UNIVERSIDAD DE MURCIA

Facultad de Biología
Universidad de Murcia

Bioinformatics against global warming:

Candidate mutation identification in low-chill spontaneous mutants for the climate adaptation of apricot

ANNEX

Tutors: José Antonio Campoy Corbalán &

Marcos Egea-Cortines

AUTHOR: Daniel González Palazón

INDEX

INTRODUCTION

This annex provides the Bash scripts used for the bioinformatic analysis in this Master's Thesis. Each script is presented with a brief description of its purpose, logic of the script, key commands, and input/output expectations.

For complete access and reproducibility, all scripts detailed in this annex are also publicly available in a dedicated GitHub repository:

https://github.com/DanielGP121/TFM_Bulida_Precoz_Repository

All scripts detailed in this annex are stored and executed from a dedicated scripts directory located at `/data/training2/analisis_TFM_Bulida_Precoz/TFM_Bulida_Precoz_Repository/` on the remote HPC computer (ladon, 172.18.15.151) at EEAD-CSIC where the TFM analysis is being conducted. Software installation and script execution are performed within a dedicated Conda environment named env_tfm_bulida, located at `/data/training2/softwares/miniconda3/envs/env_tfm_bulida`, ensuring a consistent and reproducible software environment. For each script execution, a log file is generated to capture the standard output and standard error streams, ensuring a record of the process and any potential issues. This is achieved by invoking the scripts using the nohup command to allow processes to continue running in the background even after session termination, with output redirected using the following general pattern:

```
nohup bash script_name.sh > script_name.log 2>&1 &.
```

1 Quality Control Scripts

run_full_pairwise_check.sh

- **Purpose:** The purpose of this script is to perform a comprehensive quality control check on the identity and relationship of all 8 experimental samples (4 mutant and 4 wild-type). By systematically comparing every possible pair of samples, this analysis serves two critical functions: first, to detect potential sample swaps or accidental duplications by identifying pairs with an anomalously low number of differing variants; and second, to validate the overall experimental design by confirming that biological replicates within a group (e.g., mt1 vs. mt2) are genetically more similar to each other than to samples from the opposing group (e.g., mt1 vs. wt1).
- **Logic:** The script automates the pairwise comparison process using a nested loop structure and the bcftools software suite.
    1. **Pairwise Iteration:** The script uses a nested loop to iterate through every unique pair of the 8 samples.
    2. **Temporary VCF Creation:** For each pair, it first calls bcftools view twice. Each call isolates a single sample's variant data from the main VCF and saves it to a new, temporary, correctly formatted (.vcf.gz) file. This is a robust method to prepare clean inputs for the intersection step.
    3. **Variant Intersection:** The script then uses bcftools isec on the two temporary VCF files. This command compares the files and generates a set of output files that contain the variants unique to the first sample, unique to the second sample, and those that are shared between both.
    4. **Robust Counting and Reporting:** To avoid errors when no variants are found in a given category, the script checks for the existence of each output file before using grep to count the records. The results (counts of unique and shared variants) for each pair are then appended to a master summary table.
    5. **Cleanup and Final Output:** After each comparison, all temporary files are removed. Once all pairs have been processed, the script prints the final, formatted summary table to the screen for easy review.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Performs a pairwise comparison of ALL samples (mutant and wild-
type).
# Handles cases where bcftools isec finds no shared variants and does not
create an output file.

set -euo pipefail

## CONFIGURATION ##
WORKSPACE_DIR="/data/training2/analisis_TFM_Bulida_Precoz/09_samples_double_ch
eck"
INPUT_VCF="${WORKSPACE_DIR}/final_bcftools_propDP_15pct.filtered.vcf.gz"
OUTPUT_DIR="${WORKSPACE_DIR}/pairwise_comparison_all"
# Create a dedicated directory for temporary files
TEMP_VCF_DIR="${OUTPUT_DIR}/temp_vcfs"
mkdir -p "${OUTPUT_DIR}"
mkdir -p "${TEMP_VCF_DIR}"

SAMPLES_TO_CHECK=("mt1" "mt2" "mt3" "mt4" "wt1" "wt2" "wt3" "wt4")
THREADS=8

## SCRIPT LOGIC ##
```

```
echo "--- Starting Full Pairwise Sample Identity Check for all 8 Samples ---"

SUMMARY_FILE="${OUTPUT_DIR}/full_pairwise_comparison_summary.txt"
echo -e
"Sample1\tSample2\tUnique_To_Sample1\tUnique_To_Sample2\tShared_Variants" >
"${SUMMARY_FILE}"

for (( i=0; i<${#SAMPLES_TO_CHECK[@]}; i++ )); do
    for (( j=i+1; j<${#SAMPLES_TO_CHECK[@]}; j++ )); do
        sample1=${SAMPLES_TO_CHECK[i]}
        sample2=${SAMPLES_TO_CHECK[j]}

        echo -e "\n--- Comparing: ${sample1} vs ${sample2} ---"

        VCF1="${TEMP_VCF_DIR}/${sample1}.vcf.gz"
        VCF2="${TEMP_VCF_DIR}/${sample2}.vcf.gz"

        # Step 1: Create temporary files for each sample
        bcftools view -s ${sample1} -c1 -Oz -o ${VCF1} ${INPUT_VCF}
        bcftools index ${VCF1}
        bcftools view -s ${sample2} -c1 -Oz -o ${VCF2} ${INPUT_VCF}
        bcftools index ${VCF2}

        # Step 2: Run isec on the temporary files
        PAIR_DIR="${OUTPUT_DIR}/temp_isec_${sample1}_vs_${sample2}"
        # Let bcftools isec generate its standard 4 output directories/files
        bcftools isec -p "${PAIR_DIR}" -c all ${VCF1} ${VCF2}

        # Step 3: Robustly count the results, assigning 0 if a file doesn't
exist
        FILE_UNIQUE1="${PAIR_DIR}/0000.vcf"
        FILE_UNIQUE2="${PAIR_DIR}/0001.vcf"
        FILE_SHARED="${PAIR_DIR}/0002.vcf" # This file contains records from
file1 also present in file2

        unique1=$( [ -f "${FILE_UNIQUE1}" ] && grep -vc '^#' "${FILE_UNIQUE1}"
|| echo 0 )
        unique2=$( [ -f "${FILE_UNIQUE2}" ] && grep -vc '^#' "${FILE_UNIQUE2}"
|| echo 0 )
        shared=$( [ -f "${FILE_SHARED}" ] && grep -vc '^#' "${FILE_SHARED}" ||
echo 0 )

        echo "Results: Unique to ${sample1}: ${unique1} | Unique to
${sample2}: ${unique2} | Shared: ${shared}"

        echo -e "${sample1}\t${sample2}\t${unique1}\t${unique2}\t${shared}" >>
"${SUMMARY_FILE}"

        # Clean up all temporary files for this pair
        rm -r "${PAIR_DIR}"
        rm ${VCF1}* ${VCF2}*
    done
done

# Final cleanup
rmdir "${TEMP_VCF_DIR}"
```

```
echo -e "\n### Full pairwise comparison complete! ###"
echo "Summary of results:"
cat "${SUMMARY_FILE}" | column -t
```

run_fastqc.sh

- **Purpose:** The purpose of this script is to perform a fundamental quality control (QC) analysis on all the raw sequencing read files (.fastq.gz and .fq.gz). This is a critical initial step in any sequencing analysis pipeline. It generates detailed reports that allow for the assessment of the raw data quality, helping to identify potential issues such as low-quality base scores, sequence duplication levels, or the presence of adapter contamination before proceeding with data processing and alignment.
- **Logic:** The script executes the fastqc program. It uses wildcards (*.fastq.gz, *.fq.gz) to provide a list of all raw read files located in the 00_raw_data directory as input. The script specifies two key parameters: --threads to enable parallel processing and speed up the analysis, and --outdir to direct all the generated output reports into a dedicated fastqc_reports folder for clean organization. For each input file, FastQC generates a comprehensive HTML report and a corresponding ZIP archive.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Runs FastQC on all raw read files to generate quality control
reports.

set -euo pipefail

## CONFIGURATION ##
BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
# Directory containing the raw sequencing data
RAW_DATA_DIR="${BASE_DIR}/00_raw_data"
# A new directory to store the FastQC reports for this analysis
OUTPUT_DIR="${BASE_DIR}/09_samples_double_check/fastqc_reports"
mkdir -p "${OUTPUT_DIR}"

# Number of threads to use for FastQC
THREADS=8

## SCRIPT LOGIC ##
echo "--- Starting FastQC Analysis for All Raw Samples ---"

# Check if the raw data directory exists
if [ ! -d "${RAW_DATA_DIR}" ]; then
    echo "ERROR: Raw data directory not found at ${RAW_DATA_DIR}"
    exit 1
fi

# Find all fastq files (both .fastq.gz and .fq.gz) and run FastQC on them
# The output reports will be placed in the specified OUTPUT_DIR
fastqc \
    --threads "${THREADS}" \
    --outdir "${OUTPUT_DIR}" \
    ${RAW_DATA_DIR}/*.fastq.gz \
    ${RAW_DATA_DIR}/*.fq.gz

echo -e "\n### FastQC analysis complete! ###"
echo "Reports have been generated in: ${OUTPUT_DIR}"
```

2 Initial Data Processing & Alignment

cp_rawdata_Illumina-Novogene.sh

- **Purpose:** This script is designed to copy all raw paired-end FASTQ files (with the .fq.gz extension) from the nested subdirectories of the Illumina-Novogene sequencing project into a centralized main raw data directory (00_raw_data). This step facilitates easier access and organization for subsequent data processing stages.

- **Logic:** The script first defines the source directory (RAWDIR) where the Novogene FASTQ files are located and the target directory (OUTDIR) for the TFM's raw data. It then performs a sanity check by printing these paths. The core logic involves a for loop that iterates through all files matching the pattern "$RAWDIR"/*/*.fq.gz. For each file found, the script prints a message indicating the copy operation and then uses the cp command to copy the file from its source location to the OUTDIR. A final message confirms the completion of the copy process. Shell options set -euo pipefail are used at the beginning to ensure robust error handling.

Script:

```
#!/usr/bin/env bash
# Exit immediately if a command exits with a non-zero status (-e),
# treat unset variables as an error (-u),
# and ensure any failure in a pipeline causes the whole pipeline to fail (-o
pipefail)
set -euo pipefail

# cp_rawdata_Illumina-Novogene.sh
# Copy all Illumina-Novogene FASTQ files into the working directory
#
# RAWDIR: location of the raw_data subfolders
# OUTDIR: destination for all .fq.gz files

RAWDIR="/data/training2/reads/seqs/Illumina-Novogene/X204SC20112838-Z01-
F001/raw_data"
OUTDIR="/data/training2/analisis_TFM_Bulida_Precoz/00_raw_data"

# Quick sanity-check: print script name and key variables
echo "===== DEBUG: verifying script and paths ====="
echo "Script: $0"
echo "RAWDIR: $RAWDIR"
echo "OUTDIR : $OUTDIR"
echo "============================================="

echo "Starting copy of raw Illumina-Novogene FASTQ files..."

for fichero in "$RAWDIR"/*/*.fq.gz; do
    echo "Copying: cp $fichero $OUTDIR/"
    cp "$fichero" "$OUTDIR/"
done

echo "Copy of Illumina-Novogene FASTQ files completed."
```

concat_reads.sh

- **Purpose:** This script is used to concatenate paired-end raw FASTQ files for specific samples (E, F, G, H) originating from different Max Planck sequencing projects or runs (specifically project "3649" and runs within project "4024"). Since these different files correspond to the same original biological sample, this step merges them into a single R1 file and a single R2 file for each sample, preparing them for consistent downstream processing.

- **Logic:** The script first defines the base directory for raw reads (RAWDIR) and the output directory (OUTDIR) for the concatenated files. It then enters a nested loop structure. The outer loop iterates through the specified sample prefixes (E, F, G, H), and the inner loop iterates through the read identifiers (R1, R2). Inside the loops, zcat is used to decompress and print the content of all matching FASTQ files (using wildcards to capture variations in filenames from project 3649 and different runs of project 4024) to standard output. This combined output is then piped (|) to gzip, which recompresses the data and redirects it to a new, combined FASTQ file (e.g., E_R1_combined.fastq.gz) in the OUTDIR. An echo command reports the sample and read being processed.

