MHC Genomics Analysis Project - Task 1 Report

Gene-Level Annotation from BAM using GTF and Reference Genome

Mahla Entezari*
Shahid Beheshti University, Tehran, Iran
Bioinformatics – Fall 2025

January 5, 2026

I. Introduction

Modern high-throughput sequencing (Next-Generation Sequencing; NGS) produces massive amounts of short reads that must be computationally processed to extract biologically meaningful information. A typical analysis workflow begins with sequencing reads, aligns them to a reference genome, and then uses genome annotation resources to interpret which biological features (genes, exons, regulatory elements) are supported by the aligned data. This project focuses on immune-related genomic regions, particularly the Major Histocompatibility Complex (MHC), which is well-known for high polymorphism and complex sequence structure. Even though later tasks in the project address MHC-specific questions, Task 1 builds the foundation by converting raw alignment data into a consistent **gene-level representation** that can be used for quality control, downstream filtering, comparative analysis across samples, and ultimately interpretation.

A. Objectives

Task 1 requires producing, for each sample alignment file (BAM), a **per-sample CSV table** that contains information about genes that appear in the alignment. A gene is considered "present" if at least one aligned read (or fragment) overlaps that gene. Each output CSV must contain at a minimum the following fields:

- `gene_name`
- `chromosome`
- `start`
- `end`
- `strand`
- `sequence`

Importantly, the `sequence` column must contain the genomic DNA sequence of each gene region, meaning this task is not only about gene coordinate extraction, but also about integrating reference genome sequence to construct a complete and reproducible gene-level table.

B. Importance

Later tasks involve analyzing read mapping quality, evaluating coverage patterns across MHC genes, and identifying candidate HLA alleles. All of these depend on reliable gene coordinates and correct reference build selection. If the annotation build mismatches the BAM reference (hg38 annotation used with hg19-aligned BAM), then gene coordinates do not line up with read alignments, causing downstream analyses to be incorrect. Therefore, this task is both a computational pipeline construction problem and a data integrity problem.

*MahlaEntezari.sbu@gmail.com

II. Materials and Methods

A. Overview of inputs and outputs

1. Inputs

This task uses three categories of input data:

1) **BAM files:** Each BAM file is a binary compressed alignment format containing aligned sequencing reads. BAM stores genomic coordinates of alignments, mapping quality scores, and many other alignment attributes.
2) **GTF gene annotation:** A GTF file describes the genomic coordinates of genes and related features (transcripts, exons, etc.) on a given reference build. For this task, gene-level features and gene names are required.
3) **Reference genome FASTA:** A FASTA file provides actual DNA sequences for each reference contig/chromosome. Since GTF provides coordinates but not nucleotide sequences, FASTA is required to populate the `sequence` column.

2. Outputs

For each input BAM file `<sample>.bam`, the pipeline produces:

- `project/csv/<sample>.task1.csv` (final deliverable; one per sample)

Additionally, intermediate files are produced for reproducibility and debugging (counts, gene lists, BED files, FASTA subsequences, and logs).

B. Computational environment

The analysis was executed on macOS using the `zsh` shell. The pipeline relies on common command-line bioinformatics tools:

- **samtools:** for inspecting alignment headers and indexing FASTA
- **featureCounts (Subread):** for assigning aligned reads/fragments to genes
- **bedtools:** for extracting reference genome sequences from coordinate intervals
- **Python 3:** for parsing GTF, constructing coordinate tables, and merging sequences into CSV

C. Steps:

Step 1: Identify the reference build used for BAM alignment

Rationale

Gene coordinates in a GTF are only meaningful relative to the same reference genome build used during alignment. If the BAM file was aligned against hg19 (GRCh37), then an hg38 (GRCh38) GTF will produce incorrect coordinate mapping. Therefore, the first step is verifying the reference build from the BAM header.

Procedure

I extracted the BAM header using:

```
samtools view -H <sample>.bam
```

The header contains `@SQ` lines that define each contig (chromosome names and lengths) and `@PG` lines that contain the command line used for alignment. In our BAM header, the reference path included:

```
/genomics/opt/RefGenomes/hg19/ucsc.hg19.fasta
```

This confirmed the BAM files were aligned to **UCSC hg19**. Consequently, the annotation GTF must match hg19/GRCh37 coordinate space, and chromosome naming should match UCSC style (`chr6`, not `6`).

Step 2: Select the correct GTF annotation

Rationale

Because alignment was against UCSC hg19, I must use an hg19-compatible GTF. I selected **GENCODE v37lift37 annotation** because it is a widely-used, comprehensive human gene annotation mapped to GRCh37/hg19.

