Assignment 3: Variant consequence annotation

Arturo Torres Ortiz

2026-02-24

Functional annotation of variants involve predicting the potential functional impact of a genetic variant on a gene or protein. Common annotations include the variant consequence (synonymous, non-synonymous), the variant status (pathogenic, non-pathogenic, uncertain, etc), its location (intron, exon, TTS, etc.), aminoacid location and change, etc.

In this module, we will use 2 datasets:

1. A microbial dataset to find variants associated to antimicrobial resistance
2. A human dataset to annotate variants and understand their clinical consequence

Learning objectives

At the end of this week's assessment you will be able to:

1. Annotate variants
2. Understand the functional annotation of genetic variants
3. Scan for putative variants of interest given their functional annotation
4. Visualize variants and alignments

Dataset 1: Antimicrobial resistance in *Mycobacterium tuberculosis*

For this first dataset, we will analyze Illumina short-read sequencing data from a laboratory experimental evolution project. Experimental evolution or experimental mutagenesis studies aim to evolve microbial traits of interest in a controlled laboratory environment by growing populations over multiple generations under different conditions. Typically, the starting population will have a known phenotype (and often genotype), to study adaptation over generations in response to different selective preassures. For this experiment, our organism is *Mycobacterium tuberculosis*, and the phenotype evolved is antimicrobial resistance, especifically resistance to rifampicin, a common first line antibiotic that is essential for tuberculosis treatment.

The dataset contains two different lineages. Starting from a rifampicin susceptible strain, three different populations from each lieneage are grown in the presence of rifampicin. Two susceptible strains of each lineage are grown in parallel as control (CDC1551E and T85E). At the end of the growth period, all strains underwent short-read Illumina whole-genome sequencing.

Experimental evolution strain	M. tuberculosis lineage	Resistance
CDC1551E	Lineage 4	pan-susceptible
CDC06E	Lineage 4	rifampin-monoresistant
CDC47E	Lineage 4	rifampin-monoresistant
CDC07E	Lineage 4	rifampin-monoresistant
T85E	Lineage 2	pan-susceptible
T8501E	Lineage 2	rifampin-monoresistant
T8538E	Lineage 2	rifampin-monoresistant
T8505E	Lineage 2	rifampin-monoresistant

Using the strains grown in presence of rifampicin and those grown in standard culture as control, the main goal of this study is to detect mutations likely to be associated with rifampicin resistance.

More information about this data:

- Comas, I., Borrell, S., Roetzer, A. et al. Whole-genome sequencing of rifampicin-resistant Mycobacterium tuberculosis strains identifies compensatory mutations in RNA polymerase genes. Nat Genet 44, 106–110 (2012). https://doi.org/10.1038/ng.1038

Input and outputs

The input for this assessment are the fastq files for all sequenced strains.

Fastq files

`/project2/msalomon_1816/trgn_515/3_annotation/m_tuberculosis/fastq`

H37Rv hypervariable regions

`/project2/msalomon_1816/trgn_515/3_annotation/m_tuberculosis/reference/mtb.h37rv.hypervariabl`

The final output will be a list of variants likely to be associated with rifampicin resistance.

Required software

In order to complete the assessment, the following tools need to be installed:

```
module load bwa
module load htslib
module load bcftools
module load samtools
module load fastqc
```

SLURM arrays

For this assignment we are going to work with multiple samples, applying the same commands to each of them, so it's worth learning how to properly use SLURM arrays. Job arrays allow you to submit a single SLURM job script to launch many similar jobs. Since we will be running the same bioinformatic pipeline, array jobs are perfect for this task.

To submit an array job, you only need to add this line to the SLURM job `#SBATCH --array=N0-N1`, but changing N0 and N1 for your starting and ending number, respectively (eg: `#SBATCH --array=1-10` to run 10 jobs). The only thing that changes between each of the jobs in the array is the variable `SLURM_ARRAY_TASK_ID`, which takes the value of each iteration that you specified with `#SBATCH --array=1-10`. So in the first iteration, the variable `${SLURM_ARRAY_TASK_ID}` will have a value of 1, then of 2, 3 etc. So that variable is the only element that lets you change what happens in each iteration, so you need to use it in a smart way! We can use it to select specific files in a folder, or a specific line in a file. For instance:

```
#!/bin/bash

#SBATCH --account=msalomon_1385
#SBATCH --partition=main
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=5G
#SBATCH --time=1:00:00
#SBATCH --job-name=fastqc
#SBATCH --output=/scratch1/USER/temp/fastqc_%j.%a.out
#SBATCH --error=/scratch1/USER/temp/fastqc_%j.%a.err
#SBATCH --array=1-10

# Put the input and output directories into variables for conciseness
in_dir=/project2/msalomon_1816/trgn_515/3_annotation/m_tuberculosis/fastq
out_dir=/scratch1/USER/1_exp_evolution/fastq_qc

# First, let's select one file within a directory. In each iteration we will
 ↪  get a different file:
in_file=$(ls $in_dir/ | awk "NR==${SGE_TASK_ID}")

sample=$(basename $in_file .fastq.gz) # Set up a sample variable for the
 ↪  output.
fastqc -o $out_dir/$sample --extract --dir $out_dir $in_file
```