Script:

```
#!/usr/bin/env bash
# concat_reads.sh

# 1. Environment Variables
RAWDIR="/data/training2/reads/seqs"
OUTDIR="/data/training2/analisis_TFM_Bulida_Precoz/00_raw_data"

# 2. Concatenation loop with zcat -> gzip
for SAMPLE in E F G H; do
  for READ in R1 R2; do
    echo "Proccesing sample ${SAMPLE} read ${READ}"
    zcat \
      "${RAWDIR}/3649_*_${SAMPLE}_*_L008_${READ}_001.fastq.gz" \
      "${RAWDIR}/4024/run*/4024_*_${SAMPLE}_*_L*_${READ}_001.fastq.gz" \
    | gzip > "${OUTDIR}/${SAMPLE}_${READ}_combined.fastq.gz"
  done
done

echo "Concatenation completed"
```

concat_wt4.sh

- **Purpose:** This script is specifically designed to concatenate the paired-end FASTQ files for the Novogene sample initially identified as "A_2". This sample, which corresponds to the replicate wt4 in the project's naming scheme, had its sequencing data split across multiple files. The script merges these separate files into a single R1 FASTQ file (wt4_R1.fq.gz) and a single R2 FASTQ file (wt4_R2.fq.gz) within the 00_raw_data directory.
- **Logic:** The script defines the working directory (WORKDIR) where the individual A_2 FASTQ files are located and specifies the output filenames for the concatenated R1 and R2 reads. Debugging echo commands print the intended operations. The core logic uses zcat to decompress and concatenate all files matching the pattern "$WORKDIR"/A_2*_L2_1.fq.gz (for R1 reads) and pipes the output to gzip to recompress and save it as $OUT_R1. The same process is repeated for R2 reads using the pattern "$WORKDIR"/A_2*_L2_2.fq.gz and saving to $OUT_R2. Informational messages indicate the start and completion of the process.

Script:

```
#!/usr/bin/env bash
set -euo pipefail

# concat_wt4.sh
# Concatenate the two A_2 lane files (wt4) in 00_raw_data into single R1/R2 FASTQs

WORKDIR="/data/training2/analisis_TFM_Bulida_Precoz/00_raw_data"
OUT_R1="$WORKDIR/wt4_R1.fq.gz"
OUT_R2="$WORKDIR/wt4_R2.fq.gz"

# Debug: show what will be run
echo "=== DEBUG ==="
echo "Concatenating lane-1: $WORKDIR/A_2*_L2_1.fq.gz → $OUT_R1"
echo "Concatenating lane-2: $WORKDIR/A_2*_L2_2.fq.gz → $OUT_R2"
echo "============="

echo "Starting concatenation of wt4 (A_2) reads…"

# Lane-1 → R1
echo "zcat $WORKDIR/A_2*_L2_1.fq.gz | gzip > $OUT_R1"
zcat "$WORKDIR"/A_2*_L2_1.fq.gz | gzip > "$OUT_R1"

# Lane-2 → R2
echo "zcat $WORKDIR/A_2*_L2_2.fq.gz | gzip > $OUT_R2"
zcat "$WORKDIR"/A_2*_L2_2.fq.gz | gzip > "$OUT_R2"

echo "==== Concatenation of wt4 reads complete ===="
```

rename_reads.sh

- **Purpose:** This script standardizes the filenames of raw paired-end FASTQ files to a consistent project-specific naming scheme (e.g., wt1_R1.fastq.gz, mt1_R1.fastq.gz, etc.). It processes output files of initial data copying and any necessary concatenation steps (such as those for Max Planck samples E, F, G, H, and potentially parts of Novogene sample A_2 if they were pre-combined into the names listed as keys).

- **Logic:** The script defines a working directory (WORKDIR) where the source FASTQ files are located and where the renamed files will reside. The core of the script is a Bash array named MAP, that explicitly defines key-value pairs where each key is an original filename and its corresponding value is the new desired filename. The script then iterates through all the keys (original filenames) in the MAP array. For each original filename, it constructs the full source path (src) and destination path (dst). It checks if the source file exists, if the source file exists, the mv --verbose "$src" "$dst" command is used to rename it.

- **Script:**

```
#!/usr/bin/env bash
set -euo pipefail
# rename_reads.sh
# Rename reads according to the wt1…mt4 scheme
WORKDIR="/data/training2/analisis_TFM_Bulida_Precoz/00_raw_data" #

# Declare the mapping: original → new
declare -A MAP=(
  [E_R1_combined.fastq.gz]=wt1_R1.fastq.gz #
  [E_R2_combined.fastq.gz]=wt1_R2.fastq.gz #
  [F_R1_combined.fastq.gz]=wt2_R1.fastq.gz #
  [F_R2_combined.fastq.gz]=wt2_R2.fastq.gz #
  [G_R1_combined.fastq.gz]=mt1_R1.fastq.gz #
  [G_R2_combined.fastq.gz]=mt1_R2.fastq.gz #
  [H_R1_combined.fastq.gz]=mt2_R1.fastq.gz #
  [H_R2_combined.fastq.gz]=mt2_R2.fastq.gz #

  [A_1_BDPL200002386-1A_HKHF3DSXY_L2_1.fq.gz]=wt3_R1.fq.gz #
  [A_1_BDPL200002386-1A_HKHF3DSXY_L2_2.fq.gz]=wt3_R2.fq.gz #
  [A_2_BDPL200002387-1A_HKHF3DSXY_L2_1.fq.gz]=wt4_R1.fq.gz #
  [A_2_BDPL200002387-1A_HKHF3DSXY_L2_2.fq.gz]=wt4_R2.fq.gz #
  [A_3_BDPL200002388-1A_HKHF3DSXY_L2_1.fq.gz]=mt3_R1.fq.gz #
  [A_3_BDPL200002388-1A_HKHF3DSXY_L2_2.fq.gz]=mt3_R2.fq.gz #
  [A_4_BDPL200002389-1A_HKHF3DSXY_L2_1.fq.gz]=mt4_R1.fq.gz #
  [A_4_BDPL200002389-1A_HKHF3DSXY_L2_2.fq.gz]=mt4_R2.fq.gz #
)
echo "==== Renaming reads in $WORKDIR ====" #
for orig in "${!MAP[@]}"; do #
  src="$WORKDIR/$orig" #
  dst="$WORKDIR/${MAP[$orig]}" #
  if [[ -e "$src" ]]; then #
    mv --verbose "$src" "$dst" #
  else
    echo "Warning! $orig does not exist, skipping." #
  fi
done
echo "==== Renaming complete ====" #
```

trim_reads_skewer.sh

- **Purpose:** This script automates the process of trimming adapter sequences and low-quality bases from paired-end FASTQ files for all project samples (wt1-wt4, mt1-mt4). It uses the Skewer tool to perform these cleaning steps, preparing the reads for downstream alignment.

- **Logic:** The script defines essential configuration variables: RAWDIR for the location of input FASTQ files (renamed in a previous step), TRIMDIR for the output of trimmed files, ADAPTERS for the path to the adapter sequences FASTA file, and THREADS for the number of CPU cores to be used by Skewer. A "CHECK PREREQUISITES" section verifies the existence and accessibility of the skewer command, input/output directories, and the adapter file, exiting if any check fails. A "DEBUG INFO" section then prints the configured variables for verification. The core of the script is a for loop that iterates through each sample identifier (wt1 to mt4). Inside the loop:

    1. It constructs patterns to find all R1 and R2 FASTQ files for the current sample using wildcards (e.g., ${SAMPLE}_R1*.gz) and stores these in arrays (R1_FILES, R2_FILES).
    2. It checks if files were found for both R1 and R2; if not, it skips the current sample.
    3. The skewer command is then executed with specified parameters:
        - -r 0.1 (maximum error ratio for adapter match).
        - -d 0.05 (maximum indel error ratio for adapter match).
        - -k 8 (k-mer length for adapter seed detection).
        - -q 20 (minimum base quality for 3'-end trimming).
        - -l 75 (minimum read length to keep after trimming).
        - -m pe (paired-end mode).
        - -t "$THREADS" (number of threads).
        - -x "$ADAPTERS" (adapter FASTA file).
        - "${R1_FILES[@]}" and "${R2_FILES[@]}" (input read files).
        - -z (compress output with gzip).
        - -o "$TRIMDIR/$SAMPLE" (output prefix).
    4. A message indicates whether Skewer completed successfully or failed for the sample. Finally, a completion message is printed after processing all samples.

Script:

```
#!/usr/bin/env bash
set -euo pipefail
trap 'echo "Error on line $LINENO"; exit 1' ERR

# trim_reads_skewer.sh
# Trim paired-end FASTQ files using skewer for all samples wt1…mt4
# ── CONFIGURATION ──────────────────────────────────────────────────
RAWDIR="/data/training2/analisis_TFM_Bulida_Precoz/00_raw_data"
TRIMDIR="/data/training2/analisis_TFM_Bulida_Precoz/01_trimming"
ADAPTERS="/data/training2/info/shqMergedAdapters_Primers_representative_rc.fa"
THREADS=8
# ── CHECK PREREQUISITES ────────────────────────────────────────────
command -v skewer >/dev/null || { echo "ERROR: skewer not found in PATH"; exit 1; }
[[ -d "$RAWDIR" ]]    || { echo "ERROR: RAWDIR '$RAWDIR' does not exist"; exit 1; }
mkdir -p "$TRIMDIR"
```

```
[[ -w "$TRIMDIR" ]]   || { echo "ERROR: cannot write to TRIMDIR
'$TRIMDIR'"; exit 1; }
[[ -r "$ADAPTERS" ]]  || { echo "ERROR: adapter file '$ADAPTERS' not
readable"; exit 1; }
# ── DEBUG INFO
──────────────────────────────────────────────────────────────────────

echo "===== DEBUG: verifying paths & settings ====="
echo "Script   : $0"
echo "RAWDIR   : $RAWDIR"
echo "TRIMDIR  : $TRIMDIR"
echo "ADAPTERS : $ADAPTERS"
echo "THREADS  : $THREADS"
echo "============================================="
# ── TRIMMING LOOP
──────────────────────────────────────────────────────────────────

for SAMPLE in wt1 wt2 wt3 wt4 mt1 mt2 mt3 mt4; do
  echo "Trimming sample $SAMPLE"

  # Gather all matching R1/R2 files
  R1_FILES=( "$RAWDIR"/"${SAMPLE}"_R1*.gz )
  R2_FILES=( "$RAWDIR"/"${SAMPLE}"_R2*.gz )

  # Skip if none found
  if [[ ! -e "${R1_FILES[0]}" ]]; then
    echo "No R1 files for $SAMPLE, skipping."
    continue
  fi
  if [[ ! -e "${R2_FILES[0]}" ]]; then
    echo "No R2 files for $SAMPLE, skipping."
    continue
  fi
  echo " Found R1: ${R1_FILES[*]}"
  echo " Found R2: ${R2_FILES[*]}"
  # Run skewer
  if skewer \
      -r 0.1 \
      -d 0.05 \
      -k 8 \
      -q 20 \
      -l 75 \
      -m pe \
      -t "$THREADS" \
      -x "$ADAPTERS" \
      "${R1_FILES[@]}" \
      "${R2_FILES[@]}" \
      -z \
      -o "$TRIMDIR/$SAMPLE"
  then
    echo "$SAMPLE trimmed successfully"
  else
    echo "Skewer failed on $SAMPLE" >&2
  fi
done
echo "All samples processed."
```

run_alignment_pipeline.sh

- **Purpose:** This script automates the alignment of trimmed paired-end FASTQ reads to the reference genome using BWA-MEM. Following alignment, it performs several post-processing steps using SAMtools: conversion of SAM to BAM format, sorting by read name, fixing mate-pair information, sorting by coordinate, and finally, indexing the processed BAM file. The output is an analysis-ready, coordinate-sorted, indexed BAM file for each input sample.

- **Logic:** The script begins by setting shell options for error handling. It defines variables for input/output directories, the reference genome path, number of threads, and Read Group information.

    1. **Preparation Phase:**

        - It checks if bwa and samtools commands are available in the system's PATH.
        - Verifies the existence of the input directory containing trimmed reads and the reference genome file.
        - Creates the output alignment directory (ALIGNDIR) and a subdirectory for logs (LOGDIR) if they don't already exist, and checks for write permissions.
        - It then checks if the BWA index files for the reference genome exist. If not, it indexes the reference genome using bwa index "$REF_GENOME".

