31130788
##  11: Pop1 Ind1_19  5.45065809 -2.8545523  1.55269334
##  12: Pop1  Ind1_2  6.05841415 -4.4113920  0.53852807
##  13: Pop1 Ind1_20  6.68093350 -3.0287580 -0.35276765
##  14: Pop1 Ind1_21  4.04563823 -1.9739084  0.11488610
##  15: Pop1 Ind1_22  3.84125343 -3.8429046  0.33032827
##  16: Pop1 Ind1_23  4.23686713 -3.5778049 -0.57092511
##  17: Pop1 Ind1_24  5.59311909 -2.4138992  2.46472620
##  18: Pop1 Ind1_25  5.16084634 -2.3913712 -0.61062279
##  19: Pop1  Ind1_3  6.02282539 -4.0095928 -0.88959928
##  20: Pop1  Ind1_4  4.29948378 -1.4600691  0.31435656
##  21: Pop1  Ind1_5  7.47303044 -1.6197941  1.12666333
##  22: Pop1  Ind1_6  5.58119402 -4.0847083 -1.46419246
##  23: Pop1  Ind1_7  4.26668548 -4.2645781  0.34683849
##  24: Pop1  Ind1_8  2.98584357 -3.2273712 -0.07585311
##  25: Pop1  Ind1_9  6.15882991 -2.3402031  0.45211548
##  26: Pop2  Ind2_1 -0.49553929  3.7260498  3.45471728
##  27: Pop2 Ind2_10 -1.33606316  3.8433723  3.26484283
##  28: Pop2 Ind2_11  0.16114748  2.9677950  4.21724133
##  29: Pop2 Ind2_12 -2.65191994  3.0690543  2.63189690
##  30: Pop2 Ind2_13 -0.12746645  3.6250362  4.32635609
##  31: Pop2 Ind2_14  1.71083752  3.8525617  3.97529863
##  32: Pop2 Ind2_15 -0.22077852  3.9109894  4.76089351
##  33: Pop2 Ind2_16 -0.79466500  4.2447599  4.81713616
##  34: Pop2 Ind2_17  0.71690307  4.6592647  1.70809171
##  35: Pop2 Ind2_18  0.14938917  4.1655754  3.24273685
##  36: Pop2 Ind2_19  0.03002706  4.0523065  2.25385057
##  37: Pop2  Ind2_2 -0.32740616  3.5403659  4.11904936
##  38: Pop2 Ind2_20 -0.32915924  4.7544899  4.47652135
##  39: Pop2 Ind2_21  0.44163358  4.7924579  1.74138667
##  40: Pop2 Ind2_22  0.44340985  1.6958820  4.51247647
##  41: Pop2 Ind2_23 -0.21126067  3.3116973  2.70342562
##  42: Pop2 Ind2_24 -0.82842774  4.1026803  2.84891934
##  43: Pop2 Ind2_25 -0.46848662  2.7364380  5.33304666
##  44: Pop2  Ind2_3 -1.62121778  3.5465694  3.59579517
##  45: Pop2  Ind2_4  1.34339277  1.4573914  3.19983850
##  46: Pop2  Ind2_5  0.47736952  2.7312835  1.41864672
##  47: Pop2  Ind2_6  0.82959891  4.4184274  3.36907701
##  48: Pop2  Ind2_7  0.74186962  4.1771531  4.10169965
##  49: Pop2  Ind2_8  0.87230536  2.7281628  4.43063381
##  50: Pop2  Ind2_9 -0.06412201  3.5153034  5.79952606
##  51: Pop3  Ind3_1 -2.07113726  2.4524386 -2.72471829
##  52: Pop3 Ind3_10  0.17720146  2.5441569 -4.96507443
##  53: Pop3 Ind3_11 -0.05743796  0.7014533 -5.48299389
##  54: Pop3 Ind3_12 -0.29505804  1.9031171 -2.97736769
##  55: Pop3 Ind3_13 -0.39460526  2.1595662 -5.23730133
##  56: Pop3 Ind3_14 -0.16752196  2.9623505 -3.75609624
##  57: Pop3 Ind3_15 -1.07439200  4.5495717 -2.52026249
##  58: Pop3 Ind3_16  0.98874380  3.2290858 -4.04918049
##  59: Pop3 Ind3_17  0.30750239  3.0230141 -4.53976290
##  60: Pop3 Ind3_18 -0.07242864  2.3541117 -3.40539327
##  61: Pop3 Ind3_19 -0.74374378  3.4818209 -5.52055538
##  62: Pop3  Ind3_2  0.31543110  3.1932630 -4.52298833
##  63: Pop3 Ind3_20 -1.23239840  4.5730214 -4.09130522
##  64: Pop3 Ind3_21  0.50350538  2.9349545 -4.43249585
##  65: Pop3 Ind3_22  0.70558850  2.2698076 -3.66578752
##  66: Pop3 Ind3_23 -0.40265811  3.2344241 -4.89706837
##  67: Pop3 Ind3_24 -1.35503033  2.5054565 -3.35121054
##  68: Pop3 Ind3_25 -1.14209886  2.6250304 -4.27748670
##  69: Pop3  Ind3_3  0.26972232  3.9064765 -5.23727075
##  70: Pop3  Ind3_4 -0.16793658  0.4415949 -3.90466645
##  71: Pop3  Ind3_5 -0.76002960  1.2971071 -5.75863368
##  72: Pop3  Ind3_6  0.92938746  2.0094418 -1.59330073
##  73: Pop3  Ind3_7 -0.56523816  4.8458112 -3.10733918
##  74: Pop3  Ind3_8  0.68274147  1.5564762 -4.25849567
##  75: Pop3  Ind3_9 -1.37638002  3.1883053 -5.18311133
##  76: Pop4  Ind4_1 -4.19175736 -2.5739846  1.66464534
##  77: Pop4 Ind4_10 -4.47548259 -3.2278986 -0.60662190
##  78: Pop4 Ind4_11 -6.13937229 -4.3003533  0.86857083
##  79: Pop4 Ind4_12 -4.54361723 -1.5349013  0.26589238
##  80: Pop4 Ind4_13 -5.22370798 -3.4003615  1.13388712
##  81: Pop4 Ind4_14 -2.66324835 -3.7291739 -0.72811058
##  82: Pop4 Ind4_15 -5.53841577 -3.9153489  0.86153199
##  83: Pop4 Ind4_16 -5.75116793 -2.7653009 -0.07923194
##  84: Pop4 Ind4_17 -5.99706441 -4.7518292  0.53748681
##  85: Pop4 Ind4_18 -5.05014234 -2.4785204  0.98285038
##  86: Pop4 Ind4_19 -5.61134550 -2.3713068  0.56025563
##  87: Pop4  Ind4_2 -5.16182105 -3.1702925  0.89341181
##  88: Pop4 Ind4_20 -6.09331792 -3.6287667  0.44240151
##  89: Pop4 Ind4_21 -3.65609289 -1.7768691 -0.09798054
##  90: Pop4 Ind4_22 -4.85251756 -4.4299009 -0.06820831
##  91: Pop4 Ind4_23 -4.68290413 -2.4437285  0.91570541
##  92: Pop4 Ind4_24 -4.66394262 -4.6190248  0.06442750
##  93: Pop4 Ind4_25 -2.12237178 -3.8468476  0.44076855
##  94: Pop4  Ind4_3 -5.59355982 -5.5211451 -0.04853794
##  95: Pop4  Ind4_4 -5.48032429 -4.5622706 -0.87072822
##  96: Pop4  Ind4_5 -2.24110405 -3.4921122  0.45639257
##  97: Pop4  Ind4_6 -3.84692019 -2.6176220  0.04497293
##  98: Pop4  Ind4_7 -4.20176832 -2.0632453 -1.01046273
##  99: Pop4  Ind4_8 -5.25407581 -3.1108878  2.78503093
## 100: Pop4  Ind4_9 -4.84109850 -3.7290964 -0.21207037
##       POP  SAMPLE         LD1        LD2         LD3
```

