Detecting Natural Selection

Contents

Important concept:
Green boxes summarises important concepts from the text.

Additional info
Blue boxes contain small tips and additional information for those that are interested. What's in these is hopefully useful, but not mandatory. Feel free to skip these if you want.

Advanced code
Yellow boxes contain code that you need to run in order to complete the tutorial, but that you don't necessarily need to understand.

Text in bold contains small exercises to do on your own throughout the tutorial. These are for your own understanding only, so you don't need to hand them in.

Introduction

In the last section, we spent a lot of time on the theoretical basis of natural selection. In this section, we will once again learn about selection, but this time our focus will be much more empirically based. In particular, we are going to use R to demonstrate the utility of F_{ST} (i.e. genetic differentiation among populations) as a means to infer selection in the genome. This means we will be returning to some of the things you learned about in Chapter 3 in order to properly understand F-statistics. We'll also return to the idea of empirical and statistical distributions in order to underline the basic concepts behind how we can do this. Finally, we will take some actual genome-wide F_{ST} data and visualise it in order to demonstrate just how a genome scan approach might identify genomic regions under selection.

What to expect

In this section we will:

- Learn more about functions in R
- Learn about the `ifelse()` function
- develop our understanding of F-statistics
- visualise F_{ST} across the genome and use it to detect potential divergent selection
- learn about empirical and statistical distributions in R

Getting started

The first thing we need to do is set up the R environment. We won't be using anything other than base R and the `tidyverse` package today. So you'll need to load the latter.

```
library(tidyverse)
```

1 R-programming

1.1 More on functions: vectorisation

Last week you learned a bit about creating your own functions, and hopefully saw how useful they can be for simplifying your work flow. This week you will learn more about how they work, and some tricks for using functions efficiently.

Let's use the `selection_model_simple()` function we made in last week's tutorial as an example. It takes p in the current generation, along with the relative fitness, and returns p_t, i.e. p in the next generation

```
selection_model_simple <- function(p, rel_fit){
  # define q
  q <- 1 - p
  # calculate genotype frequencies (under HWE)
  gf <- c(p^2, 2*(p*q), q^2)

  # calculate mean pop fitness
  w_bar <- sum(rel_fit*gf)

  # calculate marginal allele frequencies
  w1 <- (p*rel_fit[1]) + (q*rel_fit[2])
  w2 <- (p*rel_fit[2]) + (q*rel_fit[3])

  # calculate freq of p in the next generation
  p_t <- (p*w1)/w_bar

  # return the results
  return(p_t)
}
```

This can be used for a single value of `p`:

```r
# make a rel fit object to use for all models
rel_fit <- c(1, 1, 0.7)
p <- 0.3
selection_model_simple(p, rel_fit)
#> [1] 0.3516999
```

However, due to how R treats vectors, it can also be used for a vector of p-values!

```r
# create p-values from 0.1 to 0.9 in steps of 0.1
p <- seq(0.1, 0.9, 0.1)
# calculate genotype frequencies for all values of p!
selection_model_simple(p, rel_fit)
#> [1] 0.01234568 0.02469136 0.03703704 0.04938272 0.06172840 0.07407407 0.08641975 0.09876543
#> [9] 0.11111111
```

This is part of what we call *vectorisation*, taking a single function and applying it to a vector of values.

Important concept:
Many functions can be applied to a vector of values instead of a single value. This is a quick and simple way of running a lot of functions at the same time. Notice how the function call `selection_model_simple(p, rel_fit)` is exactly the same for single values and vectors!

Tip:
Notice that you could also have used a for loop to use the function on a range of p-values. If you have experience with other programming languages, e.g. Python, you're probably used to do it that way! In R, however, vectorising functions is often faster than using for-loops if your vectors get really large.

1.2 The `apply()` function

The `apply()` function is a tool to further vectorise your functions that works with matrices and data frames. `apply()` will *apply a function to either each row or each column of your data frame/matrix.* The general structure of using `apply()` is:

```r
apply(data, row_or_colwise, function_name)
```

Where the second argument is either `1` for operating on each row or `2` for operating on each column[1].

The best way to show how it works is probably through an example. Consider if you have the following data frame of species counts, where each column contains counts for a species, and each row is a location.

```r
species <- data.frame(
  sp_1 = c(0, 3, 2, 6, 7),
  sp_2 = c(4, 2, 3, 0, 1),
  sp_3 = c(2, 2, 0, 0, 1))
species
#>   sp_1 sp_2 sp_3
#> 1    0    4    2
#> 2    3    2    2
#> 3    2    3    0
#> 4    6    0    0
#> 5    7    1    1
```

[1] It can be tricky to remember which is which of these, so don't worry if you find yourself looking at the help page with `?apply` all the time, I sure do! Also, a general rule of thumb is that in R, *rows always comes before columns*, like when you extract values from a data frame with square brackets `[]`.

