Introduction to PLINK and R

Part 1: PLINK

> **Kernel:** run the code cells in this section with the **Bash** kernel.

Running PLINK

Here we are running PLINK in a Juypter notebook. These commands can also be run on a command line. In both cases, additional arguments ("options", "flags") specify exactly what PLINK should do. All arguments are documented on the PLINK website.

Data overview

In this exercise toy example will be used on diastolic blood pressure and the genotypes for 20 SNP markers, which are in a file in PLINK format. This exercise uses four files:

- `dbp.qt.ped` — pedigree file with family, sex, the quantitative trait (diastolic blood pressure), and genotypes
- `dbp.cc.ped` — pedigree file with family, sex, the dichotomized trait (affected yes/no based on blood pressure), and genotypes
- `dbp.map` — map file for the SNP markers (chromosome, SNP ID, genetic distance, position)
- `dbp.age.pheno` — covariate file containing the age of each individual

References

- Chang CC, Chow CC, Tellier LCAM, Vattikuti S, Purcell SM, Lee JJ (2015) Second-generation PLINK: rising to the challenge of larger and richer datasets. GigaScience, 4.
- Purcell S, Neale B, Todd-Brown K, Thomas L, Ferreira MAR, Bender D, Maller J, Sklar P, de Bakker PIW, Daly MJ & Sham PC (2007) PLINK: a toolset for whole-genome association and population-based linkage analysis. *American Journal of Human Genetics*, 81:559-575.

I. Preview Input Files

In this section we'll take a look at the contents of these files. Make sure you understand the meaning of each column.

In [ ]:
```
# Preview dbp.qt.ped
head ./data/dbp.qt.ped
```

In [ ]:
```
# Preview dbp.cc.ped
head ./data/dbp.cc.ped
```

In [ ]:
```
# Preview dbp.map
head ./data/dbp.map
```

In [ ]:
```
# Preview dbp.age.pheno
head ./data/dbp.age.pheno
```

In [ ]:
```
# Set the output directory for this exercise
output_dir="./output/intro_plink_R"
```

II. Missing Data and Filtering

Variables with too many missing values can bias a statistical analysis and lead to spurious results. We'll use PLINK to assess the extent of missing values in the data set, then filter out variables and samples with too many missing observations.

PLINK requires the data set to process as its first argument, specified with the `--ped` and `--map` options. `dbp.map` is already in the standard four-column format (chromosome, SNP ID, genetic distance, position), and this version of PLINK auto-detects the number of columns, so no extra flag is needed for it. First, let's assess the proportion of missing values for each marker and each sample.

> **Note:** All arguments of a PLINK call must go on a single line — anything after a line break is ignored. A trailing backslash (`\`) suppresses the line break and lets a single PLINK call span multiple lines, e.g.:
>
> ```
> plink --ped dbp.cc.ped \
>     --map dbp.map \
>     --missing
> ```

In [ ]:
```
# Run plink command
plink  --ped ./data/dbp.cc.ped  --map ./data/dbp.map  --missing --out ${outp
```

PLINK has created three files: `plink.log` contains all screen output, while `plink.imiss` and `plink.lmiss` contain the proportion of missing values for each sample and marker, respectively. Preview all three files below.

In [ ]:
```
# Preview output file - plink.log
head ${output_dir}/plink.log
```

In [ ]:
```
# Preview output file - plink.imiss
head ${output_dir}/plink.imiss
```

In [ ]:
```
# Preview output file - plink.lmiss
head ${output_dir}/plink.lmiss
```

Next, we'll exclude samples with more than 10% missing genotypes (`--mind 0.10`) and markers with more than 5% missing (`--geno 0.05`). We'll write this filtered data set to new files, `cleaned.ped` and `cleaned.map` , using `--recode` and `--out` . Further quality checks and analyses can then be based on this cleaned data set.

In [ ]:
```
# Run the plink command
plink  --ped ./data/dbp.cc.ped  --map ./data/dbp.map  --mind 0.10  --geno 0.
       --recode  --out ${output_dir}/cleaned
```

PLINK has created three files: a log file, `cleaned.log` (from the `--out` flag), and the two data files `cleaned.ped` and `cleaned.map` . Two markers with too many missing values (`rs1104` and `rs1106`) have been excluded. Preview all three files below.

In [ ]:
```
# Preview output file - cleaned.log
head ${output_dir}/cleaned.log
```

In [ ]:
```
# Preview output file - cleaned.ped
head ${output_dir}/cleaned.ped
```

In [ ]:
```
# Preview output file - cleaned.map
head ${output_dir}/cleaned.map
```

Next, we'll use this filtered data set to estimate the minor allele frequencies (MAF) of the markers with the `--freq` flag. This step estimates the MAFs but does not filter on a minimum frequency (use `--maf` for that). The resulting file, `cleaned.frq` , contains the frequency estimates.