```
# The population posterior probabilities
DAPC.fit$da.prob
```

```
##        POP  SAMPLE POP.PRED PROB
##   1: Pop1  Ind1_1     Pop1    1
##   2: Pop1 Ind1_10     Pop1    1
##   3: Pop1 Ind1_11     Pop1    1
##   4: Pop1 Ind1_12     Pop1    1
##   5: Pop1 Ind1_13     Pop1    1
##  ---
## 396: Pop4  Ind4_5     Pop4    1
## 397: Pop4  Ind4_6     Pop4    1
## 398: Pop4  Ind4_7     Pop4    1
## 399: Pop4  Ind4_8     Pop4    1
## 400: Pop4  Ind4_9     Pop4    1
```

```
# SNP contributions to LD1
DAPC.fit$snp.contrib[['LD1']] %>%
  hist(., main='SNP contributions on LD1', xlab='Proportion of variation')
```



```
# Sums for each axis column
DAPC.fit$snp.contrib[['LD1']] %>% sum
```

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

```
DAPC.fit$snp.contrib[['LD2']] %>% sum
```

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

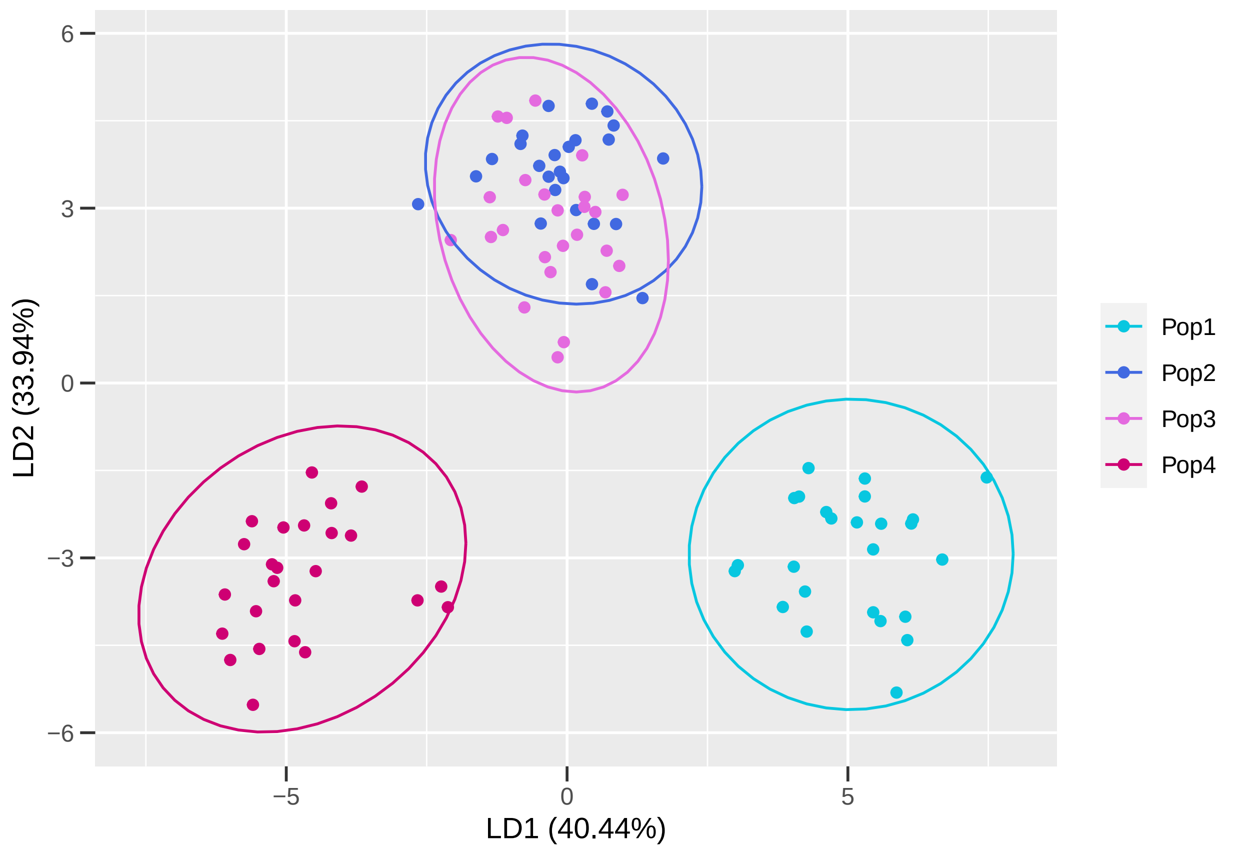
```
DAPC.fit$snp.contrib[['LD2']] %>% sum
```

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

The output from `dapc_fit(..., method=="fit")` can be passed directly to the function `dapc_plot` to visualise the results, for which we use the `type` argument to generate either a scatterplot with `"scatter"`, or a barplot of posterior probabilities for population identities using `"probs"`.

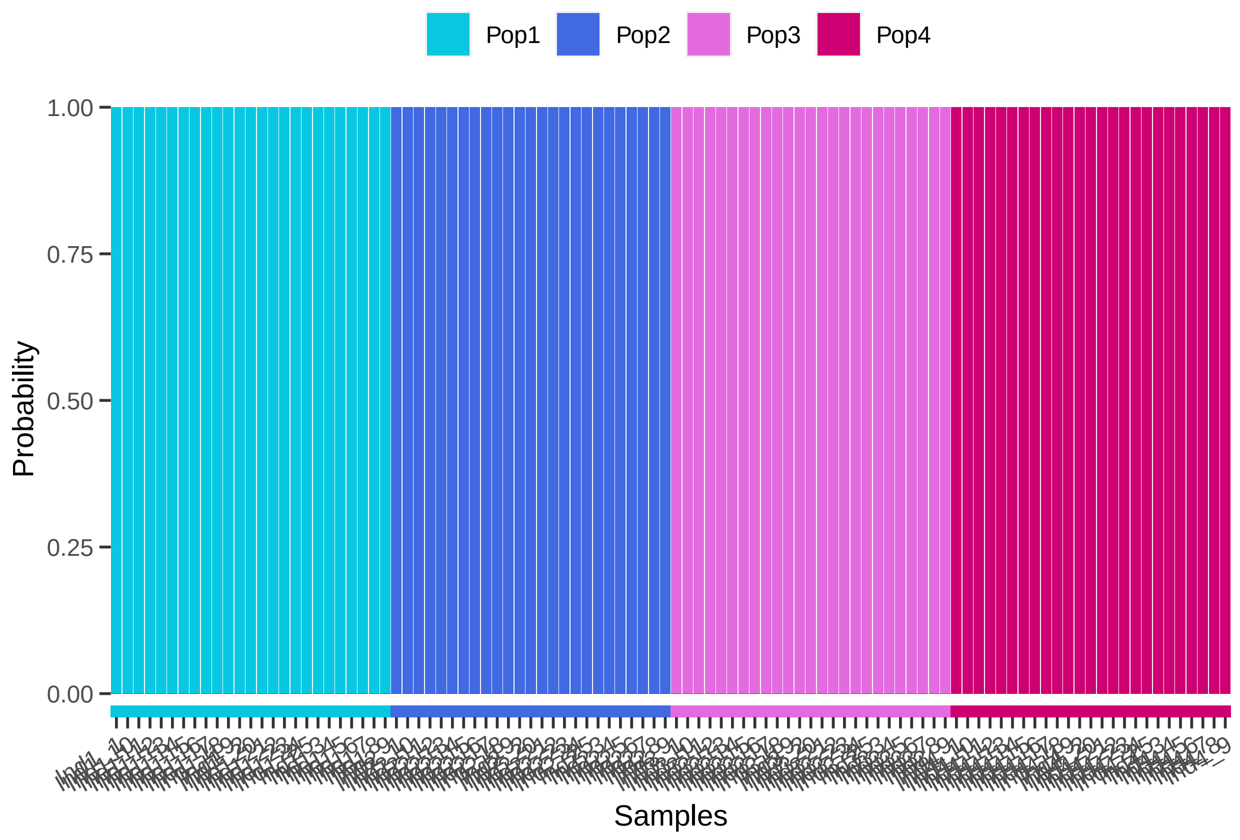
If we set `dapc_plot(..., type="scatter")`, we can control the LD axes to display with the argument `axisIndex`, which takes an integer vector of 2 values. The argument `plotColours` takes a named character vector of colours for each population, `legendPos` is used to specify the positioning of the legend.