In this example, we are listing the files in our input directory using `ls`, then we are selecting each file depending on their order using `awk` and `${SGE_TASK_ID}`. So when `${SGE_TASK_ID}` is 1, we get the first file; when it's 2, we get the second file, etc.

```
# Now let's use a file with sample ids in each line of the file
sample_list=/scratch1/USER/1_exp_evolution/sample_list.txt
sample=$(awk "NR==${SGE_TASK_ID}" $sample_list)

in_file=$sample.fastq.gz # Set up the file name variable.
fastqc -o $out_dir/$sample --extract --dir $out_dir $in_file
```

In this case, we directly have a list of sample ids that we use the array job to loop through.

There are many ways to build array jobs, and the way to structure it will depend on your specific application. Just remember, everything comes down to how you use the `${SGE_TASK_ID}` variable.

A couple of other interesting variables in the array header are `%j` and `%a`. The variable `%j` stores the job id, and the variable `%a` has the array index (1,2,3,...n), so each of the stderr and stdout outputs will have their own name.

Short-read alignment

To run any read mapping software, we first need a reference genome. In the previous assignments, the CARD server already had multiple human reference genomes for us to use. For *M. tuberculosis*, we need to get it ourselves. The most common reference genome for tuberculosis is H37Rv. The assembly name for this reference genome is ASM19595v2.

All modern mappers require building an index to easily parse the reference genome during mapping. In fact, some of the biggest speed and memory improvements over early mappers was thanks to improved indexing methods. BWA, for instance, is based on the Burrows-wheeler Transform for powerful indexing to quickly locate where a read might align within the reference genome. The indexing for BWA can be done with the following command:

```
bwa index $ref_genome
```

That will create multiple files with the same name as your reference genome, but ending in different extensions. Those are the indexes. Keep those files with the same and location as your reference genome. If a software requires those indexes, you just need to show the location of the reference genome, and it will assume the name and location of all the indexes.

Task 1: Complete the mapping and variant calling pipeline for *M. tuberculosis*.

Note

1. Find the reference genome assembly ASM19595v2 in NCBI, download it, and fill in variable ref_genome in your mapping script `ref_genome=/path/to/reference`
2. Index the reference genome
3. Perform QC in the raw reads
4. Prepare a **SLURM array job** to perform the entire mapping pipeline:
    1. Quality control and filtering
    2. Mapping
    3. Marking duplicates
    4. Indel re-alignment
    5. Variant calling
    6. Variant soft-filtering
    7. Index the vcf file (`tabix -p vcf <input_file.vcf.gz>`)

Attach your SLURM array job script as the answer to this question **(20pts)**

Multi-sample VCF files

When comparing samples, we often use multi-sample VCF files. The file structure is the same as the single-sample files we have used in the previous modules, but now we will have multiple SAMPLE columns. This is useful for quickly comparing variants between samples, but it also save disk space and file number when dealing with many samples.

Task 2: Merge the VCF files from the T85 strain and the CDC strain.**

Note

Find the right `bcftools` tool to merge the 4 VCF files from each lineage, resulting in two VCF files with 4 samples each.
NOTE: Remember to use the right flag within the command to remove non-pass variants!

1. Paste your variant merging command **(5pts)**
2. Describe each column of the resulting VCF file. How does it differ from a single-sample VCF file? **(5pts)**
3. Should we use the INFO column or the FORMAT column to filter specific samples? Why? **(5pts)**

Filtering low complexity regions

Low complexity regions are very hard to map correctly, especially when using short-reads. For that reason, often we remove any variants that fall within known regions.

If you don't have a file with hypervariable regions, you can infer them! The most classic algorithm to do this is `DustMasker` from NCBI (https://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/app/dustmasker/). There are faster implementations of the same algorithm. I can recommend Heng Li's implementation `sdust`, which is available in the `minimap2` installation.

For this assignment, I have provided you with a file of hypervariable regions. Look at the path of file in the "Input and outputs" section.

There are many ways to remove variants that fall within specific genomic coordinates.