What if you want to take the mean count for each location? I.e. calculate the mean of each row in the data set. You can't simply use `mean()` on your data frame:

```
mean(species)
#> [1] NA
```

R doesn't understand what to do in this case. However, this is a perfect opportunity for using `apply()`! Remember that the second argument should be `1` since we want to work on rows here.

```
apply(species, 1, mean)
#> [1] 2.000000 2.333333 1.666667 2.000000 3.000000
```

Great! Notice how `mean` should be written without parentheses when using it inside `apply()`. This is something you unfortunately just have to remember.

Exercise: use `apply()` calculate the mean count of each species, i.e. the means of the columns in your data.

Show hint

Use `2` as the second argument to work with columns instead of rows.

```
apply(species, 2, mean)
#> sp_1 sp_2 sp_3
#>  3.6  2.0  1.0
```

Important concept:
Use `apply()` to use a function on either all rows (`1`) or all columns (`2`) of a data frame/matrix.

This section has been a small taste on what R can do when you vectorise your functions. Vectorisation can be a bit tricky to wrap your head around in the beginning, but if you keep using it it eventually becomes a very useful tool for performing a lot of calculations simultaneously.

1.3 The `ifelse()` function

Say that you have a vector with values, and you want to group them somehow. For example, you have a vector of numbers between 0 and 10, and want to label the values that are larger than 5.

```
# generate 10 numbers between 0 and 10
set.seed(14)
x <- sample(0:10, 10)
x
#>  [1]  8 10  3  2  5  6  9  1  7  0
```

Now, you may remember from the first week that you can check which numbers are more than 5 by using a logical statement:

```
x > 5
#>  [1]  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE
```

However, what if you want to label the values "high" and "low", respectively? For this, you can use the `ifelse()` function. Conceptually, `ifelse()` works like this:

```
ifelse(logical_statement, value_if_TRUE, value_if_FALSE)
```

This means that to get "high" and "low" for our values, we can write:

```
ifelse(x > 5, "high", "low")
#>  [1] "high" "high" "low"  "low"  "low"  "high" "high" "low"  "high" "low"
```

1.3.1 `ifelse()` examples

`ifelse()` can be used for a lot of things, here are a couple of more examples:

Check if a value is higher than the mean:

```
ifelse(x > mean(x), "above mean", "below_mean")
#>  [1] "above mean" "above mean" "below_mean" "below_mean" "below_mean" "above mean" "above mean"
#>  [8] "below_mean" "above mean" "below_mean"
```

Convert a value from count to binary presence/absence:

```
ifelse(x > 0, 1, 0)
#>  [1] 1 1 1 1 1 1 1 1 1 0
```

We can also use it for character vectors, looking for a specific word to make groups:

```
animals <- c("horse", "donkey", "zebra", "horse", "zebra", "mule")
ifelse(animals == "horse", "actual horse", "almost horse")
#> [1] "actual horse" "almost horse" "almost horse" "actual horse" "almost horse" "almost horse"
```

This usage can be convenient when plotting values, as you will see in the evolution-part of the tutorial.

2 Evolutionary biology: $\mathbf{F_{ST}}$

2.1 Understanding F_{ST} - the fixation index

2.1.1 What is F_{ST}?

F_{ST}, also known as the **fixation index**, is an extremely important statistic in population genetics, molecular ecology and evolutionary biology. It is also arguably one of the most famous population genetic statistics you will encounter. *F*~ST essentially measures the level of **genetic differentiation** between two or more populations. It ranges from 0 (i.e. no genetic differentiation) to 1 (complete genetic differentiation)

Ultimately, it is quite a simple statistic to understand but it can sometimes take time to properly grasp. So we will go over the basics properly here. One of the most confusing things about *F*~ST is that are several different ways to define it. For ease of undersanding, we will use a simple formulation:

$$F_{ST} = \frac{H_T - H_S}{H_T}$$

For simplicity, imagine we are examining two populations only. With this formulate, H_T is the **expected heterozygosity** when the two populations are considered as one large meta-population. H_S is the **average expected heterozygosity** in the two populations.

You might be thinking, hang on a minute... what do we mean by expected heterozygosity? To appreciate this, we need to think back to the Hardy-Weinberg expectation we learned about in Chapter 3. Remember that at a simple bi-allelic locus, p and q are the frequencies of the two alleles. We can calculate the expected frequency of heterozygotes with $2pq$ - this is the **expected heterozygosity**.

2.1.2 A worked example of F_{ST} in humans

As an illustrative example, we will calculate F_{ST} for the SNP rs4988235 associated with lactase persistence in humans. This SNP is located ~14 Kb upstream of the *LCT* gene on Chromosome 2 and is biallelic for C/T; a T at this position is strongly associated with the ability to digest milk in adulthood. We sample 80 people each from two populations which differ in the frequency of lactase persistence - Americans of European descent and Druze people from Israel. The counts of genotypes are shown in the table below. Note that these data are modified from Bersaglieri et al. 2002.