```
plot.dapc.scatter <- dapc_plot(
  DAPC.fit, type='scatter', axisIndex=c(1,2),
  plotColours=c(Pop1='#08c7e0', Pop2='#4169e1', Pop3='#e46adf', Pop4='#ce0073'),
  legendPos='right'
)
plot.dapc.scatter
```



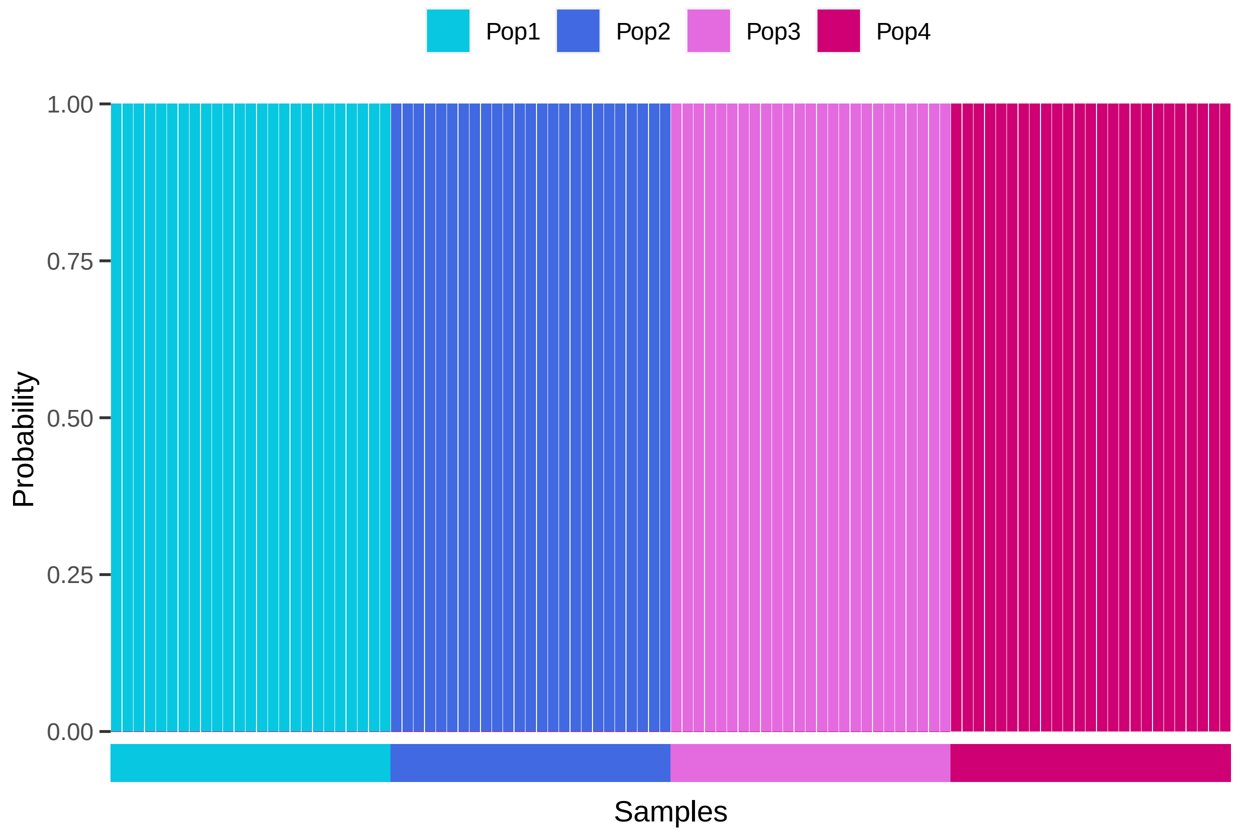
If we set `dapc_plot(..., type="probs")`, there are a number of arguments to manipulate the barplot. By default, samples are ordered by their designated population, with barplots illustrating the relative composition of posterior probabilities. Underneath the samples, a horizontal bar is used to highlight the designated populations, and sample names are plotted. Let us take a look at this default output:

```
# The default posterior probability barplot, only specifying custom colours.
plot.dapc.probs.default <- dapc_plot(
  DAPC.fit, type='probs',
  plotColours=c(Pop1='#08c7e0', Pop2='#4169e1', Pop3='#e46adf', Pop4='#ce0073')
)
plot.dapc.probs.default
```



That is a bit messy, is it not? This time, let us hide the sample names and increase the bar that delimits the original population designations. We can hide the samples using the `sampleShow` argument, and setting it to `FALSE`. We can increase the size of the population designation bar using the argument `popBarScale`, which takes a numeric value as the relative scaling size.

```
# The customised posterior probability barplot
plot.dapc.probs.custom <- dapc_plot(
  DAPC.fit, type='probs', sampleShow=FALSE, popBarScale=2,
  plotColours=c(Pop1='#08c7e0', Pop2='#4169e1', Pop3='#e46adf', Pop4='#ce0073')
)
plot.dapc.probs.custom
```



Assessing model fit

Because DAPC constructs a model of population structure, it is important to assess the fit of this model. Assessing model fit requires using the model to predict the population identity of samples that were not used to predict the model. If our model predicts population identities that are congruent with the original population designation, then our model is well fit.

There are two general approaches we can take. A **leave-one-out cross-validation** can be used when samples sizes are small, whereby samples are omitted one-by-one, the model is refit multiple times and tested on each omitted sample. However, if you can, it is better to use a **training-testing** partitioning approach, whereby the data is divided into a training set to build the model, which is then used to predict the testing set.

Again, we can use the `dapc_fit` function to assess model fit, except this time we change the `type` argument to either `"loo_cv"` to perform leave-one-out cross-validation, or `"train_test"` to perform training-testing partitioning. All the arguments are the same as if we fitting a DAPC to the entire dataset (as above). However, if `dapc_fit(..., type="train_test")`, we must also specify the argument `trainProp`, which is a numeric value between 0 and 1 that represents the proportion of the original dataset to hold as the training set. The training set will comprise `trainProp` proportion of samples from each population, so that each population is represented equally.