    2. **Processing Loop:** The script iterates through a predefined list of sample names (wt1 to mt4). For each SAMPLE:

        - Input (trimmed R1 and R2 FASTQ files) and output (intermediate and final SAM/BAM) filenames are defined.
        - It checks if the input FASTQ files for the current sample exist; if not, it skips to the next sample.
        - A Read Group string (RG_STRING) is constructed with sample-specific information (ID, SM, LB, PL).

    3. **Alignment:** bwa mem is executed with the specified threads (-t), the -M flag (to mark shorter split hits as secondary for GATK compatibility), and the constructed Read Group string (-R). The SAM output is redirected to $SAM_OUT, and standard error (BWA's progress/log) is redirected to a sample-specific error log file in $LOGDIR.

        - **SAM to BAM Conversion:** samtools view converts the SAM file to BAM format.
        - **Sort by Name:** samtools sort -n sorts the BAM file by read name, which is required for the fixmate step.
        - **Fix Mate Information:** samtools fixmate -m -O bam corrects mate-pair information in the BAM file and adds mate score tags (-m).
        - **Sort by Coordinate:** samtools sort sorts the fixmated BAM file by genomic coordinates, producing the main analysis-ready BAM file.
        - **Index BAM:** samtools index creates an index file (.bai) for the final coordinate-sorted BAM file, enabling fast random access. Throughout these steps, if any command fails, an error is reported, intermediate files generated for that specific sample up to that point are attempted to be removed, and the script continues to the next sample.

    4. **Cleanup:** After successfully generating and indexing the final BAM for a sample, intermediate files (.sam, .initial.bam, .sortedN.bam, .fixmate.bam) for that sample are removed to save disk space. The script concludes by printing a summary message indicating the completion of the process for all samples.

Script:

```
#!/usr/bin/env bash
set -euo pipefail
trap 'echo "ERROR: Script failed on line $LINENO with exit code $?"; exit 1' ERR

# run_alignment_pipeline.sh
# Aligns trimmed paired-end FASTQ files using BWA-MEM,
# then converts SAM to sorted BAM, fixmates, and indexes.

# ── CONFIGURATION ──────────────────────────────────────────────────────
TRIMDIR="/data/training2/analisis_TFM_Bulida_Precoz/01_trimming"
ALIGNDIR="/data/training2/analisis_TFM_Bulida_Precoz/02_alignment"
REF_GENOME="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta"
THREADS=8
PLATFORM="ILLUMINA"
LIB_PREFIX="lib"
LOGDIR="${ALIGNDIR}/logs" # Directory for bwa and script logs

# ── PREPARE ────────────────────────────────────────────────────────────
echo "INFO: Script started at $(date)"

if ! command -v bwa >/dev/null; then
  echo "ERROR: bwa command not found. Activate Conda environment or install."; exit 1
fi
if ! command -v samtools >/dev/null; then
  echo "ERROR: samtools command not found. Activate Conda environment or install."; exit 1
fi

if [[ ! -d "$TRIMDIR" ]]; then
  echo "ERROR: Trimmed reads directory '$TRIMDIR' does not exist"; exit 1
fi
if [[ ! -f "$REF_GENOME" ]]; then
  echo "ERROR: Reference genome file '$REF_GENOME' not found"; exit 1
fi

mkdir -p "$ALIGNDIR" "$LOGDIR"
if [[ ! -w "$ALIGNDIR" || ! -w "$LOGDIR" ]]; then
  echo "ERROR: Cannot write to output directory '$ALIGNDIR' or log directory '$LOGDIR'"; exit 1
fi

# Index the reference genome with BWA if index files are not present
if [[ ! -f "${REF_GENOME}.bwt" ]]; then
  echo "INFO: BWA index for reference genome '${REF_GENOME}' not found. Indexing now..."
  bwa index "$REF_GENOME"
  echo "INFO: BWA index created successfully."
else
  echo "INFO: BWA index for reference genome '${REF_GENOME}' already exists."
```

```
fi

echo "===== SCRIPT CONFIGURATION (run_alignment_pipeline.sh) ====="
echo "Trimmed Reads Dir    : $TRIMDIR"
echo "Alignment Output Dir: $ALIGNDIR"
echo "Logs Dir             : $LOGDIR"
echo "Reference Genome     : $REF_GENOME"
echo "Threads              : $THREADS"
echo "=========================================================="

#### ALIGNMENT AND BAM PROCESSING LOOP ####
for SAMPLE in wt1 wt2 wt3 wt4 mt1 mt2 mt3 mt4; do
  echo "--- Processing alignment for sample $SAMPLE ---"

  # Input trimmed FASTQ files
  R1_TRIMMED="$TRIMDIR/${SAMPLE}-trimmed-pair1.fastq.gz"
  R2_TRIMMED="$TRIMDIR/${SAMPLE}-trimmed-pair2.fastq.gz"

  # Intermediate and final file names for this sample
  SAM_OUT="$ALIGNDIR/${SAMPLE}.sam"
  BAM_INITIAL="$ALIGNDIR/${SAMPLE}.initial.bam" # Raw BAM from SAM
  BAM_SORTED_NAME="$ALIGNDIR/${SAMPLE}.sortedN.bam" # Sorted by name
  BAM_FIXMATE="$ALIGNDIR/${SAMPLE}.fixmate.bam" # After fixmate
  FINAL_BAM_FOR_DEDUP="$ALIGNDIR/${SAMPLE}.aligned.sorted.bam" # BAM ready for
MarkDuplicates

  # Check if input trimmed files exist
  if [[ ! -f "$R1_TRIMMED" || ! -f "$R2_TRIMMED" ]]; then
    echo "WARNING: Trimmed FASTQ files for sample $SAMPLE ('$R1_TRIMMED',
'$R2_TRIMMED') not found. Skipping."
    continue
  fi

  echo "Input R1: $R1_TRIMMED"
  echo "Input R2: $R2_TRIMMED"

  # Construct Read Group string (essential for GATK)
  RG_STRING="@RG\\tID:${SAMPLE}\\tSM:${SAMPLE}\\tLB:${LIB_PREFIX}_${SAMPLE}\\tP
L:${PLATFORM}"
  echo "Read Group for BWA: ${RG_STRING}"

  # 1. BWA-MEM Alignment
  echo "Step 1/6: Running BWA-MEM for $SAMPLE..."
  bwa mem -t "$THREADS" -M -R "$RG_STRING" "$REF_GENOME" "$R1_TRIMMED"
"$R2_TRIMMED" > "$SAM_OUT" 2> "${LOGDIR}/${SAMPLE}_bwa.err"
  if [ $? -eq 0 ]; then
    echo "SUCCESS: BWA-MEM completed."
  else
    echo "ERROR: BWA-MEM failed for $SAMPLE. Check
'${LOGDIR}/${SAMPLE}_bwa.err'."
    rm -f "$SAM_OUT" # Clean up potentially incomplete SAM
    continue
  fi

  # 2. Convert SAM to BAM
  echo "Step 2/6: Converting SAM to BAM..."
```

```
    samtools view -@ "$THREADS" -S -b "$SAM_OUT" > "$BAM_INITIAL"
    if [ $? -eq 0 ]; then
      echo "SUCCESS: SAM to BAM conversion completed."
    else
      echo "ERROR: samtools view (SAM to BAM) failed."
      rm -f "$SAM_OUT" "$BAM_INITIAL"
      continue
    fi

    # 3. Sort BAM by read name (required for samtools fixmate)
    echo "Step 3/6: Sorting BAM by name..."
    samtools sort -@ "$THREADS" -n "$BAM_INITIAL" -o "$BAM_SORTED_NAME"
    if [ $? -eq 0 ]; then
      echo "SUCCESS: BAM sorted by name."
    else
      echo "ERROR: samtools sort -n (by name) failed."
      rm -f "$SAM_OUT" "$BAM_INITIAL" "$BAM_SORTED_NAME"
      continue
    fi

    # 4. Fix mate pair information and add ms tag
    echo "Step 4/6: Running samtools fixmate..."
    samtools fixmate -@ "$THREADS" -m -O bam "$BAM_SORTED_NAME" "$BAM_FIXMATE"
    if [ $? -eq 0 ]; then
      echo "SUCCESS: samtools fixmate completed."
    else
      echo "ERROR: samtools fixmate failed."
      rm -f "$SAM_OUT" "$BAM_INITIAL" "$BAM_SORTED_NAME" "$BAM_FIXMATE"
      continue
    fi

    # 5. Sort BAM by coordinate (standard for downstream analysis)
    echo "Step 5/6: Sorting BAM by coordinate..."
    samtools sort -@ "$THREADS" "$BAM_FIXMATE" -o "$FINAL_BAM_FOR_DEDUP"
    if [ $? -eq 0 ]; then
      echo "SUCCESS: BAM sorted by coordinate."
    else
      echo "ERROR: samtools sort (by coordinate) failed."
      rm -f "$SAM_OUT" "$BAM_INITIAL" "$BAM_SORTED_NAME" "$BAM_FIXMATE"
  "$FINAL_BAM_FOR_DEDUP"
      continue
    fi

    # 6. Index the final sorted BAM (Picard MarkDuplicates also needs the index
  of the input)
    echo "Step 6/6: Indexing final BAM for $SAMPLE..."
    samtools index "$FINAL_BAM_FOR_DEDUP"
    if [ $? -eq 0 ]; then
      echo "SUCCESS: Final BAM indexed: ${FINAL_BAM_FOR_DEDUP}.bai"
    else
      echo "ERROR: samtools index failed."
      # Removing FINAL_BAM_FOR_DEDUP if indexing fails and it's critical
      rm -f "$SAM_OUT" "$BAM_INITIAL" "$BAM_SORTED_NAME" "$BAM_FIXMATE"
  "$FINAL_BAM_FOR_DEDUP"
      continue
    fi
```

```
    # Cleanup intermediate files for this sample
    echo "Cleaning up intermediate files for $SAMPLE..."
    rm -f "$SAM_OUT" "$BAM_INITIAL" "$BAM_SORTED_NAME" "$BAM_FIXMATE"
    echo "SUCCESS: Intermediate files cleaned up."

    echo "--- Finished alignment and BAM pre-processing for sample $SAMPLE ---"
done

echo "================================================================="
echo "Alignment and BAM pre-processing finished for all samples."
echo "Final BAM files (ready for deduplication) are in: $ALIGNDIR"
echo "BWA error logs are in: $LOGDIR"
echo "================================================================="
echo "INFO: Script finished at $(date)"
```

run_flagstat.sh

- **Purpose:** This script automates the generation of alignment statistics for all processed BAM files using the samtools flagstat command. These statistics provide a summary of mapping quality, such as the number of total reads, mapped reads, paired reads, and properly paired reads, which are essential for assessing the overall success of the alignment step.

- **Logic:** The script defines variables for the directory containing the input aligned BAM files (ALIGNDIR) and an output directory for the flagstat reports (FLAGSTAT_DIR).

    1. **Preparation Phase:**
        - It checks if the samtools command is available in the system's PATH.
        - Verifies the existence of the input ALIGNDIR.
        - Creates the FLAGSTAT_DIR if it doesn't already exist and checks for write permissions.
        - A "DEBUG" block prints the configuration variables for verification.
    2. **Processing Loop:** The script then iterates through a predefined list of sample names (wt1 to mt4). For each SAMPLE:
        - It constructs the full path to the input BAM file and the desired output file path for the statistics.
        - It checks if the input BAM file for the current sample exists; if not, it issues a warning and skips to the next sample.
        - The samtools flagstat "$INPUT_BAM" command is executed, and its standard output (which contains the statistics) is redirected to the OUTPUT_FLAGSTAT_FILE.
        - The script checks the exit status of the samtools flagstat command. If successful, it prints a success message; otherwise, it prints an error message and continues to the next sample. A final message indicates the completion of processing for all samples.

