**Lab 1.2. Creating a read count table**

In this lab we're going to describe how to count sequencing reads which fall into genomic features using Bioconductor. We want to end up with a count matrix with rows corresponding to genomic ranges and columns corresponding to different samples. To this end we are going to end up with “summarized experiment” object.

As an example, we will use an RNA-Seq experiment, with files in the “pasillaBamSubset” Bioconductor data package. However, the same functions can be used for other types of experiments.

Look at the help for the pasillaBam package. It explains that the package provides access to the “bam” files from the “pasilla” experiment. There we have a little bit more information. The bam files come from, single-end reads which were mapped to the drosophila dm3 genome. The experiment was described in the “pasilla” data package. For our purposes, we just need to know that they were mapped to this genome, so the dm3 genome, which will load.

We also use a “transcript database object” which contains the information for all known transcripts annotated in the Drosophila Genome. There is one transcript database for a number of well-studied organisms. For instance there's also the Homo sapiens hd19 known gene transcript database`

*Sometimes we can't find the transcript database we are interested in, so the annotation of the genes you're interested in. In these cases you can use them makeTranscriptDbFromGFF file function. And this can be used to import a GFF or GTF file which contain, for every gene, the different transcripts and different exons which make up that gene.*

We can use the exonsBy function which will go into this transcript database and, for each gene, pull out the exons. For example, if we want to know the exons of the 100th gene we can access then as in the example code.

* Sub-setting with the single square brackets (grl[100]) got us a granges list.
* If we want to get ranges, we use the double square brackets (grl[[100]]) because we're indexing from a list. It yields a granges within the eight ranges, which are exons,
* and if we want the first exon we can subset with a single square bracket (grl[[100]][1]). And this gives us the first exon of the 100th gene

That is we can obtain either a list named as the gene, and containing the exon ranges, just the ranges or just the first exon’s range. The name of the 100th gene, is based on whatever order they were in the transcript database. These are the ranges that correspond to the eight different exons on chromosome 2. Note that **the exons can be overlapping each other**. So if you have variable ending points for an exon, but it overlaps with another, these will all be contained here.

The two functions in the pasillaBamSubset package just point us to bam files, which were downloaded when you installed the pasillaBamSubset library. This will look different on different computers or different operating systems but the general idea should be the same.

Now in order to count reads from bam files, we need to load two Bioconductor libraries.

* RSamtools is for reading reads from bam files and SAM files.
* And we also need the genomicranges package which was already loaded it, looks like`

If you're using Bioconductor version 14, the one which was released in early April, you should also load the genomicalignments package. ***If you're using an earlier version, so Bioconductor version 1s with R3.0 and then another number, you don't need this library because the functions we need were previously in the genomicranges package***. You only need to load this library in addition if you’re using Bioconductor version 14`

We specify a pointer to the bam files using a function from the SAM tools library, the BamfileList function.

In addition we can specify a yield size. So the yield size is *how many reads that we want to pull from each bam file at a time.* So if you have limited memory on your machine, you can use this to gauge how much memory should be taken up at a time. So this is saying we only want to read in 50,000 reads at a time because these files are actually subsets of bam files and they don't contain all of the reads`

*If you're using real bam files which contained all of the reads of a sequencing experiment, you might choose a yield size of 1 million reads. But this depends on how much memory you have on your machine and you have to gauge`*

We'll name to the two files first and second. So now here we have the bam file list of length 2 which has first experiment and the second experiment. And the following function-- which is either in the genomicranges package or now the genomicalignments package-- summarizeOverlaps will count the reads, which is the second argument, which fall into the features, which is the first argument`

We're specifying that we don't care about strand specificity. So we can allow minus strand reads to count towards plus strand genes and vice versa. Because the experiment was not strand specific, we want to count these reads as well`

There's a number of arguments to summarizeOverlaps.

* So in addition to ignore strand, if we look up the other arguments for summarizeOverlaps, there are three different counting modes. The default is union. But there's a linked image in the genomic ranges of genomic alignments package which displays what these mean. We won't go into it now`
* And also inter.feature is important. So inter.feature means that if a read overlaps multiple features, it should not be counted, which the default. So we're only interested in counting reads which uniquely align to one feature. If you set this to false then it will count these reads which align to multiple features`
* Another important argument is singleEnd, which says that these files have single-end reads instead of paired-end reads.
* The fragments argument is, if you're counting reads in a paired-end experiment, it's for specifying that you also want to count reads where only one of the two reads in a pair aligns. So you also want to count these reads`

But let's continue with our first summarizeOverlaps call. So now we have an object so1, which is a summarized experiment. The row data contains information about the features and the call data contains information about the files which we specified. So the count matrix, which we're interested in generating, is contained in the assays slot. If you just use the assay function, it just pulls the first matrix from the assay slot, which is which is called **counts**. So we use the assay function and get the very top of the count matrix. So we can confirm that the count matrix is 15,000 features long and M samples wide`

And if we ask for the sums of this count matrix we get the number of reads which aligned-- uniquely-- to these features. That yields 156,000 for the first sample and 122,000 reads for the second sample`

The row data is the information about the features, that is the genes. The call data is information about the samples`

Here it's empty but we could provide some information here by adding columns. If we name these samples *one* and *two*, now we have a new column which has information about the samples.

We also have a meta-data on the features. So if you ask for the meta-data, for the row data, it tells you information which is how this row data was generated. It says it was generated from a transcriptDB using the genomic features package. The data source was UCSC, the genome was dm3, the organism was drosophila, etcetera`

So this is pretty useful for reproducibility of count tables. We also have the stamp of the version numbers for all the packages that were used and the creation time`

So further, we can do some basic EDA on these counts. So let's store the first column of counts to the variable x. We can look at a histogram. So now we've already excluded those counts which are a count of zero. We can see that there's at least one very, very large count here. So more than 40,000. And if we exclude accounts over 10,000, we can see a bit more of the distribution`

So there's basically a dropping off. So we have a lot of very small counts, or counts between 0 and 1,000, and then kind of a long tail of higher counts. And if we call plot, we only have two columns o] assay matrix. So calling plot will do a scatter plot for us. And we've looked at the log of the x and y-axes so we can see that these two samples generally are highly correlated. That when one sample has Q high gene count, than the other sample also has hygiene count. And these counts here are the zeros because we added a pseudo count of one`

So the second file we should mention is actually a paired-end sequencing experiment. So we see here that they're paired-end reads. And we treated it previously as if it was a single-end experiment because single-end defaults in the summarizeOverlaps function to true. But if you were actually doing this generating this count matrix, you would want to count the reads in the second file only once. So you would only want to count each fragment towards the count matrix. So what we did previously was to count each read alone. But what we want to do is, if we have a pair of reads from the same fragment to only count that once. And to do this you need to do a separate call to summarizeOverlaps where you specify a singleEnd=FALSE. And we're also going to specify fragments=TRUE, which means if only one of the pair of reads maps, we also want to count this. So we'll just count the second file. And we'll run this second call to summarizeOverlaps. So now we can look at the number of reads which align this time. And we can note that the last time, when we count each read alone, we had about two times more reads. So 60,000 to 120,000`

We can also make a scatter plot of the reads from the first counting to the second counting. When we counted each read singly and we counted only pairs. And if you add an “y equals x” plot line and an “y equals 1/2 x” line, you can see essentially these counts are about half` because when we did the single-end counting, we were counting each read instead of each fragment`