```
DAPC.loo.cv <- dapc_fit(
  genoData, scaling='covar', sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT',
  popCol='POP', pcPreds=3, method='loo_cv',
)
```

```
DAPC.train.test <- dapc_fit(
  genoData, scaling='covar', sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT',
  popCol='POP', pcPreds=3, method='train_test', trainProp=0.8
)
```

The outputs of these function calls is a `list` object with the following indexes:

1. `$tab` is a `data.table` of predicted populations for each sample.
2. `$global` is a single numeric value, which represents the global assignment rate of samples correctly assigned to their original designated population across the entire dataset.
3. `$pairs.long` is a `data.table` of pairwise population assignment rates in long-format.
4. `$pairs.wide` is a `data.table` of pairwise population assignment rates in wide-format.

```
# List names
names(DAPC.train.test)
```

```
## [1] "tab"        "global"     "pairs.long" "pairs.wide"
```

```
# Table of predictions for training-testing
DAPC.train.test$tab
```

```
##      POP  SAMPLE POP.PRED
##  1: Pop1 Ind1_10     Pop1
##  2: Pop1 Ind1_11     Pop1
##  3: Pop1 Ind1_12     Pop1
##  4: Pop1 Ind1_14     Pop1
##  5: Pop1 Ind1_16     Pop1
##  6: Pop2 Ind2_12     Pop2
##  7: Pop2 Ind2_16     Pop2
##  8: Pop2 Ind2_17     Pop2
##  9: Pop2  Ind2_4     Pop1
## 10: Pop2  Ind2_6     Pop2
## 11: Pop3 Ind3_10     Pop3
## 12: Pop3 Ind3_14     Pop3
## 13: Pop3  Ind3_2     Pop3
## 14: Pop3 Ind3_22     Pop1
## 15: Pop3  Ind3_5     Pop3
## 16: Pop4 Ind4_11     Pop4
## 17: Pop4 Ind4_16     Pop4
## 18: Pop4 Ind4_21     Pop4
## 19: Pop4 Ind4_25     Pop1
## 20: Pop4  Ind4_4     Pop4
```

```
# Global assignment rate for training-testing
DAPC.train.test$global
```

```
## [1] 0.85
```

```
# Compare to the global assignment rate for leave-one-out cross-validation
DAPC.loo.cv$global
```

```
## [1] 0.86
```

```
# The two pairiwse assignment rate data tables
DAPC.train.test$pairs.long
```