Population	TT	CT	CC
American	48	28	4
Druze	0	3	77

Knowing these numbers, we will first calculate the allele frequences in each population. We will use p to denote the frequency of the T allele at this locus.

```
# set up genotype counts
a <- c(48, 28, 4) # americans
d <- c(0, 3, 77) # druze

# get the number of people sampled (same for both)
n <- sum(a)

# calculate the frequency of the T allele - or p
# for americans
p_a <- ((a[1]*2) + a[2])/(2*n)
# for druze
p_d <- ((d[1]*2) + d[2])/(2*n)

# calculating the frequency of C (or q) is then trivial
q_a <- 1 - p_a
q_d <- 1 - p_d
```

Next we can calculate the allele frequencies for the metapopulation - i.e. Americans of European descent and Druze considered as a single population. This is as simple as taking the mean of the two allele frequencies.

```
# calculate total allele frequency
p_t <- (p_a + p_d)/2
q_t <- 1 - p_t
```

With these allele frequencies calculated, we can very easily calculate expected heterozygosities - remember this is just $2pq$.

```
# first calculate expected heterozygosity for the two populations
# americans
hs_a <- 2*p_a*q_a
```

```
# druze
hs_d <- 2*p_d*q_d
# then take the mean of this
hs <- (hs_a + hs_d)/2

# next calculate expected heterozygosity for the metapopulations
ht <- 2*p_t*q_t
```

With all the relevant expected heterozygosities in place, we are now ready to calculate F_{ST} which we can do like so:

```
# calculate fst
fst <- (ht - hs)/ht
```

If your calculations were correct, then you should have an F_{ST} estimate of 0.59 - this is very high for between two human populations. One way to interpret the F_{ST} value we have here is that 59% of genetic variance we observe differs between populations. Since population can explain such a large difference in this case, we might expect selection to be responsible…

2.1.3 Writing a set of F_{ST} functions

The code in the previous section was useful to demonstrate how we can calculate F_{ST}, but it would be a lot of work to run through this every single time we want estimate the statistic for a locus. This being R, we can of course easily create a function that will do all of the leg work for us! We will take the code we wrote out in the last section and use it here to write two functions that we can use when we want to calculate F_{ST}. Note that for simplicity, we will only write functions that work for **two populations**.

First, we will write a function called `calc_af` which will take genotype counts from a population and calculate allele frequencies. This will probably be similar to the function you made in last week's assignment.

```
# a simple function to calculate allele frequencies in two populations
calc_af <- function(counts){
  # get the number of samples
  n <- sum(counts)
  # calculate frequency of 1st allele - p
  p <- ((counts[1]*2) + counts[2])/(2*n)
  return(p)
}
```

Since it is very straightforward for use to calculate the frequency of the second allele once we have the frequency of the first (i.e. $q = 1 - p$), our `calc_af` function will only calculate p for both populations. Let's test it on the data from our previous example.

```
# testing our function on the american/druze example
af_american <- calc_af(c(48, 28, 4))
af_druze <- calc_af(c(0, 3, 77))
```

So now that we have a function that calculates allele frequencies in the two populations, we can write our `calc_fst` function to take these frequencies and calculate F_{ST} from them.

```r
# a function to calculate fst
calc_fst <- function(p_1, p_2){

  # calculate q1 and q2
  q_1 <- 1 - p_1
  q_2 <- 1 - p_2

  # calculate total allele frequency
  p_t <- (p_1 + p_2)/2
  q_t <- 1 - p_t

  # calculate expected heterozygosity
  # first calculate expected heterozygosity for the two populations
  # pop1
  hs_1 <- 2*p_1*q_1
  # pop2
  hs_2 <- 2*p_2*q_2
  # then take the mean of this
  hs <- (hs_1 + hs_2)/2

  # next calculate expected heterozygosity for the metapopulations
  ht <- 2*p_t*q_t

  # calculate fst
  fst <- (ht - hs)/ht

  # return output
  return(fst)
}
```

Let's test our function on the allele frequencies we calculated with our `calc_af` function.

```r
# testing our function on the american/druze example
calc_fst(af_american, af_druze)
```

This should be the same as you got before, but with a lot less work. Next, we'll look at applying a function to a bigger dataset with `apply()`.

2.1.4 Applying functions to matrices and data frames

Extending our *LCT* and lactase persistence example, let's get some data from multiple human populations. You can download the data here

Import the data into R using the `read.table()` function. If you're unsure about how you do this, remember that you can go back and check in the tutorial from the first week

Show hint

The data is separated by tabulator (`"\t"`), and has a header.

```r
lct_counts <- read.table("lct_count.tsv", header = TRUE, sep = "\t")
```

You should now have a data frame in your R environment with allele counts for the SNP rs4988235 for 53 populations. Again, these data are all from Bersaglieri et al. 2002.