- **Script**:

```
#!/usr/bin/env bash
set -euo pipefail
trap 'echo "ERROR: Script failed on line $LINENO with exit code $?"; exit 1' ERR

# run_flagstat.sh
# Calculates alignment statistics using samtools flagstat for all processed BAM files.

# ── CONFIGURATION ──────────────────────────────────────────────────────────
ALIGNDIR="/data/training2/analisis_TFM_Bulida_Precoz/02_alignment"
FLAGSTAT_DIR="${ALIGNDIR}/flagstat_reports" # Directory to store flagstat reports

# ── PREPARE ────────────────────────────────────────────────────────────────
echo "INFO: Script started at $(date)"

if ! command -v samtools >/dev/null; then
  echo "ERROR: samtools command not found. Activate Conda environment or install."; exit 1
fi

if [[ ! -d "$ALIGNDIR" ]]; then
  echo "ERROR: Alignment directory '$ALIGNDIR' does not exist. Run alignment script first."; exit 1
```

```
fi

mkdir -p "$FLAGSTAT_DIR"
if [[ ! -w "$FLAGSTAT_DIR" ]]; then
  echo "ERROR: Cannot write to flagstat reports directory '$FLAGSTAT_DIR'";
exit 1
fi

echo "===== SCRIPT CONFIGURATION (run_flagstat.sh) ====="
echo "Aligned BAM Dir      : $ALIGNDIR"
echo "Flagstat Reports Dir : $FLAGSTAT_DIR"
echo "================================================"

#### FLAGSTAT LOOP ####
for SAMPLE in wt1 wt2 wt3 wt4 mt1 mt2 mt3 mt4; do
  echo "--- Calculating flagstat for sample $SAMPLE ---"

  INPUT_BAM="$ALIGNDIR/${SAMPLE}.aligned.sorted.bam"
  OUTPUT_FLAGSTAT_FILE="$FLAGSTAT_DIR/${SAMPLE}.flagstat.txt"
  # Check if input BAM file exists
  if [[ ! -f "$INPUT_BAM" ]]; then
    echo "WARNING: Aligned BAM file '$INPUT_BAM' not found for sample
$SAMPLE. Skipping."
    continue
  fi
  echo "Input BAM: $INPUT_BAM"
  echo "Output File: $OUTPUT_FLAGSTAT_FILE"

  # Run samtools flagstat and redirect output to a file
  echo "Running samtools flagstat for $SAMPLE..."
  samtools flagstat "$INPUT_BAM" > "$OUTPUT_FLAGSTAT_FILE"
  if [ $? -eq 0 ]; then
    echo "SUCCESS: samtools flagstat completed for $SAMPLE. Report saved to
$OUTPUT_FLAGSTAT_FILE"
  else
    echo "ERROR: samtools flagstat failed for $SAMPLE."
    continue
  fi

  echo "--- Finished flagstat for sample $SAMPLE ---"
done

echo "================================================"
echo "samtools flagstat processing finished for all samples."
echo "Reports are in: $FLAGSTAT_DIR"
echo "================================================"
echo "INFO: Script finished at $(date)"
```

concat_flagstats.sh

- **Purpose:** This script is designed to consolidate multiple individual samtools flagstat output files (each ending in .flagstat.txt) into a single, comprehensive summary file. This allows for easier review and comparison of alignment statistics across all processed samples.
- **Logic:** The script defines two variables: REPORTS_DIR, specifying the directory where the individual flagstat.txt reports are located, and OUTPUT_FILE, defining the name and path for the combined summary report. The core of the script is a for loop that iterates through all files within REPORTS_DIR matching the pattern *.flagstat.txt. For each file found ($f):

Script:

```
#!/usr/bin/env bash

#concat_flagstats.sh
# Script to concatenate flagstat reports
# Directory where the reports are located
REPORTS_DIR="/data/training2/analisis_TFM_Bulida_Precoz/02_alignment/flagstat_reports"
# Consolidated output file
OUTPUT_FILE="${REPORTS_DIR}/all_samples.flagstat_summary.txt"

echo "Concatenating .flagstat.txt files into ${OUTPUT_FILE}..."
# Iterate over all .flagstat.txt files in the specified directory
for f in "${REPORTS_DIR}"/*.flagstat.txt; do
  if [[ -f "$f" ]]; then
    echo "Processing: $f"
    # Add the filename as a header to the output file
    echo "=== Content of: $(basename "$f") ===" >> "$OUTPUT_FILE"
    # Add the content of the current file to the output file
    cat "$f" >> "$OUTPUT_FILE"
    # Add a blank line to separate sections
    echo -e "\n" >> "$OUTPUT_FILE"
  fi
done

echo "Concatenation complete. The summary is in: $OUTPUT_FILE"
```

run_mosdepth_coverage.sh

- **Purpose:** This script calculates the depth and breadth of sequencing coverage for each aligned BAM file using the mosdepth tool. Coverage statistics are essential for assessing the quality of the sequencing and alignment, ensuring sufficient data for reliable variant calling, and identifying potential biases or issues across the genome.

- **Logic:** The script defines configuration variables for the input directory containing aligned BAM files (ALIGNDIR), the base output directory for coverage statistics (COVERAGE_BASE_DIR), the path to the reference genome FASTA file (REF_GENOME_FASTA), and the number of threads (THREADS) for mosdepth.

    1. **Preparation Phase:**

        - It records the script start time and checks if mosdepth and samtools (needed for faidx) are accessible.
        - Verifies the existence of the input ALIGNDIR and the REF_GENOME_FASTA file.
        - It explicitly checks for the FASTA index file (.fai) corresponding to the reference genome. If the .fai file is not found, it attempts to create it using samtools faidx.
        - Creates the COVERAGE_BASE_DIR if it doesn't exist and checks for write permissions.
        - Prints a summary of the configuration settings.
    2. **Processing Loop:** The script iterates through the predefined list of sample names (wt1 to mt4). For each SAMPLE:
        - It constructs the full path to the input BAM file and defines an OUTPUT_PREFIX_PATH which mosdepth will use as a prefix for all its output files for that sample.
        - It checks if the input BAM file and its corresponding index (.bai) exist; if not, it issues a warning and skips to the next sample.
    3. **The mosdepth command is executed:**
        - -t "$THREADS": Specifies the number of threads for reading/writing BAM files.
        - -f "$REF_GENOME_FASTA": Provides the reference FASTA file to mosdepth. The tool will use this and its associated .fai index to determine contig names and lengths for generating per-chromosome/contig statistics.
        - "$OUTPUT_PREFIX_PATH": The prefix for all output.
        - "$INPUT_BAM": The input coordinate-sorted BAM file.
        - The script checks the exit status of mosdepth. If successful, it prints a success message; otherwise, it prints an error message and continues to the next sample. Finally, summary messages are printed indicating the completion of the process and the location of the output reports.

Script:

```
#!/usr/bin/env bash
set -euo pipefail
trap 'echo "ERROR: Script failed on line $LINENO with exit code $?"; exit 1' ERR

# run_mosdepth_coverage.sh
# Calculates depth of coverage using mosdepth for all aligned BAM files.

# ── CONFIGURATION ──────────────────────────────────────────────────────
ALIGNDIR="/data/training2/analisis_TFM_Bulida_Precoz/02_alignment"
COVERAGE_BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz/02_alignment/coverage_stats" # Base directory for all coverage outputs
REF_GENOME_FASTA="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta" # Reference FASTA file
THREADS=8 # Number of threads to use for mosdepth

# ── PREPARE ────────────────────────────────────────────────────────────
echo "INFO: Script started at $(date)"

if ! command -v mosdepth >/dev/null; then
  echo "ERROR: mosdepth command not found. Please ensure it is installed and your Conda environment is activated."
  exit 1
fi
if ! command -v samtools >/dev/null; then # samtools needed for faidx check
  echo "ERROR: samtools command not found. Please ensure it is installed."
  exit 1
fi

if [[ ! -d "$ALIGNDIR" ]]; then
  echo "ERROR: Alignment directory '$ALIGNDIR' does not exist. Run alignment script first."
  exit 1
fi
if [[ ! -f "$REF_GENOME_FASTA" ]]; then
  echo "ERROR: Reference genome FASTA file '$REF_GENOME_FASTA' not found."
  exit 1
fi

# Ensure the .fai index exists for the REF_GENOME_FASTA, as mosdepth with -f will look for it
REF_GENOME_FAI="${REF_GENOME_FASTA}.fai"
if [[ ! -f "$REF_GENOME_FAI" ]]; then
  echo "INFO: FASTA index '$REF_GENOME_FAI' not found. Attempting to create it with 'samtools faidx'..."
  samtools faidx "$REF_GENOME_FASTA"
  if [[ ! -f "$REF_GENOME_FAI" ]]; then
    echo "ERROR: Failed to create FASTA index '$REF_GENOME_FAI'. Please create it manually."
```

```
      exit 1
    fi
    echo "INFO: FASTA index '$REF_GENOME_FAI' created successfully."
  else
    echo "INFO: FASTA index '$REF_GENOME_FAI' already exists."
  fi

  mkdir -p "$COVERAGE_BASE_DIR"
  if [[ ! -w "$COVERAGE_BASE_DIR" ]]; then
    echo "ERROR: Cannot write to coverage base directory
  '$COVERAGE_BASE_DIR'"; exit 1
  fi

  echo "===== SCRIPT CONFIGURATION (run_mosdepth_coverage_alt.sh) ====="
  echo "Aligned BAM Dir          : $ALIGNDIR"
  echo "Coverage Stats Base Dir  : $COVERAGE_BASE_DIR"
  echo "Reference FASTA          : $REF_GENOME_FASTA (will look for .fai)"
  echo "Threads                  : $THREADS"
  echo "============================================================"

  #### MOSDEPTH LOOP ####
  for SAMPLE in wt1 wt2 wt3 wt4 mt1 mt2 mt3 mt4; do
    echo "--- Calculating coverage for sample $SAMPLE ---"

    INPUT_BAM="$ALIGNDIR/${SAMPLE}.aligned.sorted.bam"
    OUTPUT_PREFIX_PATH="${COVERAGE_BASE_DIR}/${SAMPLE}"

    if [[ ! -f "$INPUT_BAM" ]]; then
      echo "WARNING: Aligned BAM file '$INPUT_BAM' not found for sample
  $SAMPLE. Skipping."
      continue
    fi
    if [[ ! -f "${INPUT_BAM}.bai" && ! -f "${INPUT_BAM%.bam}.bai" ]]; then
      echo "WARNING: Index file for '$INPUT_BAM' not found. Please index
  it first. Skipping $SAMPLE."
      continue
    fi

    echo "Input BAM      : $INPUT_BAM"
    echo "Output Prefix  : $OUTPUT_PREFIX_PATH"

    # mosdepth with -f will look for a .fai index in the same location as
  the FASTA file.
    echo "Running mosdepth for $SAMPLE (using -f $REF_GENOME_FASTA)..."
    mosdepth -t "$THREADS" -f "$REF_GENOME_FASTA" "$OUTPUT_PREFIX_PATH"
  "$INPUT_BAM"

    if [ $? -eq 0 ]; then
      echo "SUCCESS: mosdepth completed for $SAMPLE using -f. Coverage
  reports saved with prefix $OUTPUT_PREFIX_PATH"
    else
      echo "ERROR: mosdepth failed for $SAMPLE using -f. (See error above
  or in nohup.err if used)"
      echo "This might still indicate an issue with the .fai file or the
  FASTA itself."
```

```
      continue # Skip to next sample if the primary method fails
    fi

    echo "--- Finished coverage calculation for sample $SAMPLE ---"
  done

  echo "============================================================"
  echo "Mosdepth coverage calculation finished for all samples."
  echo "Coverage reports are located in subdirectories within:
  $COVERAGE_BASE_DIR"
  echo "Key summary file for each sample: e.g.,
  ${COVERAGE_BASE_DIR}/wt1.mosdepth.summary.txt"
  echo "============================================================"
  echo "INFO: Script finished at $(date)"
```

concat_mosdepth_summary.sh.

- **Purpose:** This script is designed to consolidate multiple individual mosdepth summary reports (files ending with .mosdepth.summary.txt) into a single, comprehensive text file. This combined file allows for easier review and comparison of coverage statistics across all processed samples.
- **Logic:** The script defines two variables: REPORTS_DIR, specifying the directory where the individual mosdepth.summary.txt files are located, and OUTPUT_FILE, defining the name and path for the combined summary report. The core of the script is a for loop that iterates through all files within REPORTS_DIR matching the pattern *.mosdepth.summary.txt. For each file found ($f).