```
##      POP POP.PRED ASSIGN
##  1: Pop1     Pop1    1.0
##  2: Pop1     Pop2    0.0
##  3: Pop1     Pop3    0.0
##  4: Pop1     Pop4    0.0
##  5: Pop2     Pop1    0.2
##  6: Pop2     Pop2    0.8
##  7: Pop2     Pop3    0.0
##  8: Pop2     Pop4    0.0
##  9: Pop3     Pop1    0.2
## 10: Pop3     Pop2    0.0
## 11: Pop3     Pop3    0.8
## 12: Pop3     Pop4    0.0
## 13: Pop4     Pop1    0.2
## 14: Pop4     Pop2    0.0
## 15: Pop4     Pop3    0.0
## 16: Pop4     Pop4    0.8
```

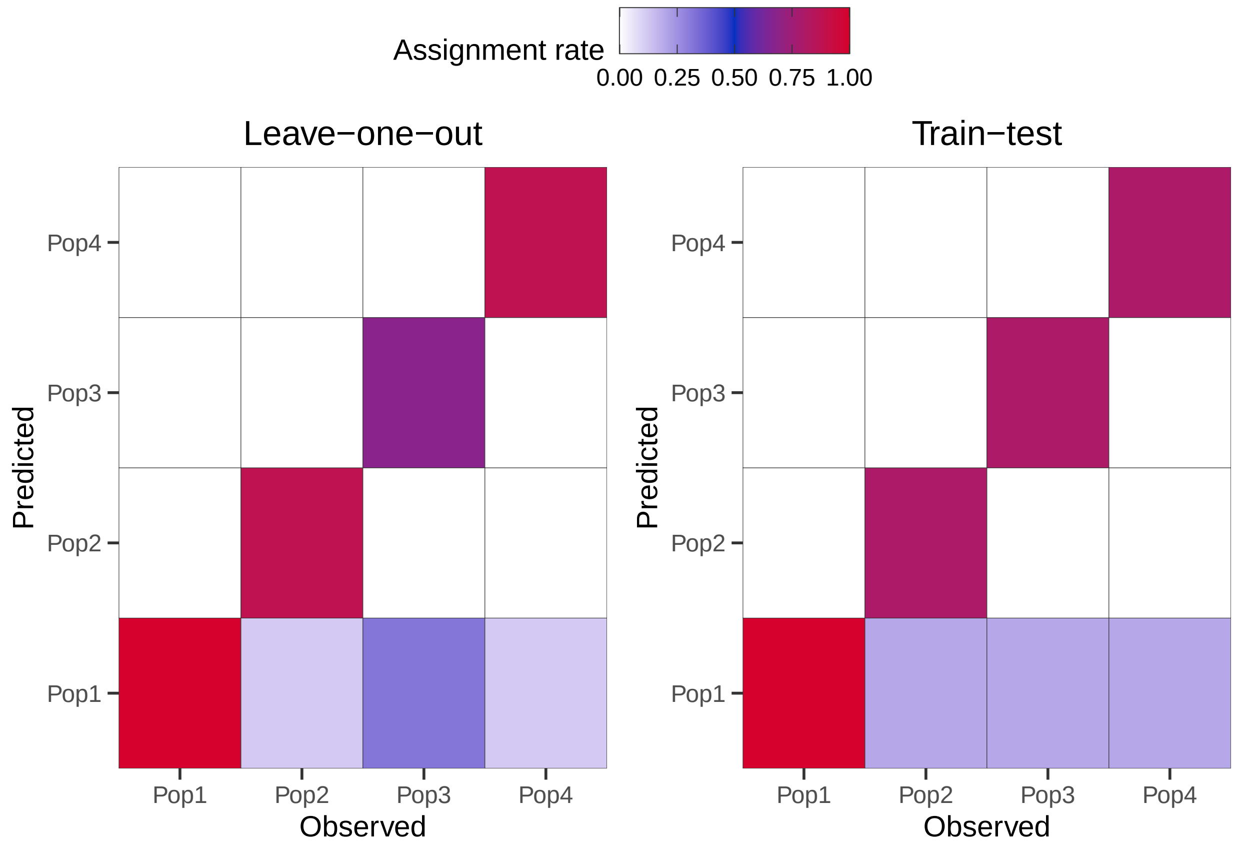
```
DAPC.train.test$pairs.wide
```