Note: MAF is always ≤ 0.50 — PLINK automatically assigns the reference allele so that `A2` is the major (more frequent, "baseline") allele and `A1` is the minor (less frequent, "risk") allele. This automatic allele flipping applies throughout many PLINK analyses, including association testing, so always check which allele is the baseline and which is the minor/risk allele in your results. For marker `rs1101` , for example, allele `2` is the major allele `A2` and allele `1` is the minor allele `A1` . To avoid automatic allele flipping altogether, use the `--keep-allele-order` flag throughout.

In [ ]:
```
# Run the plink command
plink  --ped ${output_dir}/cleaned.ped  --map ${output_dir}/cleaned.map  --f
```

In [ ]:
```
# Preview the output - cleaned.frq
head ${output_dir}/cleaned.frq
```

Now let's check for deviations from Hardy-Weinberg equilibrium (HWE).

In [ ]:
```
# Run the plink command
plink  --ped ${output_dir}/cleaned.ped  --map ${output_dir}/cleaned.map  --h
```

This step tests for deviations but doesn't filter on a P-value threshold (use `--hwe` for that). The output file `cleaned.hwe` has three result lines per marker: one each for cases, controls, and the complete sample. Each line contains the baseline allele (`A2`), the observed genotype counts (`GENO`), the observed and expected heterozygote frequencies (`O(HET)` and `E(HET)`), and the corresponding P-value (`P`). Preview the resulting file, `cleaned.hwe` , below.

In [ ]:
```
# Preview the output - cleaned.hwe
head ${output_dir}/cleaned.hwe
```

Importing PLINK-Format Data into R

Importing data in PLINK (LINKAGE) format into R can be tricky. A helpful approach is to create tab-separated text files, where columns are separated by a single, well-defined tab character (`\t`). PLINK genotypes normally span two columns (one per allele); since both belong to the same variable, a different separator between the two allele columns — a space instead of a tab — is preferable. Adding the `--tab` flag produces a text file that can be easily read into R with `read.table(..., sep="\t")` .

In [ ]:
```
# Run the plink command
plink  --ped ${output_dir}/cleaned.ped  --map ${output_dir}/cleaned.map  --c
       --tab
```

> **Note:** PLINK can create a large number of files and will overwrite existing ones without warning, which can get confusing — and is a common source of errors when it's unclear which filtering criteria were actually applied to a given data set before an analysis. Planning your filtering/exporting steps ahead of time, and using clear file names and separate output subdirectories, is highly recommended.

III. Binary PLINK Format

Genome-wide marker genotype data can be massive, resulting in very large file sizes. To reduce file size and speed up calculations, genotype information is usually compressed. Let's convert the text files into binary PLINK format:

In [ ]:
```
# Run the plink command
plink  --ped ./data/dbp.cc.ped  --map ./data/dbp.map  --make-bed  --out ${ou
plink  --ped ./data/dbp.qt.ped  --map ./data/dbp.map  --make-bed  --out ${ou
```

PLINK has created four files: `dbp.log` contains all screen output, `dbp.fam` contains the family information, `dbp.bim` the marker information, and `dbp.bed` the marker genotypes in binary (compressed) form. Preview a sample of the `.fam` and `.bim` files below.

In [ ]:
```
# Preview the output - dbp.fam
head ${output_dir}/dbp.fam
```

In [ ]:
```
# Preview the output - dbp.bim
head ${output_dir}/dbp.bim
```

Part 2: R

> **Kernel:** run the code cells in this section with the **R** kernel.

Starting R

If you're unsure how to use a function, or want to see its full set of arguments, use the `help()` function and `?` help operator in R to access the documentation pages for R functions, data sets, and other objects.

References

- R Core Team (2025), R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing

I. Data Types

Data can be of different types — for example numeric, string, or logical values. Suppose we want to compile a short list of European cities with a few features for each.