What we have is the counts of allleles but what we actually want is the allele frequency for T - that is how we can calculate F_{ST}. We can use our `calc_af()` function for this, so let's try this function out on counts for a single population. We use indexing here to select the first row and only columns 2:4, since our function is only expecting the count data, not the population name.

```
calc_af(lct_counts[1, 2:4])
```

Great! So this works well. Now let's get p (the frequency of the T allele) for all the populations. We can do this extremely fast and easily using the `apply()` function that you learned about in the R-section. Note that the data needs to be columns 2 through 4 of the data, and the second argument needs to be `1` since we're working on rows.

```
p <- apply(lct_counts[,2:4], 1, calc_af)
```

We can now combine our vector of allele frequencies with the population names to create a data.frame of frequencies. Like so

```
lct_freq <- data.frame(pop = lct_counts$pop, p)
```

Now we can easily calculate a pairwise F_{ST} with our `calc_fst` function. For example, let's calculate F_{ST} for European Americans and East Asians. We will use `dplyr` commands for this. First we make subsets of the data with `filter()`. We then end up with a single row of our data. We provide the `p`-column of that subset to `calc_fst` using `$`[2].

```
# extract frequencies
af_euram <- filter(lct_freq, pop == "European_American")
af_eastasian <- filter(lct_freq, pop == "East_Asian")
# calculate fst
calc_fst(af_euram$p, af_eastasian$p)
```

As with our previous example, we can see F_{ST} is actually pretty high between these populations for this SNP. What about if we compared East Asians with the Bedouin people from Israel?

```
# extract frequencies
af_bedouin <- filter(lct_freq, pop == "Bedouin_Negev_Israel")
# calculate fst
calc_fst(af_eastasian$p, af_bedouin$p)
```

Here we see F_{ST} is substantially lower. Allele frequency differences are lower between these populations.

2.2 Visualising F_{ST} along a chromosome

Next, we will combine vectorisation and our custom functions to calculate F_{ST} for a series of SNPs in the vicinity of the *LCT* gene on chromosome 2. This is essentially a genome scan, an approach that can be used to detect signatures of selection in the genome. You can download the data here

Read in the file `LCT_snps.tsv`.

[2]Tip: to better understand this code, try printing the objects `af_euram` and `af_eastasian`, as well as the columns `af_euram$p` and `af_eastasian$p`. This way you can follow what the code is doing.

```r
lct_snps <- read.table("LCT_snps.tsv", header = TRUE, sep = "\t")
```

This data is also from from Bersaglieri et al. 2002. It is the allele frequency in various human populations for one allele at a set of 101 biallelic SNP markers close to the *LCT* gene on chromosome 2 in the human gene. Each row is a SNP and there are three frequencies - one for North Americans of European descent, one for African Americans and one for East Asians.

Since we have the allele frequencies, we can easily calculate F_{ST} for each of these SNPs. For our example here, we will do this between `european_americans` and `east_asians`. First of all, let's use our `calc_fst` function on just a single SNP.

```r
calc_fst(lct_snps[1, "european_americans"], lct_snps[1, "east_asians"])
```

So it works great for a single row. Actually, `calc_fst()` will work with vectors of values as well, so to calculate the F_{ST} for all loci simultaneously, we can supply the entire columns as arguments with the `$` operator.

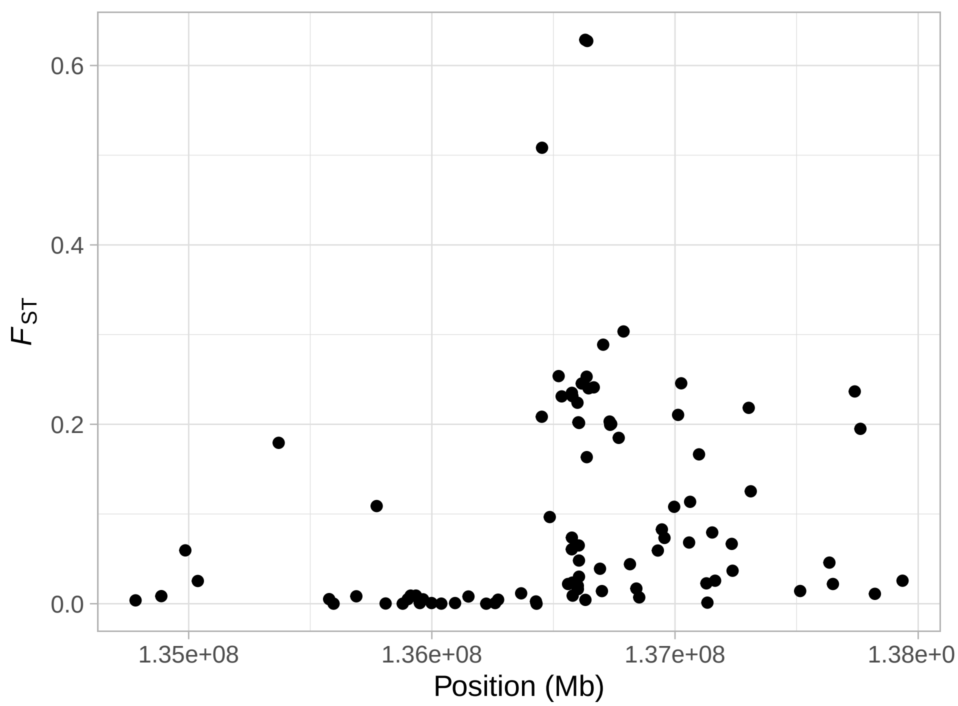
```r
calc_fst(lct_snps$european_americans, lct_snps$east_asians)
```