Script:

```
#!/usr/bin/env bash

# concat_mosdepth_summary.sh
# Script to concatenate mosdepth summary reports

# Directory where the mosdepth summary reports are located
REPORTS_DIR="/data/training2/analisis_TFM_Bulida_Precoz/02_alignment/coverage_stats"
# Consolidated output file
OUTPUT_FILE="${REPORTS_DIR}/all_samples.mosdepth_summary_combined.txt"

echo "Concatenating *.mosdepth.summary.txt files into ${OUTPUT_FILE}..."

# Iterate over all .mosdepth.summary.txt files in the specified directory
for f in "${REPORTS_DIR}"/*.mosdepth.summary.txt; do
  # Check if the file exists and is a regular file
  if [[ -f "$f" ]]; then
    echo "Processing: $f"
    # Add the filename as a header to the output file
    echo "=== Content of: $(basename "$f") ===" >> "$OUTPUT_FILE"
    # Add the content of the current file to the output file
    cat "$f" >> "$OUTPUT_FILE"
    # Add a couple of blank lines for better separation between file contents
    echo -e "\n\n" >> "$OUTPUT_FILE"
  fi
done

echo "Concatenation complete. The combined summary is in: $OUTPUT_FILE"
```

run_picard_markduplicates.sh

- **Purpose:** This script is responsible for identifying and marking PCR duplicate reads in the coordinate-sorted BAM files that were generated during the alignment stage. Marking duplicates is a crucial step before variant calling, as it prevents biases caused by overrepresentation of reads originating from the same DNA fragment. It uses the Picard MarkDuplicates tool.

- **Logic:** The script defines several configuration variables: ALIGNDIR (input directory containing aligned BAMs), DEDUPDIR (output directory for BAMs with duplicates marked), TMP_PICARD_DIR (a dedicated temporary directory for Picard's intermediate files), LOGDIR (for script execution logs), and MEM_GB (memory allocation for the Java Virtual Machine running Picard).

    1. **Preparation Phase:**

        - It records the script start time and checks if the picard command is accessible in the PATH.
        - Verifies the existence of the input alignment directory (ALIGNDIR).
        - Creates the output directory (DEDUPDIR), Picard's temporary directory (TMP_PICARD_DIR), and the log directory (LOGDIR) if they do not already exist, also checking for write permissions.
        - A summary of the configuration settings is printed for verification.
    2. **Processing Loop:** The script iterates through the predefined list of sample names (wt1 to mt4). For each SAMPLE:
        - It constructs the full paths for the input BAM file, the output BAM file with duplicates marked, and the output metrics file.
        - It checks if the input BAM file and its corresponding index (.bai) exist; if not, it issues a warning and skips to the next sample.
        - Before calling Picard, it sets and exports the _JAVA_OPTIONS environment variable. This variable is used to pass JVM arguments, specifically the maximum heap size and the Java temporary.
    3. **The picard MarkDuplicates command is executed**. Key parameters used are:
        - I="$INPUT_BAM": Input BAM file.
        - O="$OUTPUT_DEDUP_BAM": Output BAM file.
        - M="$METRICS_FILE": Output metrics file.
        - REMOVE_DUPLICATES=false: Duplicates are flagged in the BAM file but not physically removed, which is the recommended practice for GATK workflows.
        - VALIDATION_STRINGENCY=LENIENT: Allows processing to continue even with some minor BAM format inconsistencies.
        - TMP_DIR="$TMP_PICARD_DIR": Specifies the temporary directory for Picard.
        - MAX_FILE_HANDLES_FOR_READ_ENDS_MAP=1000: An optimization for handling large numbers of reads.
        - ASSUME_SORT_ORDER=coordinate: Informs Picard that the input BAM is already sorted by coordinate.
        - CREATE_INDEX=true: Instructs Picard to create an index for the output BAM file automatically.
        - The script checks the success or failure of the Picard command for the current sample.
        - After the Picard command, _JAVA_OPTIONS is unset to prevent it from affecting other Java applications. Finally, after processing all samples, summary messages are printed indicating completion and the location of output files.

Script:

```
#!/usr/bin/env bash
set -euo pipefail
trap 'echo "ERROR: Script failed on line $LINENO with exit code $?"; exit 1' ERR # Error trapping
# run_picard_markduplicates.sh
# Marks PCR duplicates in coordinate-sorted BAM files using Picard MarkDuplicates.

# ── CONFIGURATION ──────────────────────────────────────────────────────
ALIGNDIR="/data/training2/analisis_TFM_Bulida_Precoz/02_alignment"
DEDUPDIR="/data/training2/analisis_TFM_Bulida_Precoz/03_deduplication_picard"
TMP_PICARD_DIR="${DEDUPDIR}/tmp_picard"
LOGDIR="${DEDUPDIR}/logs"
MEM_GB=20 # Memory in GB for Java Virtual Machine

# ── PREPARATION ────────────────────────────────────────────────────────
echo "INFO: Picard MarkDuplicates script started at $(date)"

if ! command -v picard >/dev/null; then
  echo "ERROR: picard command not found. Please ensure Picard Tools is installed and in your PATH (activate Conda env if needed)."
  exit 1
fi

if [[ ! -d "$ALIGNDIR" ]]; then
  echo "ERROR: Input alignment directory '$ALIGNDIR' does not exist. Run Stage 1 script first."
  exit 1
fi

mkdir -p "$DEDUPDIR" "$TMP_PICARD_DIR" "$LOGDIR"
if [[ ! -w "$DEDUPDIR" || ! -w "$TMP_PICARD_DIR" || ! -w "$LOGDIR" ]]; then
  echo "ERROR: Cannot write to one or more output/temp/log directories for deduplication."
  exit 1
fi

echo "===== SCRIPT CONFIGURATION (run_picard_markduplicates.sh) ====="
echo "Aligned BAM Dir         : $ALIGNDIR"
echo "Deduplicated Output Dir : $DEDUPDIR"
echo "Picard Temp Dir         : $TMP_PICARD_DIR"
echo "Logs Dir                : $LOGDIR"
echo "Java Memory (GB)        : $MEM_GB"
echo "============================================================="

#### MARK DUPLICATES LOOP ####
for SAMPLE in wt1 wt2 wt3 wt4 mt1 mt2 mt3 mt4; do
  echo "--- Marking duplicates for sample: $SAMPLE ---"

  INPUT_BAM="$ALIGNDIR/${SAMPLE}.aligned.sorted.bam"
```

```
    OUTPUT_DEDUP_BAM="$DEDUPDIR/${SAMPLE}.dedup.bam"
    METRICS_FILE="$DEDUPDIR/${SAMPLE}.dedup_metrics.txt"

    if [[ ! -f "$INPUT_BAM" ]]; then
      echo "WARNING: Input BAM file '$INPUT_BAM' not found for sample
$SAMPLE. Skipping."
      continue
    fi
    if [[ ! -f "${INPUT_BAM}.bai" && ! -f "${INPUT_BAM%.bam}.bai" ]]; then
      echo "WARNING: Index file for '$INPUT_BAM' not found. Please ensure
it is indexed. Skipping $SAMPLE."
      continue
    fi

    echo "Input BAM         : $INPUT_BAM"
    echo "Output Marked BAM : $OUTPUT_DEDUP_BAM"
    echo "Metrics File      : $METRICS_FILE"

    echo "Setting Java options and launching Picard MarkDuplicates for
$SAMPLE..."

    # Set Java options using _JAVA_OPTIONS environment variable
    # This variable is often picked up by the JVM when launched via
wrapper scripts.
    export _JAVA_OPTIONS="-Xmx${MEM_GB}G -
Djava.io.tmpdir=${TMP_PICARD_DIR}"

    # Run Picard MarkDuplicates
    ( \
      picard MarkDuplicates \
      I="$INPUT_BAM" \
      O="$OUTPUT_DEDUP_BAM" \
      M="$METRICS_FILE" \
      REMOVE_DUPLICATES=false \
      VALIDATION_STRINGENCY=LENIENT \
      TMP_DIR="$TMP_PICARD_DIR" \
      MAX_FILE_HANDLES_FOR_READ_ENDS_MAP=1000 \
      ASSUME_SORT_ORDER=coordinate \
      CREATE_INDEX=true && \
    echo "SUCCESS: Picard MarkDuplicates completed for $SAMPLE." \
    ) || \
    { echo "ERROR: Picard MarkDuplicates failed for $SAMPLE. Check script
output/error logs."; }

    # Unset _JAVA_OPTIONS so it doesn't affect other Java applications
    unset _JAVA_OPTIONS
    echo "--- Finished marking duplicates for sample: $SAMPLE ---"
done

echo "============================================================"
echo "Duplicate marking finished for all applicable samples."
echo "BAM files with duplicates marked are in: $DEDUPDIR"
echo "============================================================"
echo "INFO: Script finished at $(date)"
```

concatenate_dedup_metrics.sh

- **Purpose:** This script is used to collect and combine the individual metrics files generated by Picard MarkDuplicates (those ending in .dedup_metrics.txt) into a single, consolidated text file. This aggregation facilitates easier review and comparison of duplication statistics across all processed samples.
- **Logic:** The script initiates by defining the source directory (METRICS_DIR) for the individual .dedup_metrics.txt files and the target OUTPUT_FILE for the consolidated report, ensuring the output file is cleared before use. It then iterates through all files matching the *.dedup_metrics.txt pattern within the METRICS_DIR. For each valid file found, the script appends a header indicating the source filename (using basename) and the entire content of that metrics file to the OUTPUT_FILE, followed by two blank lines for separation, and concludes with a completion message.

Script:

```
#!/usr/bin/env bash

# concatenate_dedup_metrics.sh
# Script to concatenate Picard MarkDuplicates metrics files

# Directory where the MarkDuplicates metrics files are located
METRICS_DIR="/data/training2/analisis_TFM_Bulida_Precoz/03_deduplication_picard"
# Consolidated output file
OUTPUT_FILE="${METRICS_DIR}/all_samples.dedup_metrics_combined.txt"

# Ensure the output file is empty at the start
> "$OUTPUT_FILE"

echo "Concatenating *.dedup_metrics.txt files into ${OUTPUT_FILE}..."

# Iterate over all .dedup_metrics.txt files in the specified directory
for f in "${METRICS_DIR}"/*.dedup_metrics.txt; do
  # Check if the file exists and is a regular file
  if [[ -f "$f" ]]; then
    echo "Processing: $f"
    # Add the filename as a header to the output file
    # $(basename "$f") extracts just the filename from the full path
    echo "=== Metrics from: $(basename "$f") ===" >> "$OUTPUT_FILE"
    # Add the content of the current file to the output file
    cat "$f" >> "$OUTPUT_FILE"
    # Add a couple of blank lines for better separation between file contents
    echo -e "\n\n" >> "$OUTPUT_FILE"
  fi
done

echo "Concatenation complete. The combined metrics are in: $OUTPUT_FILE"
```

3 Variant Calling

run_gatk_haplotypecaller.sh

- **Purpose:** This script automates the first step of the GATK variant calling workflow. It runs the GATK HaplotypeCaller tool on each processed and deduplicated BAM file. The script operates in "Genomic VCF" (GVCF) mode, which produces an intermediate file (.g.vcf.gz) for each sample. These GVCFs contain information about both variant sites and regions of the genome that are confidently homozygous-reference, making them suitable for a subsequent joint-genotyping step across all samples.

- **Logic:** The script begins by defining configuration variables for input/output directories, the reference genome path, and memory/thread settings for GATK.

1. **Preparation Phase:**
    - It verifies that the necessary commands (gatk, picard, samtools) are available in the system's PATH.
    - It checks for the existence of the input directory (containing deduplicated BAMs) and the reference genome FASTA file.
    - A critical step is verifying the presence of the reference genome's FASTA index (.fai) and sequence dictionary (.dict), as both are required by GATK. If either file is missing, the script attempts to generate it using samtools faidx and picard CreateSequenceDictionary, respectively.
    - It creates the main output directory for GVCFs, along with subdirectories for logs and temporary files used by GATK.
2. **Processing Loop:** The script iterates through a predefined list of sample names (wt1 to mt4). For each SAMPLE:
    - It defines the full paths for the input deduplicated BAM file and the output GVCF file.
    - It checks that the input BAM file and its corresponding index (.bai) exist; if not, it skips to the next sample.
    - The gatk HaplotypeCaller command is executed. Key parameters include:
        - --java-options: Sets the maximum memory (-Xmx) for the Java Virtual Machine and specifies a dedicated temporary directory.
        - -R: Specifies the reference genome FASTA file.
        - -I: Provides the input deduplicated BAM file.
        - -O: Defines the output GVCF file path.
        - -ERC GVCF: This crucial flag sets the tool to "Emit Reference Confidence" mode, generating a GVCF file.
        - --native-pair-hmm-threads: Specifies the number of CPU threads to use for the PairHMM algorithm, accelerating the computation.
    - The script includes error handling to report if the GATK command fails for any sample and then continues to the next.
3. **Conclusion:** After processing all samples, the script prints a final message indicating completion and suggesting the next step in the GATK workflow, which is typically joint-calling with GenotypeGVCFs.