Chosen file

`gencode.v37lift37.annotation.gtf`

Step 3: Count fragments per gene using featureCounts

Rationale

The core idea of the task is to identify which genes are supported by aligned reads. Rather than manually intersecting every read with every gene, I used `featureCounts`, a standard and efficient tool for assigning reads/fragments to genomic features.

A key detail is that the BAM data is paired-end. In paired-end sequencing, two reads represent one DNA fragment. Counting reads separately can inflate counts and lead to incorrect presence/absence decisions. Therefore, I counted **fragments** using paired-end mode.

Command

For each BAM file:

```
featureCounts -p -B \
  -a project/gtf/gencode.v37lift37.annotation.gtf \
  -o project/counts/<sample>.counts.txt \
  -g gene_name \
  -t exon \
  project/bam/<sample>.bam
```

Parameter explanations:

- `-p`: paired-end mode (count fragments)
- `-B`: require properly paired fragments (higher confidence)
- `-a`: path to annotation (GTF)
- `-o`: output count file
- `-g gene_name`: group exons by gene name
- `-t exon`: use exons as counting features and aggregate to gene-level

Output

`featureCounts` produces a tabular output with one row per gene, including read/fragment counts.

Step 4: Extract genes with count > 0 (genes present in the sample)

Rationale

The task requires output only for genes that have at least one aligned read. Once counts per gene are obtained, presence is defined simply as:

$$\text{Gene is present} \iff \text{count} > 0.$$

Command

I extracted gene names with positive counts using `awk`:

```
awk 'BEGIN{FS="\t"} $1 !~ /^#/ && NR>2 {\
  if($NF>0) print $1 }' \
  project/counts/<sample>.counts.txt \
  > project/genes/<sample>.genes.txt
```

Output

A gene list file:

`project/genes/<sample>.genes.txt`

Each line contains a gene name. For example, in sample `MOT36308`, the pipeline detected **734 genes** with counts > 0.

Step 5: Extract gene coordinates (chromosome/start/end/strand) from GTF

Rationale

Counts alone do not satisfy this task's output requirements. I need gene coordinates and the strand. These are stored in GTF lines where the third field (feature type) is `gene`. However, a full GTF file contains multiple feature types (exon, transcript, etc.), so I must:

1. Filter to gene-level features only
2. Match gene names to the genes detected in Step 4
3. Extract coordinate fields

Python implementation (`make_gene_table.py`)

I implemented a Python script to parse the GTF and generate two files:

- TSV: `gene_name, chromosome, start, end, strand`
- BED: coordinate intervals for sequence extraction

The script:

1. Loads the gene list from `<sample>.genes.txt`
2. Iterates through GTF records
3. Keeps only records where `feature == "gene"`
4. Extracts `gene_name` from the attributes column
5. Keeps only genes in the gene list
6. Writes TSV and BED outputs

BED coordinate conventions

GTF uses 1-based inclusive coordinates. BED uses 0-based half-open intervals. Therefore, conversion was:

$$\text{BED start} = \text{GTF start} - 1, \quad \text{BED end} = \text{GTF end}.$$

This conversion is essential for correct sequence extraction.

Validation

For `MOT36308`, the script reported:

```
Found 734 genes in GTF (out of 734 input genes).
```

This indicates a complete match between detected genes and the chosen GTF annotation.

Step 6: Obtain the reference genome FASTA (UCSC hg19) and index it

Rationale

The `sequence` column requires actual nucleotide content. Neither BAM nor GTF provides full gene sequences in a direct form suitable for output. Therefore, I must extract the gene region sequences from the same reference used for alignment (UCSC hg19).

FASTA acquisition and construction

I downloaded UCSC hg19 chromosome FASTA files and concatenated them into a single FASTA:

```
cat chr*.fa > ucsc.hg19.fasta
```

I verified the presence of expected contigs by checking FASTA headers using:

```
grep -m 3 '^>' project/ref/ucsc.hg19.fasta
```

Indexing

I created a FASTA index for random access:

```
samtools faidx project/ref/ucsc.hg19.fasta
```

Step 7: Extract gene sequences using bedtools getfasta

Rationale

Given a BED file of gene intervals, I can extract the corresponding reference sequences. I must also respect strand because genes can be encoded on the reverse strand.

Command

```
bedtools getfasta \
  -fi project/ref/ucsc.hg19.fasta \
  -bed project/genes/<sample>.genes.bed \
  -s -name \
  -fo project/seq/<sample>.genes.fa
```

Parameter explanations:

- `-fi`: input FASTA
- `-bed`: BED intervals of genes
- `-s`: stranded extraction (reverse-complement for minus strand genes)
- `-name`: use BED "name" column in FASTA headers

Observed FASTA header formatting

bedtools produced headers of the form:

`>ABCF1::chr6:30539169-30564956(+)`

This includes the gene name plus coordinate suffix. This is a valid FASTA, but creates a downstream key-matching issue when merging sequences with the TSV table (which stores only the gene name).