```
##     POP Pop1 Pop2 Pop3 Pop4
## 1: Pop1  1.0  0.0  0.0  0.0
## 2: Pop2  0.2  0.8  0.0  0.0
## 3: Pop3  0.2  0.0  0.8  0.0
## 4: Pop4  0.2  0.0  0.0  0.8
```

In the above, we can see that the leave-one-out cross-validation global assignment rate was higher than that for the training-testing partitioning? (Note, your exact result may differ slightly from the one here). Leave-one-out cross-validation may be upwardly biased because only a single sample is predicted at a time and the rest of the dataset (minus 1) is used to fit the model.

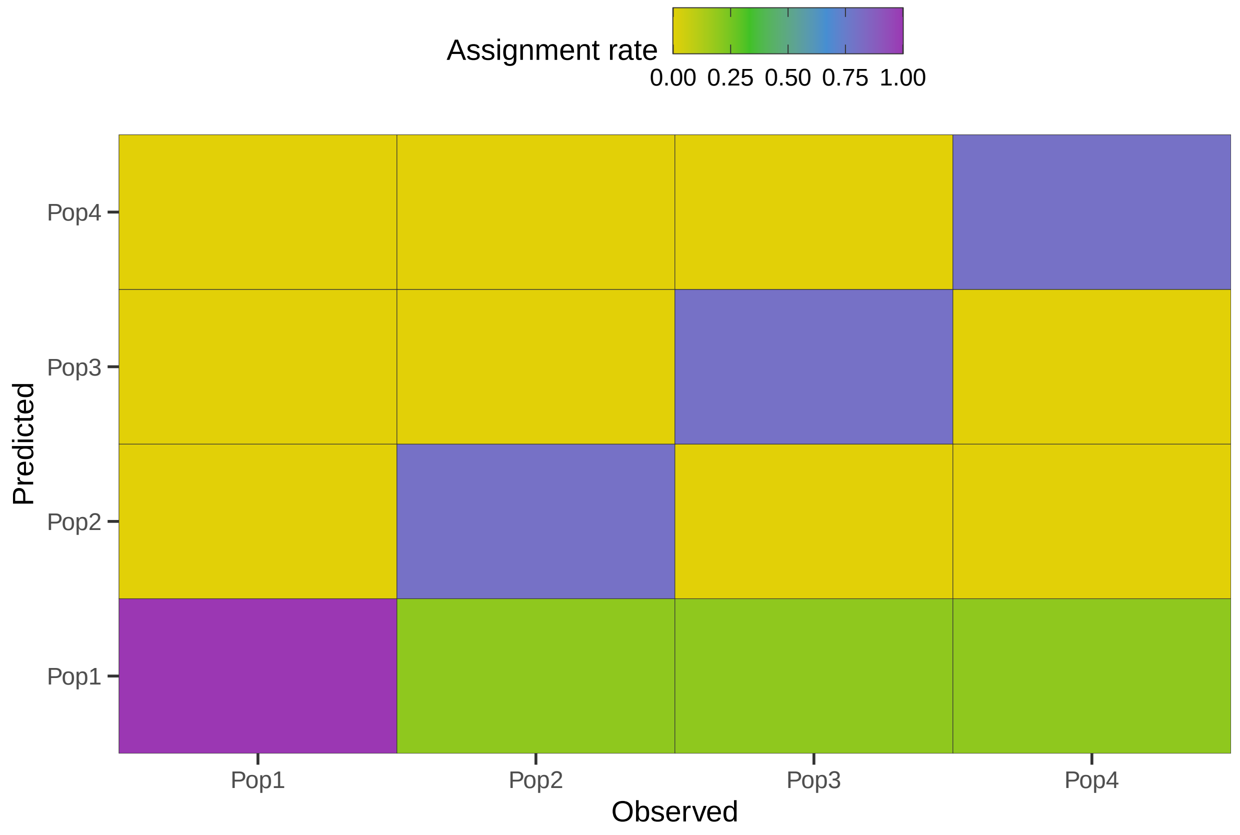
We can pass the output from `dapc_fit(..., type="assign")` to `dapc_plot` to visualise the pairwise assignment rates. We must set the `type` argument in `dapc_plot` to `"assign"`. Let us take a look at the default plot and compare the pairwise outputs for leave-one-out cross-validation and training-testing partitioning side-by-side:

```
plot.dapc.loo.cv <- dapc_plot(DAPC.loo.cv, type='assign')
plot.dapc.tt <- dapc_plot(DAPC.train.test, type='assign')

ggarrange(
  plot.dapc.loo.cv +
    ggtitle('Leave-one-out') +
    theme(plot.title=element_text(hjust=0.5)),
  plot.dapc.tt +
    ggtitle('Train-test') +
    theme(plot.title=element_text(hjust=0.5)),
  common.legend=TRUE
)
```



By default, `dapc_plot(..., type="assign")` will use a gradient from blue to red to represent assignment rates from 0 to 1. We can override this colour gradient by specifying our own spectrum of colours with the `plotColours` argument:

```
plot.dapc.assign.custom <- dapc_plot(
  DAPC.train.test, type='assign',
  plotColours=c('#E2D007','#41C126','#468ED2','#9B37B3'),
  )

plot.dapc.assign.custom
```



Postamble

In this lesson, you have learnt how to analyse population structure with *genomalicious*. You should be familiar with how to calculate F_{ST} and perform PCA and DAPC. You should also be familiar with the *genomalicious* functions that can help you visualise patterns of population structure.

References

Jombart et al. (2010) Discriminant analysis of principal components: a new method for the analysis of genetically structured populations. *BMC Genetiics.* doi: 10.1186/1471-2156-11-94. doi: 10.1038/s41437-020-0348-2.

Miller et al. (2020) The influence of a priori grouping on inference of genetic clusters: simulation study and literature review of the DAPC method. *Heredity.*

Patterson et al. (2006) Population structure and eigenanalysis. *PLoS Genetics.* doi: 10.1371/journal.pgen.0020190.

Thia (2022) Guidelines for standardising the application of discriminant analysis of principal components to genotype data. *Molecular Ecology Resources.* doi: 10.1111/1755-0998.13706.

Weir & Cockerham (1984) Estimating F-statistics for the analysis of population structure. *Evolution.* doi: 10.1111/j.1558-5646.1984.tb05657.x.

Wright (1951) The genetical structure of populations. *Annals of Human Genetics.* doi: 10.1111/j.1469-1809.1949.tb02451.x