1. Hard filtering

We can use `bcftools` for this. We can use the flag `-T`. Using it on its own will keep regions listed in the file. Using it with the symbol ^ will remove regions listed in the file.

Sometimes, to use `bcftools` and quickly jump between lines, we need to first compress and index both the VCF and the BED files. This is useful when using flags like `-r` or `-R`:

```
# Compress and index the VCF
bgzip input.vcf
tabix -p vcf input.vcf.gz

# Compress and index the BED file
bgzip regions.bed
tabix -p bed regions.bed.gz
```

For the `-T` flag, bcftools actually reads the file line by line, so we don't need to index it.

We can then use the `-T` flag with the ^ symbol in front of the region file:

```
bcftools view -T ^regions.bed input.vcf.gz -O z -o hard_filtered.vcf.gz
```

Another quick option if you want to avoid compressing and indexing file is using `bedtools`:

```
bedtools intersect -header -v -a input.vcf -b regions.bed > hard_filtered.vcf
```

1. Soft filtering

This is a bit more complex than hard filtering, since we are adding a conditional tag to the FILTER column.

First, we need to devine what our new tag is in the VCF header:

```
echo '##FILTER=<ID=hypervar,Description="Variant falls within a hypervariable
 ↪  region">' > filter_header.txt
```

We can then use `bcftools annotate` to add that filter in the VCF file. We will use the `--mark-sites (-m)` flag to append the tag to the existing FILTER options. We will also use the `-h` flag to append the new FILTER header line to the existing header.

```
bcftools annotate -a regions.bed -h filter_header.txt -m '+hypervar'
 ↪  input.vcf.gz -O z -o soft_filtered.vcf.gz
```

Task 3: Removing variants that fall within hypervariable regions.

Note

1. Hard-filtered the variants from your multi-sample VCF that fall within the hypervariable regions in the .bed file. Attach your command **(5pts)**

Variant functional annotation

Functional annotation of variants involve predicting the potential functional impact of a genetic variant on a gene or protein. Common annotations include the variant consequence (synonymous, non-synonymous), it's location (intron, exon, TTS, etc.), aminoacid location and change, etc.

Variant functional consequence prediction typically requires an annotation file. This annotation file comes in a variety of formats and with varying information. The most used formats are GFF (General Feature Format) and GTF (Gene Transfer Format), where gene location and coordinates are the main source of information. These formats are pretty similar to the VCF format, and many fields will seem familiar.

Task 4: Variant annotation file

Note

1. Find in NCBI and download the annotation file in GFF format for ASM19595v2, and fill in variable gff_file **(5pts)** `gff_file=/path/to/gff_file`

These annotation files sometimes come unsorted. To ensure proper formatting of the GFF file and to index it, we can use genometools with the following code:

```
conda activate /project2/msalomon_1816/trgn_515/genometools

gt gff3 -sort -tidy -retainids input.gff | bgzip -c > sorted.gff.gz
tabix index -p gff sorted.gff.gz
```

Using genometools ensures proper formatting. GFF format stores genomic features (gene, mRNA, exon…). These features need to be ordered hierarchically, with parent features (e.g gene) preceding their child features (e.g mRNA, exon, CDS). Using genometools ensures that the GFF file is sorted by coordinate while respecting the hierarchical order.

We can also sort it using bedtools, but this won't strictly enforce the hierarchical order of genomic features:

```
bedtools sort -header -i input.gff | bgzip -c > sorted.gff.gz
```

We can also use simple bash scripting, but this command will remove the header, which may cause problems with some tools:

```
grep -v "#" $gff_file | sort -k1,1 -k4,4n -k5,5n -t$'\t' | bgzip -c >
 ↪  $gff_file.gz
tabix -p gff $gff_file.gz
```

There are many tools to perform variant annotation. Some of the most commonly used include `VEP` (Variant Effect Predictor, from Ensembl); `SnpEff`; `BCFtools csq`, or `ANNOVAR`. Ideally, we would like a simple software that takes a VCF file as an input, gives an annotated VCF file as output, and can be easily used with command pipes.