Script:

```
#!/usr/bin/env bash
set -euo pipefail
trap 'echo "ERROR: Script failed on line $LINENO with exit code $?"; exit 1' ERR #
Error trapping

# run_gatk_haplotypecaller.sh
# Runs GATK HaplotypeCaller in GVCF mode for each deduplicated BAM file.

# ── CONFIGURATION
─────────────────────────────────────────────────────────

# Input directory: BAM files with duplicates marked
DEDUPDIR="/data/training2/analisis_TFM_Bulida_Precoz/03_deduplication_picard" #
# Output directory for GVCF files
VCFDIR="/data/training2/analisis_TFM_Bulida_Precoz/04_variant_calling_gatk"
# Reference Genome FASTA file
REF_GENOME="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta" #
# Reference Genome Dictionary (same base name as FASTA, with .dict extension)
REF_GENOME_DICT="${REF_GENOME%.fasta}.dict"
# Reference Genome FASTA Index
REF_GENOME_FAI="${REF_GENOME}.fai"

# GATK and Java options
# Memory for Java Virtual Machine (GATK); adjust based on server resources
MEM_MB=20480 # Approx 20GB (20 * 1024)
# Number of CPU threads for GATK HaplotypeCaller's implementation
THREADS_GATK=8

# Log directory for this script's output
LOGDIR="${VCFDIR}/logs"

# ── PREPARATION
──────────────────────────────────────────────────────────

echo "INFO: GATK HaplotypeCaller script started at $(date)"

# Verify necessary commands are available
if ! command -v gatk >/dev/null; then
  echo "ERROR: gatk command not found. Please ensure GATK is installed and your
Conda environment is activated."
  exit 1
fi
if ! command -v picard >/dev/null; then # Needed for CreateSequenceDictionary if
.dict is missing
    echo "WARNING: picard command not found. If reference dictionary is missing,
script may fail."
fi
if ! command -v samtools >/dev/null; then # Needed for faidx if .fai is missing
    echo "WARNING: samtools command not found. If reference .fai index is missing,
script may fail."
fi

# Check for input directory and reference files
if [[ ! -d "$DEDUPDIR" ]]; then
  echo "ERROR: Deduplicated BAM directory '$DEDUPDIR' does not exist. Run
MarkDuplicates script first."
  exit 1
fi
if [[ ! -f "$REF_GENOME" ]]; then
  echo "ERROR: Reference genome FASTA file '$REF_GENOME' not found."
```

```
  exit 1
fi

# Ensure .fai index exists
if [[ ! -f "$REF_GENOME_FAI" ]]; then
  echo "INFO: FASTA index '$REF_GENOME_FAI' not found. Attempting to create it
with 'samtools faidx'..."
  samtools faidx "$REF_GENOME"
  if [[ ! -f "$REF_GENOME_FAI" ]]; then
    echo "ERROR: Failed to create FASTA index '$REF_GENOME_FAI'. Please create it
manually."
    exit 1
  fi
  echo "INFO: FASTA index '$REF_GENOME_FAI' created successfully."
else
  echo "INFO: FASTA index '$REF_GENOME_FAI' already exists."
fi

# Ensure sequence dictionary exists (GATK HaplotypeCaller requires this)
if [[ ! -f "$REF_GENOME_DICT" ]]; then
  echo "INFO: Sequence dictionary '$REF_GENOME_DICT' not found. Attempting to
create it with Picard CreateSequenceDictionary..."
  if command -v picard >/dev/null; then
    picard CreateSequenceDictionary R="$REF_GENOME" O="$REF_GENOME_DICT"
    if [[ ! -f "$REF_GENOME_DICT" ]]; then
        echo "ERROR: Failed to create sequence dictionary '$REF_GENOME_DICT'.
Please create it manually."
        exit 1
    fi
    echo "INFO: Sequence dictionary '$REF_GENOME_DICT' created successfully."
  else
    echo "ERROR: picard command not found, cannot create sequence dictionary
'$REF_GENOME_DICT'. Please create it manually."
    exit 1
  fi
else
  echo "INFO: Sequence dictionary '$REF_GENOME_DICT' already exists."
fi

# Create output and log directories
mkdir -p "$VCFDIR" "$LOGDIR" "${VCFDIR}/tmp_gatk" # GATK temporary directory
if [[ ! -w "$VCFDIR" || ! -w "$LOGDIR" || ! -w "${VCFDIR}/tmp_gatk" ]]; then
  echo "ERROR: Cannot write to one or more output/temp/log directories for GATK."
  exit 1
fi

echo "===== SCRIPT CONFIGURATION (run_gatk_haplotypecaller.sh) ====="
echo "Deduplicated BAM Dir : $DEDUPDIR"
echo "GVCF Output Dir      : $VCFDIR"
echo "GATK Temp Dir        : ${VCFDIR}/tmp_gatk"
echo "Logs Dir             : $LOGDIR"
echo "Reference Genome     : $REF_GENOME"
echo "GATK Java Memory MB  : $MEM_MB"
echo "GATK HMM Threads     : $THREADS_GATK"
echo "============================================================="
echo # Blank line

#### HAPLOTYPECALLER LOOP ####
for SAMPLE in wt1 wt2 wt3 wt4 mt1 mt2 mt3 mt4; do
  echo "--- Running GATK HaplotypeCaller for sample: $SAMPLE ---"
```

```
  INPUT_DEDUP_BAM="$DEDUPDIR/${SAMPLE}.dedup.bam"
  OUTPUT_GVCF="$VCFDIR/${SAMPLE}.g.vcf.gz" # Outputting in GVCF format

  # Check if input deduplicated BAM file exists
  if [[ ! -f "$INPUT_DEDUP_BAM" ]]; then
    echo "WARNING: Deduplicated BAM file '$INPUT_DEDUP_BAM' not found for sample
$SAMPLE. Skipping."
    continue
  fi
  # GATK HaplotypeCaller also requires the index for the input BAM
  if [[ ! -f "${INPUT_DEDUP_BAM}.bai" && ! -f "${INPUT_DEDUP_BAM%.bam}.bai" ]];
then
    echo "WARNING: Index file for '$INPUT_DEDUP_BAM' not found. Please ensure it
is indexed. Skipping $SAMPLE."
    continue
  fi

  echo "Input Deduplicated BAM: $INPUT_DEDUP_BAM"
  echo "Output GVCF           : $OUTPUT_GVCF"

  echo "  Launching GATK HaplotypeCaller for $SAMPLE..."
  # Using a subshell to group command and error check
  (gatk --java-options "-Xmx${MEM_MB}m -Djava.io.tmpdir=${VCFDIR}/tmp_gatk"
HaplotypeCaller \
    -R "$REF_GENOME" \
    -I "$INPUT_DEDUP_BAM" \
    -O "$OUTPUT_GVCF" \
    -ERC GVCF \
    --native-pair-hmm-threads "$THREADS_GATK" && \
  echo "SUCCESS: GATK HaplotypeCaller completed for $SAMPLE." \
  ) || \
  { echo "ERROR: GATK HaplotypeCaller failed for $SAMPLE. Check script
output/error logs from nohup."; continue; }

  echo "--- Finished HaplotypeCaller for sample: $SAMPLE ---"
done

echo "============================================================"
echo "GATK HaplotypeCaller (GVCF mode) processing finished for all applicable
samples."
echo "GVCF files are located in: $VCFDIR"
echo "Next steps typically involve GenotypeGVCFs for joint calling if analyzing
multiple samples together,"
echo "or direct filtering of individual VCFs (after conversion from GVCF if
needed) for specific comparisons."
echo "============================================================"
echo "INFO: Script finished at $(date)"
```

run_mutect2_pon_creation.sh

- **Purpose:** This script creates a Panel of Normals (PoN) for use with the GATK Mutect2 somatic variant caller. A PoN is a crucial resource built from a cohort of normal (wild-type) samples. It catalogs recurrent technical artifacts from sequencing and processing, as well as common germline variants present in the population. By providing this PoN to Mutect2 during variant calling on the mutant samples, it can more accurately filter out these noisy sites, leading to a much cleaner and more reliable set of candidate somatic mutations.

- **Logic:** The script executes a multi-step workflow to generate the PoN from the four wild-type (wt1-wt4) samples.

  1. **Mutect2 in Tumor-Only Mode:** The script first runs gatk Mutect2 on each individual wild-type BAM file. Although this is a somatic caller, running it on normal samples in this "tumor-only" mode is the recommended first step for identifying potential germline variants and sequencing artifacts specific to each sample.
  2. **VCF Normalization:** The resulting single-sample VCFs are then normalized using gatk LeftAlignAndTrimVariants. This standardizes the representation of indels and splits complex events like Multi-Nucleotide Polymorphisms (MNPs) into their simplest forms, ensuring consistency before merging.
  3. **Variant Type Filtering:** The script uses gatk SelectVariants to keep only SNPs and INDELs, explicitly filtering out the MNPs that were split in the previous step.
  4. **Merge VCFs:** The four normalized and filtered VCFs (one for each wild-type sample) are merged into a single, multi-sample VCF using bcftools merge. This consolidated file contains all the variants found across the normal cohort.
  5. **Create Panel of Normals:** Finally, gatk CreateSomaticPanelOfNormals is run on the merged VCF. This tool aggregates the information from all normal samples to build the final PoN file (bulida_pon.vcf.gz), which is then ready to be used as a filter in subsequent Mutect2 runs on the mutant samples.