It works equally great for the entire data set! Let's add these F_{ST}s as a column in the `lct_snps` data frame.

```r
# make an fst column
lct_snps$fst <- calc_fst(lct_snps$european_americans, lct_snps$east_asians)
```

Now that we have F_{ST} estimates for each of our SNPs, we can visualise the variation along the chromosome with `ggplot2`.

```r
a <- ggplot(lct_snps, aes(coord, fst)) + geom_point()
a <- a + xlab("Position (Mb)") + ylab(expression(italic(F)[ST]))
a <- a + theme_light()
a
```

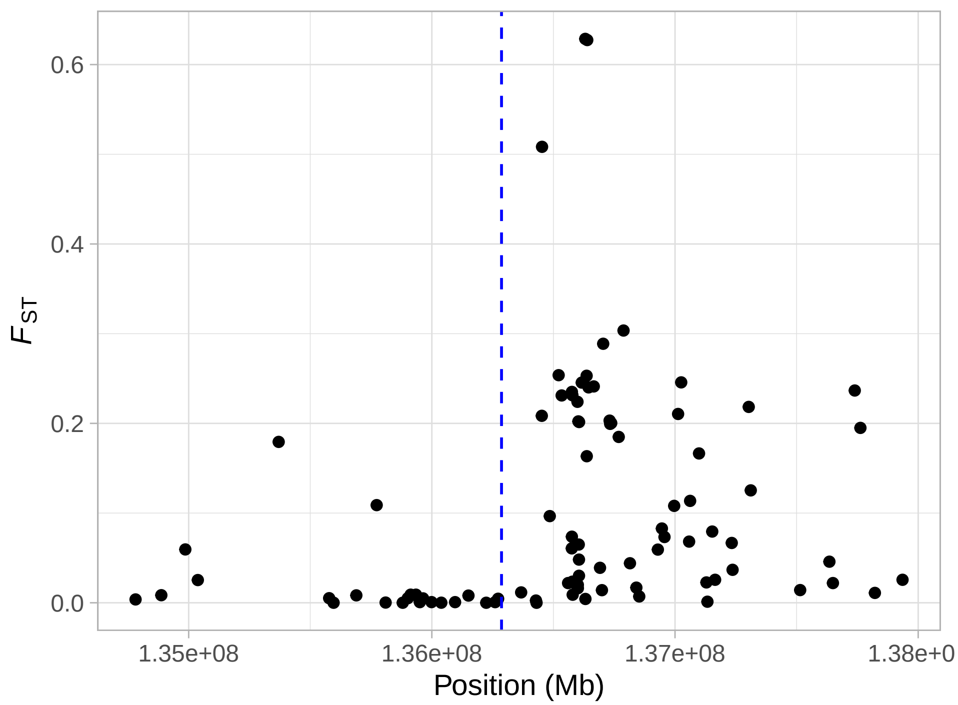


What are we seeing here? Quite clearly, there is a an increase in F_{ST} along the chromosome, with a few SNPs showing extremely high values. It might make things a bit clearer if we mark on our plot the midpoint of the *LCT* gene. We know the gene occurs between 136,261,885 bp and 136,311,220 bp on Chromsome 2 (from the UCSC Genome Browser). So first we will find the midpoint of the gene.

```
# define the start and stop positions of the gene
lct_start <- 136261885
lct_stop <- 136311220
# calculate the midpoint
lct_mid <- (lct_start + lct_stop)/2
```

All we need to do to add it to our plot is add the `geom_vline()` geom.

```
a <- a + geom_vline(xintercept = lct_mid, lty = 2, col = "blue")
a
```



When the mid point of the gene is marked, it is clear that there is an increase in F_{ST} just upstream from the *LCT* gene. Perhaps we want to highlight the SNP that we calculated F_{ST} for in our first example?

This is a perfect opportunity to use the `ifelse()` function we learned about in Section 1.3. We can make a new column in our data based on whether or not `lct_snps$snp_id` is exactly equal to "rs4988235".

Exercise: add a column named status to `lct_snps` using `ifelse()`. The column should contain "Yes" if the id is rs4988235, and "No" if it's not.

Show hint

Start with `lct_snps$status <-` to assign the result to a new column.