In [ ]:
```
# I: Data Types
city       = c("Oslo", "Bergen", "Munich", "Berlin", "Rome", "Milan")
population = c(0.58, 0.25, 1.3, 3.4, 2.7, 1.3)
country    = factor( c("Norway" , "Norway", "Germany",
                       "Germany", "Italy", "Italy"   ))
capital    = c(TRUE, FALSE, FALSE, TRUE, TRUE, FALSE)
updated    = 2009
```

You've now created several data objects — city names, population sizes, countries, capital-city status, and year of last update — in R's working memory using the assignment operator `=`. To print an object's contents, simply type its name:

In [ ]:
```
# Return the contents of the objects
city
population
country
capital
```

Each of these objects is a **vector** — all its elements share the same data type. For example, `city` contains only strings (characters), while `capital` contains only logical values indicating whether each city is its country's capital. Vectors can be concatenated with `c()`, and `summary()` gives a short summary of an object — what it does depends on the object's data type (its *class* in R); for instance, a mean can be calculated for numeric variables but not for nominal ones (represented as the `factor` type in R). An object's type can be checked with various functions. Data type is one attribute of an object, but objects can have more than one attribute — another example is **length**, the number of elements in the object (i.e. the number of entries in a vector):

In [ ]:
```
# Concatenate the vectors
c(city, city)
c(population, updated)

# Get a summary of the objects
summary (city)
summary (population)
summary (country)
summary (capital)

# Assess the type of object
is.numeric(city)
is.character(city)
is.factor(city)

class (city)
class (population)
class (country)
class (capital)

# Return the length (the number of elements of an object)
length(city)
```

II. Names & Indexes

For better data organization, access, and presentation, elements in a vector can have names. In many analyses you'll want to access only part of a data set, or even a single element.

For example, markers should often be tested for deviations from Hardy-Weinberg equilibrium (HWE) separately in affected and unaffected samples. Elements of data objects are accessed via **indexes**, of which there are three kinds. The simplest is addressing by position, shown in the code block below.

If an object's elements have names, those names can also be used to access them. A third option for vectors is a **logical index**, where only elements marked `TRUE` are accessed — for example, selecting only the capitals from our list of cities:

In [ ]:
```
# II: Names & Indexes #

# Return the name of the elements in a vector
names(population) = city
population

# Access elements of objects by position
city [3]
city [2:4]
city[c(1,5:6)]
population[3]

# Access element of an object by names
population["Oslo"]
population[c("Berlin","Rome")]

# Access elements using a logical index
population
capital
population[capital]
```

Logical indexes are powerful: you can formulate a condition, store the result as a logical vector, and reuse that vector to repeatedly access only the elements meeting the condition. For example, the following commands select only the cities with a population of at least one million:

In [ ]:
```
# Select only those cities from our list which have a population of at least
population>=1.0
population[population>=1.0]
```

III. Data Frames

Data sets are often presented in tabular form: columns usually represent features or measurements and share a single data type, while rows represent observations or samples and may mix different data types. Worksheets from SPSS or Excel, and tables extracted from SQL databases, typically follow this format.

Data frames are special lists where all vectors (features/columns) have identical length, plus added functionality for printing, summarizing, and so on. They have two dimensions — rows and columns — and both can have names.

In [ ]:
```
# III: Data frames

# Representation of a data frame
cities = data.frame (city=city, pop=population,
```

```
                      country=country, capital=capital, stringsAsFactors=F)
cities
length(cities)
dim(cities)
is.data.frame(cities)
is.list(cities)

# Retrieve column names of the data frame
colnames(cities)
# Retrieve row names of the data frame
rownames(cities)
```

Indexing works similarly to vectors, but since data frames have two dimensions (rows and columns), we need two indexes. We can access single elements, or entire rows or columns.

In [ ]:
```
# Index the data frame using logical indexes
cities$city
cities[,1]
cities[2,]
cities[2,3]
cities$pop[3]
cities[capital,]
cities[cities$pop>=1.0,]
```

IV. Export & Import

All R objects live in working memory, and are lost when you quit R unless you've saved them to disk. There are several ways to save objects. It's often useful to export a data set to text format so it can be opened in a text editor (or Word) or imported into other software — for data frames, this can be done with `write.table()`.

In [ ]:
```
# IV: Export & Import

# Get an overview on which objects are currently held in the working memory:
ls()

# Save the objects cities, city, and country in an external archive file cal
# Note that this file is in a format that is only readable with R!
save(cities, city, country, file="output/intro_plink_R/myobjects.R")

# Export the data set to text format
write.table(cities, file="output/intro_plink_R/cities.txt")

# Re-direct output from R console to a text file
sink ("output/intro_plink_R/cities.output.txt")
print (cities)
sink ()

# Check if these files have properly been created in the working directory c
dir()
```

The `dir()` command lists the contents of your current working directory. If `cities.txt` and `cities.output.txt` weren't created, check your script for errors or ask for help. Once they've been created, delete all objects from R's working memory.

In [ ]:
```
# Delete all objects from R working memory
rm(list=ls())
ls()

# Import the data frame from the external text file
new.table = read.table ("./output/intro_plink_R/cities.txt")
ls()
new.table

# Import the objects from the R archive file
load ("output/intro_plink_R/myobjects.R")
ls()
cities
new.table
```