Script:

```
#!/usr/bin/env bash

# run_mutect2_pon_creation.sh

# Create a Panel of Normals (PoN) by merging filtered single-sample VCFs
# using bcftools, then calling CreateSomaticPanelOfNormals on the merged VCF.

set -euo pipefail
trap 'echo "ERROR on line $LINENO: exit code $?" >&2; exit 1' ERR

# --- CONFIGURATION ---
# Directory containing deduplicated BAMs
DEDUP_DIR="/data/training2/analisis_TFM_Bulida_Precoz/03_deduplication_picard"

# Main output directory for Mutect2 and PoN
PON_DIR="/data/training2/analisis_TFM_Bulida_Precoz/04_mutect2_pon"

# Reference FASTA
REF_GENOME="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta"

# Full path to GATK jar
GATK_JAR="/data/training2/softwares/conda_envs/env_tfm_bulida/share/gatk4-4.6.2.0-
0/gatk-package-4.6.2.0-local.jar"

# List of normal sample IDs (no extensions)
NORMAL_SAMPLES="wt1 wt2 wt3 wt4"
```

```
# JVM memory (MB) and threads for pair-HMM
MEM_MB=20480
THREADS=8

# --- PREPARE DIRECTORIES ---
mkdir -p \
  "${PON_DIR}/single_sample_vcfs" \
  "${PON_DIR}/normalized_vcfs" \
  "${PON_DIR}/filtered_vcfs"

# --- STEP 1: Run Mutect2 in tumor-only mode on each normal sample ---
echo "=== STEP 1: Mutect2 (tumor-only) on normals ==="
for SAMPLE in ${NORMAL_SAMPLES}; do
  BAM="${DEDUP_DIR}/${SAMPLE}.dedup.bam"
  RAW_VCF="${PON_DIR}/single_sample_vcfs/${SAMPLE}.single_sample.vcf.gz"
  if [[ ! -f "${BAM}" ]]; then
    echo "WARNING: BAM not found (${BAM}), skipping ${SAMPLE}"
    continue
  fi
  if [[ -f "${RAW_VCF}" ]]; then
    echo "INFO: VCF already exists (${RAW_VCF}), skipping"
    continue
  fi

  gatk --java-options "-Xmx${MEM_MB}m" Mutect2 \
    -R "${REF_GENOME}" \
    -I "${BAM}" \
    -O "${RAW_VCF}" \
    --native-pair-hmm-threads ${THREADS}
done

# --- STEP 2: Normalize VCFs and split MNPs ---
echo -e "\n=== STEP 2: Normalize & split MNPs ==="
for SAMPLE in ${NORMAL_SAMPLES}; do
  RAW_VCF="${PON_DIR}/single_sample_vcfs/${SAMPLE}.single_sample.vcf.gz"
  NORM_VCF="${PON_DIR}/normalized_vcfs/${SAMPLE}.normalized.vcf.gz"
  if [[ -f "${NORM_VCF}" ]]; then
    echo "INFO: Normalized VCF exists (${NORM_VCF}), skipping"
    continue
  fi

  gatk --java-options "-Xmx${MEM_MB}m" LeftAlignAndTrimVariants \
    -R "${REF_GENOME}" \
    -V "${RAW_VCF}" \
    -O "${NORM_VCF}" \
    --split-multi-allelics \
    --split-mnp

  gatk IndexFeatureFile -I "${NORM_VCF}"
done

# --- STEP 3: Filter out MNPs (keep only SNPs and INDELs) ---
echo -e "\n=== STEP 3: Filter out MNPs ==="
for SAMPLE in ${NORMAL_SAMPLES}; do
  NORM_VCF="${PON_DIR}/normalized_vcfs/${SAMPLE}.normalized.vcf.gz"
  FILT_VCF="${PON_DIR}/filtered_vcfs/${SAMPLE}.filtered.vcf.gz"
  if [[ -f "${FILT_VCF}" ]]; then
    echo "INFO: Filtered VCF exists (${FILT_VCF}), skipping"
    continue
  fi
```

```
  gatk --java-options "-Xmx${MEM_MB}m" SelectVariants \
    -R "${REF_GENOME}" \
    -V "${NORM_VCF}" \
    -O "${FILT_VCF}" \
    --select-type-to-include SNP \
    --select-type-to-include INDEL

  gatk IndexFeatureFile -I "${FILT_VCF}"
done

# --- STEP 4: Merge all filtered VCFs into a single multi-sample VCF using
bcftools ---
echo -e "\n=== STEP 4: bcftools merge ==="
MERGED_VCF="${PON_DIR}/all_normals_for_pon.vcf.gz"

if [[ ! -f "${MERGED_VCF}" ]]; then
  bcftools merge \
    $(for SAMPLE in ${NORMAL_SAMPLES}; do
        echo -n "${PON_DIR}/filtered_vcfs/${SAMPLE}.filtered.vcf.gz "
      done) \
    -Oz -o "${MERGED_VCF}" \
    --threads ${THREADS}

  tabix -p vcf "${MERGED_VCF}"
fi

# --- STEP 5: Create the Panel of Normals from the merged VCF ---
echo -e "\n=== STEP 5: CreateSomaticPanelOfNormals ==="
gatk --java-options "-Xmx${MEM_MB}m" CreateSomaticPanelOfNormals \
  -R "${REF_GENOME}" \
  -V "${MERGED_VCF}" \
  -O "${PON_DIR}/bulida_pon.vcf.gz"

echo -e "\nPanel of Normals successfully created
at:\n${PON_DIR}/bulida_pon.vcf.gz"
```

run_mutect2_pooled_wt.sh

- **Purpose:** This script performs somatic variant calling with GATK Mutect2 by comparing each mutant sample against a pooled normal control. Creating a pooled normal by merging all wild-type BAM files provides a robust baseline of the non-mutant genome, increasing the power to detect true somatic mutations. This script automates the entire process, from creating the pooled normal to calling and filtering variants for each mutant.

- **Logic:** The script is divided into two main stages.

  1. **Merge Wild-Type BAMs:** First, the script identifies all individual wild-type BAM files (wt1.dedup.bam to wt4.dedup.bam). It then uses samtools merge to combine them into a single, large BAM file (pooled_wild_type.dedup.bam). This pooled BAM is then indexed to prepare it for variant calling.

  2. **Somatic Calling and Filtering Loop:** The script then iterates through each mutant sample (mt1 to mt4). For each mutant, it performs a three-step GATK workflow:

     - **Mutect2:** It calls gatk Mutect2, providing the mutant BAM (-I), the pooled wild-type BAM (-I), the sample names of the normals (-normal), and the Panel of Normals (-pon) created by the previous script. This generates a raw VCF file containing all potential somatic variants.
     - **GetPileupSummaries:** This tool is run on the mutant BAM to calculate allele fractions at variant sites, which helps in estimating contamination.
     - **FilterMutectCalls:** The final step uses the raw VCF and the contamination estimates to apply a series of sophisticated filters, producing a high-confidence, filtered VCF file for each mutant sample.

Script

```
#!/usr/bin/env bash
set -euo pipefail
trap 'echo "ERROR: Script failed on line $LINENO with exit code $?"; exit 1'
ERR

# run_mutect2_pooled_wt.sh
# Merges all wild-type (wt) BAMs to create a pooled control sample,
# then runs GATK Mutect2 for each mutant sample against this pooled wild-type.
# CORRECTED to pass all normal sample names to Mutect2.

# ── CONFIGURATION
─────────────────────────────────────────────────────────────
DEDUPDIR="/data/training2/analisis_TFM_Bulida_Precoz/03_deduplication_picard"
MUTECT2_DIR="/data/training2/analisis_TFM_Bulida_Precoz/04_mutect2_calling"
REF_GENOME="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta"
PON_VCF="/data/training2/analisis_TFM_Bulida_Precoz/04_mutect2_pon/bulida_pon.
vcf.gz"

# GATK and Java options
MEM_MB=20480   # Approx 20GB [cite: User's Request]
THREADS=8      # Used for samtools merge [cite: User's Request]

# Wild-type and mutant sample lists
WILD_TYPE_SAMPLES="wt1 wt2 wt3 wt4"
MUTANT_SAMPLES="mt1 mt2 mt3 mt4"

# Define the name for the pooled wild-type BAM file
POOLED_WT_BAM="${DEDUPDIR}/pooled_wild_type.dedup.bam"
```

```
# ── PREPARATION ──────────────────────────────────────────────────────────
echo "INFO: GATK Mutect2 Pooled Wild-Type script started at $(date)"

# ... (Prerequisite checks for gatk, samtools, and directories would be here)
...

# ── STAGE 1: MERGE WILD-TYPE BAMS ────────────────────────────────────────
echo "--- Stage 1: Merging wild-type BAMs to create a pooled control sample --
-"

WT_BAM_LIST=()
for SAMPLE in $WILD_TYPE_SAMPLES; do
    WT_BAM_PATH="${DEDUPDIR}/${SAMPLE}.dedup.bam"
    if [[ -f "$WT_BAM_PATH" ]]; then
        WT_BAM_LIST+=("$WT_BAM_PATH")
    else
        echo "WARNING: Wild-type BAM not found for ${SAMPLE}, excluding." ]
    fi
done

if [ ${#WT_BAM_LIST[@]} -eq 0 ]; then
    echo "ERROR: No wild-type BAM files found to create a pool." #
    exit 1
fi

echo "  Merging into ${POOLED_WT_BAM}:"
printf "    %s\n" "${WT_BAM_LIST[@]}"

# Use samtools merge: output BAM first, then inputs. -r attaches RG from first
file. -h includes headers from all.
(samtools merge -@ $THREADS -r -h "${WT_BAM_LIST[0]}" -f "$POOLED_WT_BAM"
"${WT_BAM_LIST[@]}" && \
  echo "  SUCCESS: Merged wild-type BAMs." && \
  samtools index "$POOLED_WT_BAM" && \
  echo "  SUCCESS: Pooled wild-type BAM indexed." \
) || { echo "  ERROR: Failed to merge or index wild-type BAMs."; exit 1; }

# ── STAGE 2: MUTECT2 VARIANT CALLING & FILTERING ─────────────────────────
echo "--- Stage 2: Running Mutect2 for each mutant against the pooled wild-
type ---"
for MUTANT_SAMPLE in $MUTANT_SAMPLES; do
  echo
  echo "--- Processing mutant sample: $MUTANT_SAMPLE vs Pooled Wild-Type ---"

  MUTANT_BAM="${DEDUPDIR}/${MUTANT_SAMPLE}.dedup.bam"

  RAW_VCF_OUT="${MUTECT2_DIR}/raw_vcfs/${MUTANT_SAMPLE}_vs_pooled_wt.raw.vcf.g
z"
  FILTERED_VCF_OUT="${MUTECT2_DIR}/filtered_vcfs/${MUTANT_SAMPLE}_vs_pooled_wt
.filtered.vcf.gz"

  [[ -f "$MUTANT_BAM" ]] || { echo "WARNING: $MUTANT_BAM not found,
skipping."; continue; }

  # Build an array of normal sample arguments for Mutect2
```

```
  NORMAL_ARGS=()
  for WT_SAMPLE in $WILD_TYPE_SAMPLES; do
      NORMAL_ARGS+=("-normal" "$WT_SAMPLE")
  done

  # Step 1: Run Mutect2
  echo "  Step 1/3: Running Mutect2 for ${MUTANT_SAMPLE}..."
  (gatk --java-options "-Xmx${MEM_MB}m" Mutect2 \
    -R "$REF_GENOME" \
    -I "$MUTANT_BAM" \
    -I "$POOLED_WT_BAM" \
    "${NORMAL_ARGS[@]}" \
    -pon "$PON_VCF" \
    --f1r2-tar-gz "${MUTECT2_DIR}/stats/${MUTANT_SAMPLE}.f1r2.tar.gz" \
    -O "$RAW_VCF_OUT" && \
  echo "  SUCCESS: Mutect2 completed for ${MUTANT_SAMPLE}" \
  ) || { echo "  ERROR: Mutect2 failed for ${MUTANT_SAMPLE}."; continue; }

  # Step 2: GetPileupSummaries
  echo "  Step 2/3: Running GetPileupSummaries for ${MUTANT_SAMPLE}..."
  (gatk GetPileupSummaries \
    -I "$MUTANT_BAM" \
    -V "$RAW_VCF_OUT" \
    -L "$RAW_VCF_OUT" \
    -O "${MUTECT2_DIR}/stats/${MUTANT_SAMPLE}.pileups.table" && \
  echo "  SUCCESS: GetPileupSummaries completed for ${MUTANT_SAMPLE}." \
  ) || echo "  WARNING: GetPileupSummaries failed; contamination estimate may
be off."

  # Step 3: FilterMutectCalls
  echo "  Step 3/3: Running FilterMutectCalls for ${MUTANT_SAMPLE}..."
  (gatk FilterMutectCalls \
    -V "$RAW_VCF_OUT" \
    --contamination-table
"${MUTECT2_DIR}/stats/${MUTANT_SAMPLE}.pileups.table" \
    -O "$FILTERED_VCF_OUT" && \
  echo "  SUCCESS: FilterMutectCalls completed for ${MUTANT_SAMPLE}" \
  ) || { echo "  ERROR: FilterMutectCalls failed for ${MUTANT_SAMPLE}.";
continue; }

done

echo "============================================================"
echo "Mutect2 calling and filtering finished for all mutant samples."
echo "Filtered VCFs in: ${MUTECT2_DIR}/filtered_vcfs/"
echo "Script finished at $(date)"
```

01_prepare_references.sh

- **Purpose:** This script performs the essential preparatory steps on the reference genome FASTA file (BUL_cur_guided.v1.0.fasta) to make it compatible with downstream analysis tools, particularly the Genome Analysis Toolkit (GATK). Generating these standard index and dictionary files is a mandatory prerequisite for almost all subsequent alignment processing and variant calling stages.

- **Logic:** The script automates three sequential actions, checking for the existence of each output file before creation to prevent redundant work.

  1. **FASTA Index Creation:** It first calls samtools faidx to create a FASTA index file (.fai). This index allows programs to fetch sequences from specific genomic coordinates without reading the entire genome file into memory.
  2. **Sequence Dictionary Generation:** Next, it uses gatk CreateSequenceDictionary to generate a sequence dictionary (.dict). This file contains the names, lengths, and order of the contigs in the reference genome, a format required by GATK and Picard tools.
  3. **Interval List Creation:** Finally, it processes the .fai index with the cut command to produce a simple text file (.list) containing the name of each contig. This list is used by GenomicsDBImport in a later step to specify which genomic intervals to process.