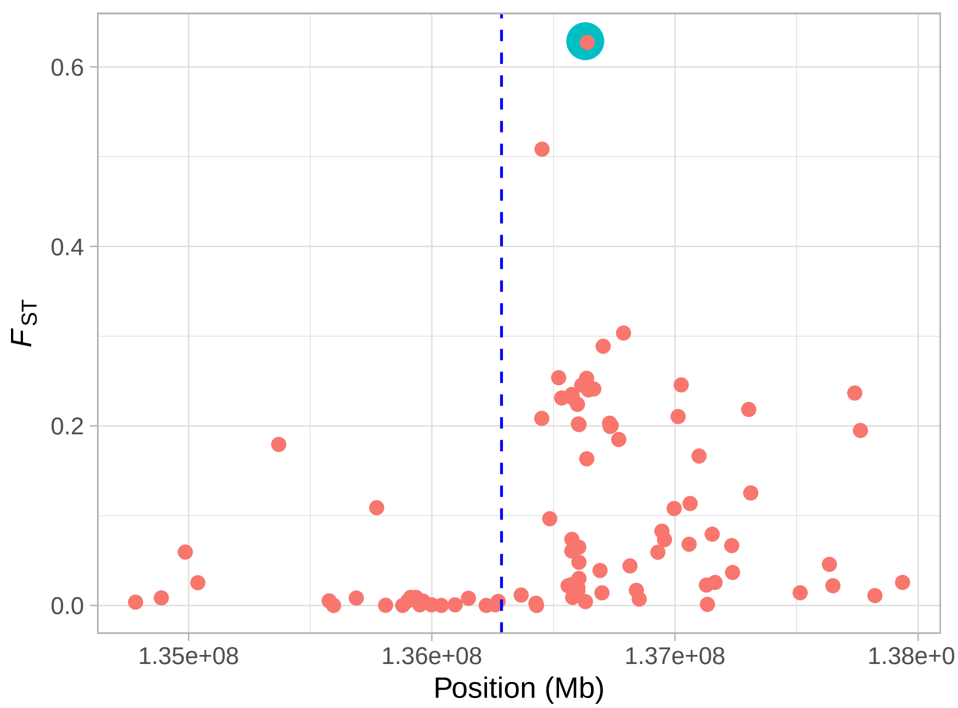
Show another hint

The logical statement you need to use is `lct_snps$snp_id == "rs4988235"`

```r
lct_snps$status <- if_else(lct_snps$snp_id == "rs4988235", "Yes", "No")
```

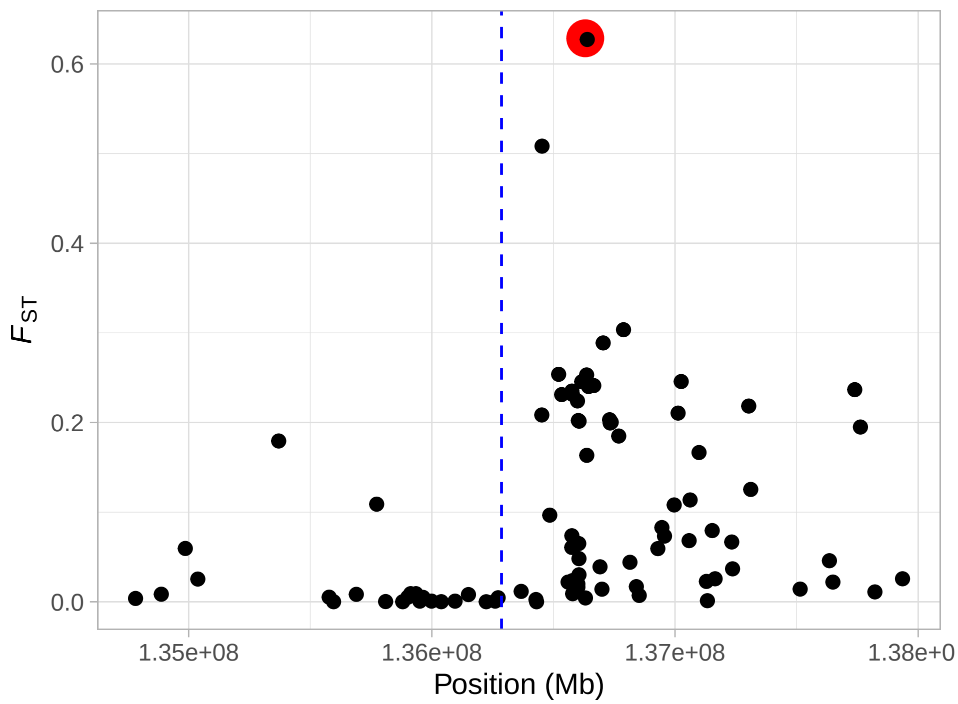
Now to highlight the SNP on our plot. We can make it big and colored this by mapping it to both the `col` and `size` aesthetic.

```r
a <- ggplot(lct_snps, aes(coord, fst, col = status, size = status)) + geom_point()
a <- a + xlab("Position (Mb)") + ylab(expression(italic(F)[ST]))
a <- a + geom_vline(xintercept = lct_mid, lty = 2, col = "blue")
a <- a + theme_light() + theme(legend.position = "none")
a
```