VEP

VEP is a very popular choice as it's one of the most comprehensive tools and it has a seamless integration with the Ensembl databases. It is, howver, one of the slowest softwares for variant annotation, so you should be careful when using it for large genomes (such as the human one).

```
vep # Main call to VEP
-i <vcf_file|stdin>
-o <vcf_file|stdout>
--gff <gff_file>
--fasta <reference_genome>
```

```
-distance <n> # Include consequences in genes n basepairs upstream and
 ↪  downstream. Useful for variants in regulatory regions or terciary
 ↪  structures
--vcf # Output in vcf format
--compress_output bgzip # Compress output
--no_stats # Do not report stats file
```

BCFtools csq

As most bcftools tools, `BCFtools csq` is memory efficient and incredibly fast. Additionally, unlike most other tools, `BCFtools csq` predicts consequences within a genotype. That means that if a variant corrects another variant dowstream, the consequence of both variants will be reverted.

```
bcftools csq
-f <ref_genome>
-g <gff_file>
-Oz
-o <vcf_file|stdout>
<vcf_file|stdin>
```

Unfortunately, `BCFtools csq` is very particular about the format of the GFF file, and it requires that the file adheres perfectly to the Ensembl format. That means that even files downloaded from NCBI may cause errors. For most Eukaryotic genomes, you can find GFF files on the Ensembl portal.

SnpEff

`SnpEff` is another widely used variant prediction tool. The performance is slightly better than `VEP`, but a lot worse than `BCFtools csq`. However, in some cases it offers more detailed variant prediction.

```
snpEff eff
<database_name>
<input_vcf_file|stdin>
```

The database name for M. tuberculosis H37Rv is Mycobacterium_tuberculosis_h37rv. If your genome is not present in their standard databases, you can build your own, but this process is quite cumbersome.

Task 5: Variant functional annotation

Note

1. Use one of VEP, bcftools, or SnpEff to annotate the variants from the multi-sample filtered VCF file **(5pts)**

Variant consequence

There can be many different types of variant consequences in our output. Some good resources to check what those consequences mean:

- https://useast.ensembl.org/info/genome/variation/prediction/predicted_data.html
- http://sequenceontology.org/browser/current_svn/term/

For easy visualizaiton of the results, it is a good idea to parse the VCF file into other formats.

The final data we need to finish the assignment is **gene name**, **variant consequence** and what **samples** that variant is present.

You can write a script to parse the VCF file and gather that information. For easy interpretation of results, let's assume any position with heterozygous variants is subjected to sequencing errors. A quick way to do this within `bcftools` would be `bcftools query`.

For instance (considering output from VEP):

```
paste <(bcftools query -f '%CHROM\t%POS\t%REF\t%ALT[\t%SAMPLE=%GT]\n'
 ↪  CDC_all.vcf.gz) <(bcftools query -f '%CSQ\n' CDC_all.vcf.gz | cut -d '|'
 ↪  -f4,2 | tr '|' '\t') | egrep -v "0/1" > CDC_results.txt

paste <(bcftools query -f '%CHROM\t%POS\t%REF\t%ALT[\t%SAMPLE=%GT]\n'
 ↪  T85_all.vcf.gz) <(bcftools query -f '%CSQ\n' T85_all.vcf.gz | cut -d '|'
 ↪  -f4,2 | tr '|' '\t') | egrep -v "0/1" > T85E_results.txt
```

We can also use the bcftools plugin `bcftools +split-vep`. There are many plugins to bcftools. These are usually for specific tasks outside of the main bcftools functionality.

To use these plugins, you need to call them using the + sign in front of the plugin name. You can get a list of all available plugins using:

```
bcftools plugin -l
```

We can use this plugin to select the features we want from the VCF file. For the VEP output:

```
bcftools view -e 'GT="0/1"' CDC_all.vcf.gz | \
bcftools +split-vep -d -f
↪ '%CHROM\t%POS\t%REF\t%ALT[\t%SAMPLE=%GT]\t%Consequence\t%Gene\n' >
↪ CDC_results.txt
```

You will have to adapt this to bcftools csq or snpeff.

You can use the `split-vep` plugin to get a list of the subfields so you can modify the previous command to adapt it to bcftools or snpeff:

```
# For SnpEff
bcftools +split-vep -a ANN -l CDC_all.vcf.gz

# For bcftools csq
bcftools +split-vep -a BCSQ -l CDC_all.vcf.gz
```

Task 6: Parse required information from VCF files

Note

Use any of the options listed above to parse the variant functional annotation from the VCF file.

1. Attach your parsing command to the assignment **(5pts)**

Task 7: Detect variants likely to be associated to rifampicin resistance**.

Note

Answer the following questions:

1. What criteria does a variant need to meet to be a good candidate for rifampicin resistance in our experiment? **(10pts)**
2. What are the genes with the highest number of variants? **(5pts)**
3. What are the genes with the highest number of non-synoymous variants? In how many samples are these mutations present? **(5pts)**
4. What are the genes with the highest proportion of non-synoymous variants to synonymous variants? **(5pts)**
5. Review the variants within those genes in IGV and determine if the alignment is reliable **(5pts)**
6. What antibiotic resistant mechanism from the ones reviewed in class explains the functional consequence of the variant? (i.e: target site modification, efflux pump, etc.)

(10pts)