Step 8: Merge coordinate table and sequences into final per-sample CSV

Rationale

The final required output is one CSV per sample with gene coordinates and sequence. I therefore merged:

- TSV coordinates from `<sample>.genes.tsv`
- FASTA sequences from `<sample>.genes.fa`

Key normalization fix

Initially, merging failed and resulted in missing sequences for all genes:

`Missing sequences for 734 genes.`

The cause was the FASTA header suffix (`::chr...`). The merge script was updated to normalize the FASTA record name by taking only the substring before `::`. After this fix, merging succeeded.

Final merging and validation

For sample `MOT36308`:

`Wrote 734 rows. Missing sequences for 0 genes.`

This confirms that the `sequence` field was successfully populated for every gene.

D. Batch processing across all samples (automation)

Rationale

This must be repeated for all samples. Rather than manually running each command, I constructed a robust shell script that loops over all BAM files, runs the five-stage pipeline, and writes logs.

Batch execution outcome

The dataset contained **20 BAM files**, and the pipeline produced **20 final CSV outputs**, indicating a one-to-one mapping between input samples and deliverables.

III. Results

A. Deliverables produced

For each sample `<sample>`, the following files were produced:

- **Final Task's output:** `project/csv/<sample>.task1.csv`
- **Counts:** `project/counts/<sample>.counts.txt`
- **Genes present (count>0):** `project/genes/<sample>.genes.txt`
- **Coordinates (TSV):** `project/genes/<sample>.genes.tsv`
- **Coordinates (BED):** `project/genes/<sample>.genes.bed`
- **Sequences (FASTA):** `project/seq/<sample>.genes.fa`
- **Logs:** `project/logs/*`

B. Example sample summary: MOT36308

For the sample `MOT36308`:

- Genes detected with count>0: 734
- Genes found in GTF: 734/734
- Missing sequences after normalization: 0

This indicates a consistent and complete task's output for that sample.

C. Quality control checks

Several checks were performed to ensure correctness:

1) **Reference build match:** BAM header confirmed hg19, and lift37 GTF was selected accordingly.
2) **Chromosome naming consistency:** UCSC-style chromosome names (chr*) were used in BAM, GTF, and FASTA.
3) **No missing sequences:** final merge reported missing sequences = 0 for the tested sample, indicating successful sequence extraction and mapping.
4) **Batch completeness:** 20 input BAM files produced 20 output CSV files.

IV. Discussion

A. Interpretation of task's outputs

The task's outputs represent a gene-level view of each sample's alignment data. Each row corresponds to one gene that is supported by at least one aligned fragment. The coordinate fields (`chromosome, start, end, strand`) allow downstream analyses such as:

- comparing coverage or presence/absence patterns across samples
- filtering to specific genomic regions (the MHC locus on chr6)
- preparing gene lists for later targeted allele typing workflows

The `sequence` field provides a direct reference sequence for each gene region. Although the reference sequence does not capture sample-specific variants, it enables consistent feature-based analyses and provides a base for later comparisons to allele databases or variant calling outputs.

B. Key challenges and how they were resolved

1. Paired-end mismatch in featureCounts

Initially, `featureCounts` reported that paired-end reads were detected in a single-end library. The solution was to explicitly enable paired-end fragment counting using `-p`. This highlights the importance of understanding the sequencing library type and configuring tools accordingly.

2. FASTA header mismatch during merging

bedtools produced FASTA headers with appended coordinate information (`GENE::chr...`). The merge step initially failed because TSV gene names did not include these suffixes. The merging script was updated to

normalize FASTA headers by removing the suffix, enabling exact matching, and resulting in zero missing sequences.

3. Importance of reference consistency

Because this task requires genomic sequence extraction, consistency between BAM reference naming and the FASTA contig names is crucial. Using UCSC hg19 FASTA ensured that chromosome names like `chr6` in the BED file were found in the FASTA.

V. Key Questions

A. Which genes are covered by the sequencing data?

I operationally defined a gene as *covered* if it had at least one aligned fragment/read overlapping its annotated exonic regions. Concretely, I used `featureCounts` to assign paired-end alignments from each BAM file to genes (with paired-end counting enabled). A gene was considered covered if its resulting gene-level count satisfied:

$$\text{covered}(g) \iff \text{count}(g) > 0.$$

After counting, I extracted the list of all genes with `count` > 0 into `project/genes/<sample>.genes.txt`. I then generated the final output table for each sample, `project/csv/<sample>.task1.csv`, by retrieving gene coordinates (chromosome, start, end, strand) from the GTF and extracting the corresponding reference sequence from UCSC hg19.