Now we see, our focal SNP is highlighted in the plot. We'll change the colours to make it a little bit clearer.

```r
a + scale_colour_manual(values = c("black", "red"))
```

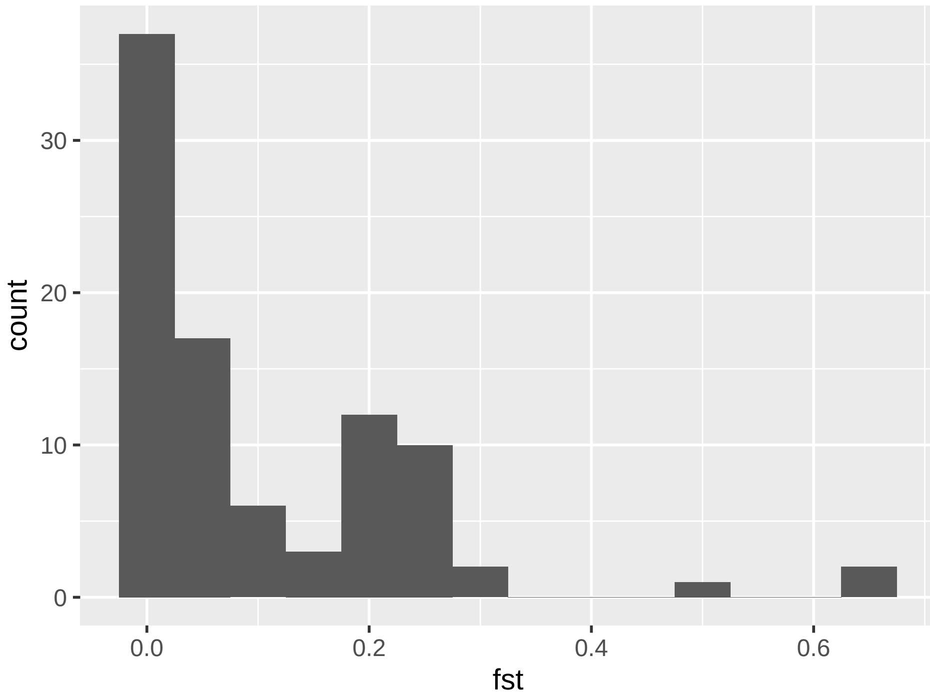


In the next section, we'll demonstrate how we can use the distribution of F_{ST} to identify **outliers** as potential targets of selection.

2.2.1 Identifying outliers in our F_{ST} distribution

How can we identify outliers in our F_{ST} data? First of all, we can look at the distribution of our data by making a histogram of `fst`.

```
ggplot(lct_snps, aes(fst)) + geom_histogram(binwidth = 0.05)
```



Now, it's apparent that some values are way larger than the rest, but where do we set the threshold? One way to do it is to set some arbitrary value, and say that all values larger than this should be considered outliers. This can for instance be that we mark the highest 5% as outliers. In R, we can get this value with the `quantile()` function.

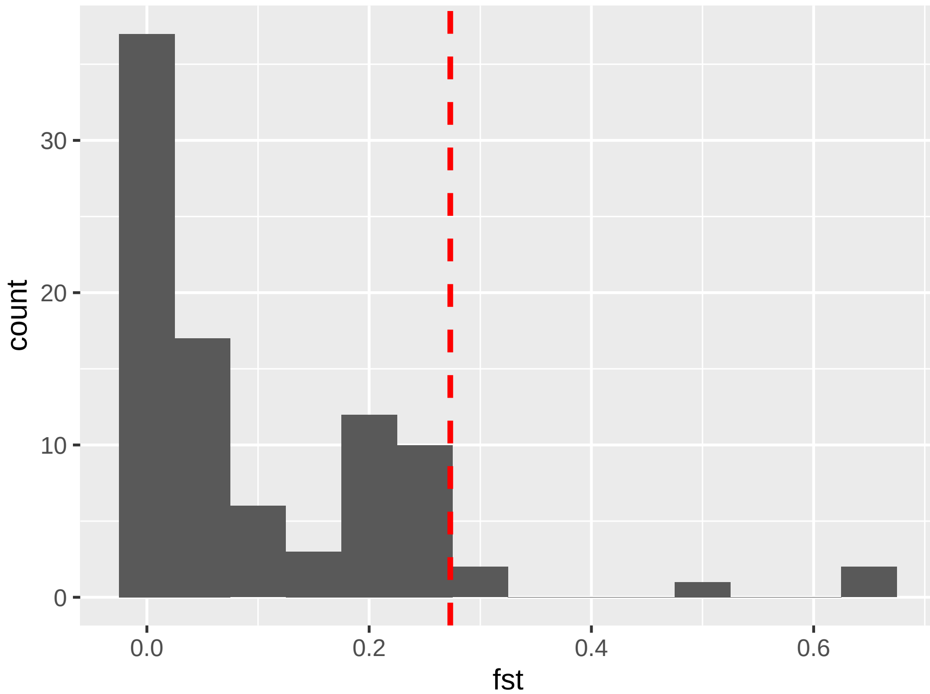
In the following example, we make a vector of numbers from 0 to 100, and use the quantile function to find the highest 5% (or lowest 95%).

```
x <- 0:200
quantile(x, 0.95)
#> 95%
#> 190
```

The function returns `190`, which means that 95% of the values are below 190, and 5% of the values are above 190 (which shouldn't be all that surprising).