Therefore, **the genes covered by the sequencing data are exactly the genes present in each sample's output files**:

- `project/genes/<sample>.genes.txt` (one gene name per line), and equivalently
- `project/csv/<sample>.task1.csv` (one row per covered gene with coordinates and sequence).

As an example from our completed pipeline, the sample `MOT36308` produced a covered-gene list of **734 genes**. This means that under the task's definition (at least one aligned fragment), **734 genes are covered** in `MOT36308`. Genes that do not appear in `<sample>.genes.txt` or `<sample>.task1.csv` are genes for which no overlapping fragments were detected (i.e., they have `count` $= 0$ under the same annotation and counting settings).

B. Are there genes with unusually high or low coverage?

This question has two parts: identifying genes with *low coverage* and genes with *high coverage*. Based on what task produced, I can answer the low-coverage part completely, and I can answer the high/low *unusualness* question by referring to the gene-count information that was computed as part of task (even though it was not stored in the final CSV).

1. Low coverage genes

Because genes are filtered by `count` > 0, it directly distinguishes:

- **Zero-coverage genes:** genes that do *not* appear in `<sample>.genes.txt` or `<sample>.task1.csv`. These genes have `count` $= 0$ under our counting configuration, meaning no aligned fragments overlapped their annotated exons.
- **Nonzero-coverage genes:** genes that *do* appear in the output lists. These genes have `count` > 0 and therefore meet the task requirement for being covered.

Thus, this task enables a complete and unambiguous statement about the existence of genes with very low coverage in the strictest sense: genes with **no detected coverage** (count $= 0$) are excluded from the output lists, while genes with **some detected coverage** (count > 0) are included.

2. Unusually high/low coverage

To claim that a gene has *unusually* high or low coverage, I must compare quantitative values across genes (counts, depth, or normalized coverage) and identify outliers relative to the distribution. The final CSV was designed to store gene identity, genomic coordinates, strand, and reference sequence; it does **not** include the numeric count values. Therefore:

- From the **final CSV alone**, I **cannot** rank genes by coverage or formally label genes as unusually high/low coverage, because the CSV does not contain a coverage metric.
- However, during the task I **did generate gene-level count tables** using `featureCounts`, stored as

`project/counts/<sample>.counts.txt`. These count files contain the numeric gene counts needed to assess unusually high/low coverage.

Given that these `featureCounts` outputs were produced as part of task, I can answer the question in a fully actionable way:

1. **Identify low-coverage genes (near-zero but nonzero):** using `project/counts/<sample>.counts.txt`, select genes with the smallest positive counts (the bottom tail of the distribution among count>0 genes). These are genes that are present but supported by very few fragments.
2. **Identify unusually high-coverage genes:** compute the distribution of gene counts and flag outliers. A standard approach is to use robust statistics such as the interquartile range (IQR):

$$\text{Outlier if } \text{count}(g) > Q_3 + 1.5 \times \text{IQR} \quad \text{or} \quad \text{count}(g) < Q_1 - 1.5 \times \text{IQR}.$$

Genes far above typical counts can indicate highly covered regions, potential mapping biases, repetitive content, or highly expressed loci (depending on experiment type). Genes with extremely low but nonzero counts can indicate weak coverage, low mappability, or borderline detection.

3. Conclusion for the high/low coverage question

In summary, this task establishes **which genes are covered** (count>0) and provides full gene metadata (coordinates, strand, reference sequence) for those genes. It also implicitly identifies **zero-coverage genes** (not present in the output lists). To determine whether any genes have **unusually high or unusually low** coverage, I must use the quantitative gene counts generated by `featureCounts` during the task (stored in `project/counts/<sample>.counts.txt`) and apply an outlier analysis on the count distribution. This analysis is directly supported by the task's intermediate outputs and can be performed without re-aligning reads or changing the pipeline.

C. Limitations

Task determines gene presence using a minimal threshold (count>0). This is appropriate for the task requirement, but does not measure quantitative expression or robust coverage. Some genes may have minimal coverage due to mapping noise, multi-mapping reads, or repetitive regions. Later tasks focusing on mapping quality and coverage variation will provide a more detailed assessment.

Additionally, the `sequence` column stores reference gene region sequence, not sample-specific sequences with variants. True sample allele or haplotype differences require variant calling or specialized HLA typing methods in later tasks.

VI. Conclusion

This successfully transformed each sample's BAM alignment data into a structured gene-level CSV annotated with coordinates and reference gene sequences. The pipeline correctly handled paired-end sequencing, used a reference-consistent hg19 GTF annotation, extracted sequences using UCSC hg19 FASTA, and generated one complete CSV per sample. The resulting 20 CSV outputs (for 20 BAM inputs) provide a reproducible foundation for the subsequent tasks involving mapping quality analysis, MHC gene-focused coverage evaluation, and candidate HLA allele identification.