Now we use the `quantile()` function on our data to set the outlier threshold. Note that this time we need to add `na.rm = T` in order to ignore some loci which have no F_{ST} estimates.

```
# set threshold
threshold <- quantile(lct_snps$fst, 0.95, na.rm = T)
# plot histogram with threshold marked
a <- ggplot(lct_snps, aes(fst)) + geom_histogram(binwidth = 0.05)
a + geom_vline(xintercept = threshold, colour = "red", lty = 2, size = 1)
```

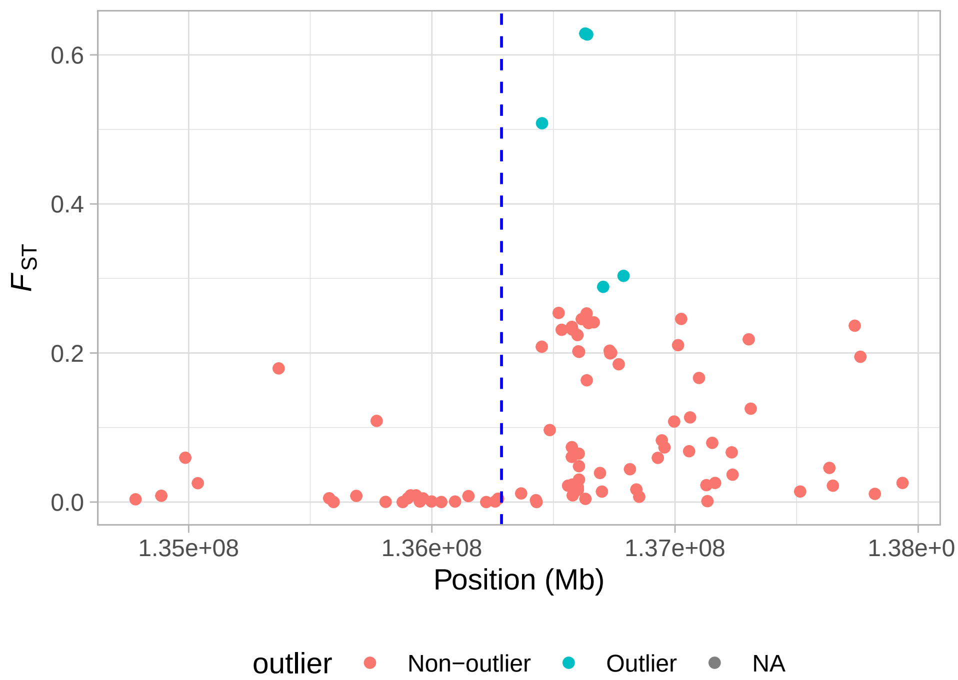


Now what if we want to visualise this on our chromosome-wide plot? Once again, we need to use the `ifelse()` function.

```
lct_snps$outlier <- ifelse(lct_snps$fst > threshold, "Outlier", "Non-outlier")
```

Take a look at the `lct_snps` data frame - you should now see an additional column which is a character vector with the status of each locus as either outlier or non-outlier. Next we can incorporate this into our plotting:

```
a <- ggplot(lct_snps, aes(coord, fst, colour = outlier)) + geom_point()
a <- a + xlab("Position (Mb)") + ylab(expression(italic(F)[ST]))
a <- a + geom_vline(xintercept = lct_mid, lty = 2, col = "blue")
a <- a + theme_light() + theme(legend.position = "bottom")
a
```



So now our potential outlier SNPs are marked on the figure. There are only 5 of them but they all occur just upstream from the *LCT* locus.

We cannot say for certain that these SNPs have increased F_{ST} values because of selection - other processes such as genetic drift or demographic history (i.e. a bottleneck in one of the two populations) might be responsible. However, given our knowledge that *LCT* is involved in lactase persistence, we can at least hypothesise that this is the case.

One important point to note here is that the threshold we set to identify a SNP as being potentially under selection is **entirely arbitrary**. In a way this line of thinking forces us to think of selection acting in some binary way on some SNPs and not others. This is obviously not the case. Still, for SNP data like this an F_{ST} scan can be a very useful tool.

2.3 Study questions

For study questions on this tutorial, download the `Chapter5_R_questions.R` from Canvas or find it here.

2.4 Going further

- Graham Coop's notes on F statistics
- A detailed tutorial on calculating population differentiation with several R-based population genetic packages
- A nice, thorough exploration of the normal distribution using R functions and plotting