Script:

```
#!/usr/bin/env bash
# STEP 1: Prepare reference genome files (indices and interval list).

set -euo pipefail

## CONFIGURATION ##
BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
REF_GENOME="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta"
OUTPUT_DIR="${BASE_DIR}/05_variant_filtering"
mkdir -p "${OUTPUT_DIR}"
INTERVALS_FILE="${OUTPUT_DIR}/all_contigs.list"
REF_DICT="${REF_GENOME%.fasta}.dict"

echo "--- PREPARATION: Verifying reference files ---"

# Check for FASTA index (.fai)
if [ ! -f "${REF_GENOME}.fai" ]; then
    echo "Reference FASTA index (.fai) not found. Creating it..."
    samtools faidx "${REF_GENOME}"
else
    echo "FASTA index (.fai) already exists."
fi

# Check for sequence dictionary (.dict)
if [ ! -f "${REF_DICT}" ]; then
    echo "Reference sequence dictionary (.dict) not found. Creating it..."
    gatk CreateSequenceDictionary -R "${REF_GENOME}" -O "${REF_DICT}"
else
    echo "Sequence dictionary (.dict) already exists."
fi

# Create interval list file
if [ ! -f "${INTERVALS_FILE}" ]; then
    echo "Creating interval file: ${INTERVALS_FILE}"
    cut -f1 "${REF_GENOME}.fai" > "${INTERVALS_FILE}"
else
```

```
    echo "Interval file already exists."
fi

echo "Preparation complete. All reference files are ready."
```

02_run_joint_calling.sh

- **Purpose:** The purpose of this script is to execute the GATK best-practices workflow for joint-calling on a cohort of samples. It takes the individual per-sample GVCF files previously generated by HaplotypeCaller and consolidates them to produce a single, multi-sample VCF file. This joint-calling approach increases statistical power and allows for a more accurate genotyping of all 8 samples (4 mutant, 4 wild-type) simultaneously, providing a comprehensive view of all variant sites across the entire dataset.

- **Logic:** The script performs the joint-calling process in the two main stages recommended by GATK, after first defining the necessary file paths and sample lists.

    1. **Data Consolidation with GenomicsDBImport**: The script first calls gatk GenomicsDBImport. This tool takes the list of all 8 individual GVCF files (-V) and the interval list (-L) created by the previous script. It efficiently merges the data from all these files into a specialized, single GenomicsDB database directory. This step is designed to be highly scalable for large numbers of samples.

    2. **Joint Genotyping with GenotypeGVCFs**: Once the database is created, the script calls gatk GenotypeGVCFs. This tool takes the GenomicsDB workspace as its single input (-V "gendb://...") and performs a joint analysis across all samples stored within. It calculates the final genotypes for every sample at every variant site and produces the final output: a single, multi-sample VCF file ready for quality filtering.

The entire process is wrapped in a conditional check to see if the final output VCF already exists, preventing the re-execution of this computationally intensive step.

Script:

```
#!/usr/bin/env bash
# STEP 2: Perform joint-calling using GenomicsDBImport and GenotypeGVCFs.

set -euo pipefail

## CONFIGURATION ##
BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
GVCF_DIR="${BASE_DIR}/04_variant_calling_gatk"
REF_GENOME="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta"
OUTPUT_DIR="${BASE_DIR}/05_variant_filtering"

DEFAULT_THREADS=8
DEFAULT_MEM_GB=30
THREADS=${1:-$DEFAULT_THREADS}
MEM_GB=${2:-$DEFAULT_MEM_GB}
GATK_JAVA_OPTS="--java-options -Xmx${MEM_GB}g"

GENOMICSDB_PATH="${OUTPUT_DIR}/genomicsdb_workspace_8_samples"
INTERVALS_FILE="${OUTPUT_DIR}/all_contigs.list"
JOINT_VCF="${OUTPUT_DIR}/all_samples_8.joint.vcf.gz"

WT_SAMPLES=("wt1" "wt2" "wt3" "wt4")
MT_SAMPLES=("mt1" "mt2" "mt3" "mt4")
ALL_SAMPLES=("${WT_SAMPLES[@]}" "${MT_SAMPLES[@]}")

echo "--- STEP 2: Performing Joint-Calling (using ${THREADS} threads,
${MEM_GB}GB RAM) ---"

if [ ! -f "${JOINT_VCF}" ]; then
    echo "STEP 2A: Importing GVCFs into GenomicsDB..."
```

```
    if [ -d "${GENOMICSDB_PATH}" ]; then
        echo "Warning: Removing existing GenomicsDB to start fresh."
        rm -rf "${GENOMICSDB_PATH}"
    fi

    GATK_GVCF_ARGS=()
    for SAMPLE in "${ALL_SAMPLES[@]}"; do
        GATK_GVCF_ARGS+=("-V" "${GVCF_DIR}/${SAMPLE}.g.vcf.gz")
    done

    gatk GenomicsDBImport \
      ${GATK_JAVA_OPTS} \
      "${GATK_GVCF_ARGS[@]}" \
      --genomicsdb-workspace-path "${GENOMICSDB_PATH}" \
      -L "${INTERVALS_FILE}" --reader-threads "${THREADS}" --overwrite-
existing-genomicsdb-workspace

    echo "STEP 2B: Running GenotypeGVCFs..."
    gatk GenotypeGVCFs \
      ${GATK_JAVA_OPTS} \
      -R "${REF_GENOME}" \
      -V "gendb://${GENOMICSDB_PATH}" \
      -O "${JOINT_VCF}"

    echo "Joint-calling complete. Output: ${JOINT_VCF}"
else
    echo "Joint VCF file already exists. Skipping."
fi
```

05a_build_snpeff_db.sh

- **Purpose:** This script automates the entire process of building a custom SnpEff database for the project's specific reference genome (BUL_cur_guided.v1.0.fasta). Creating this database is a mandatory, one-time setup step required before functional annotation can be performed. The script ensures that SnpEff recognizes the custom genome and has access to its corresponding gene annotation data to predict the effects of variants.

- **Logic:** The script performs three sequential stages to prepare the environment and build the database.

  1. **Directory and File Setup:** First, the script creates the required directory structure within the main SnpEff data folder (e.g., data/bulida_v1/). It then copies the project's reference FASTA and GFF3 annotation files into this new directory, renaming them to the standard filenames that SnpEff expects: sequences.fa and genes.gff, respectively.
  2. **Configuration File Modification:** The script then programmatically checks the main snpEff.config file. If an entry for the new genome (bulida_v1) does not already exist, it appends the necessary definition line. This step officially registers the new genome within the SnpEff framework, linking the genome name to its data directory.
  3. **Database Construction:** Finally, the script executes the snpEff build command. Crucially, it includes the -noCheckCds and -noCheckProtein flags. These parameters were found to be necessary to instruct SnpEff to bypass errors related to minor inconsistencies between the custom reference sequence and the GFF3 annotation file, thereby forcing the successful creation of the binary database file (snpEffectPredictor.bin).

Script:

```
#!/usr/bin/env bash
# STEP 1: Build the custom SnpEff database for the Bulida genome.

set -euo pipefail

## CONFIGURATION ##
SNPEFF_DIR=$(dirname $(which snpEff))/../share/snpeff-5.2-1
DB_NAME="bulida_v1"
DB_DIR="${SNPEFF_DIR}/data/${DB_NAME}"
REF_GENOME="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta"
GFF_FILE="/data/training2/info/assemblies/Bulida_annotationRNAseq.gff3"

echo "--- BUILDING SNPEFF DATABASE: ${DB_NAME} ---"

# Check if the database binary already exists. If so, building is not needed.
if [ -f "${DB_DIR}/snpEffV.bin" ]; then
    echo "SnpEff database '${DB_NAME}' already exists. Nothing to do."
    exit 0
fi

# --- 1A. Create directory structure and copy files ---
echo "Preparing database directory and files..."
mkdir -p "${DB_DIR}"
if [ ! -f "${DB_DIR}/sequences.fa" ]; then
    cp "${REF_GENOME}" "${DB_DIR}/sequences.fa"
fi
if [ ! -f "${DB_DIR}/genes.gff" ]; then
    cp "${GFF_FILE}" "${DB_DIR}/genes.gff"
fi

# --- 1B. Add genome to config file ---
if ! grep -q "${DB_NAME}.genome" "${SNPEFF_DIR}/snpEff.config"; then
```

```
    echo "Adding '${DB_NAME}' to snpEff.config..."
    echo -e "\n# Custom Bulida genome for TFM\n${DB_NAME}.genome :
Apricot_Bulida_v1" >> "${SNPEFF_DIR}/snpEff.config"
else
    echo "Genome '${DB_NAME}' already in snpEff.config."
fi

# --- 1C. Build the database with the correct flags for SnpEff v5.2 ---
echo "Building database... This may take a few minutes."
# Using -noCheckCds and -noCheckProtein instead of the unsupported -noCheck
flag
snpEff build -gff3 -v "${DB_NAME}" -noCheckCds -noCheckProtein

echo "Database build process finished. Please check the log for errors."
```

12_merge_mutect2_vcfs.sh

- **Purpose:** The purpose of this script is to process and consolidate the individual VCF files generated by the GATK Mutect2 pipeline. Since Mutect2 was run in paired mode (mutant vs. normal), each output VCF contains data for two samples. This script's goal is to extract only the high-confidence (PASS) somatic variants from the mutant sample in each file and then merge them into a single, clean, multi-sample VCF. This final VCF contains only the mutant replicates and is the required input for the subsequent biological filtering step.

- **Logic:** The script performs a three-stage workflow to robustly handle the merging process and avoid potential "duplicate sample" errors.

    1. **Individual Variant Extraction and Cleaning:** The script first iterates through each of the four VCF files produced by FilterMutectCalls. For each file, it uses bcftools view to perform a simultaneous two-part filtering: it selects only variants marked as PASS (-f PASS) and extracts only the data column corresponding to the mutant sample (--samples). This process generates four new, clean, single-sample VCFs which are stored in a temporary directory.
    2. **Merging of Clean VCFs:** Next, the script uses bcftools merge to take the four clean, single-sample VCF files created in the previous stage as input. It combines them into one final multi-sample VCF file (mutect2_merged.vcf.gz) that now contains the high-confidence variants and genotype information for all four mutant replicates, ready for cross-sample comparison.
    3. **Cleanup:** The final step in the script is to remove the temporary directory and all the intermediate single-sample files, ensuring a clean workspace and conserving disk space.

Script:

```
#!/usr/bin/env bash
# SCRIPT 08 (Definitive Version): Extracts PASS-only variants for each mutant
sample and merges them.
# This single script handles the extraction and merging to avoid duplicate
sample errors.

set -euo pipefail

## CONFIGURATION ##
BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
# Input directory with the original filtered VCFs from Mutect2
MUTECT2_FILTERED_DIR="${BASE_DIR}/04_mutect2_calling/filtered_vcfs"
# Output directory for the final merged file
FINAL_DIR="${BASE_DIR}/07_final_candidates"
# Temporary directory for intermediate single-sample, PASS-only VCFs
TEMP_DIR="${FINAL_DIR}/temp_mutect2_pass_merge"
mkdir -p "${FINAL_DIR}"
mkdir -p "${TEMP_DIR}"

THREADS=8
MUTANT_SAMPLES=("mt1" "mt2" "mt3" "mt4")

# Final output file
MERGED_VCF_MUTECT2="${FINAL_DIR}/mutect2_merged.vcf.gz"

echo "--- MERGING MUTECT2 VCFs (Definitive Workflow) ---"

if [ ! -f "${MERGED_VCF_MUTECT2}" ]; then
```

```
    # --- STEP A: Extract PASS variants for EACH mutant sample individually --
-
    echo "Step A: Extracting PASS variants for single mutant samples..."
    VCF_LIST=() # Array to store the paths of the new, clean VCFs
    for sample in "${MUTANT_SAMPLES[@]}"; do
        INPUT_VCF="${MUTECT2_FILTERED_DIR}/${sample}_vs_pooled_wt.filtered.vcf
.gz"
        OUTPUT_VCF="${TEMP_DIR}/${sample}.pass.vcf.gz"

        if [ -f "${INPUT_VCF}" ]; then
            echo "Processing sample '${sample}': Filtering for PASS and
selecting only this sample..."
            # This command does both things: selects PASS variants AND only
the specified mutant sample
            bcftools view --threads "${THREADS}" -f PASS --samples "${sample}"
-Oz -o "${OUTPUT_VCF}" "${INPUT_VCF}"
            bcftools index --threads "${THREADS}" "${OUTPUT_VCF}"
            VCF_LIST+=("${OUTPUT_VCF}")
        else
            echo "Warning: Input file ${INPUT_VCF} not found. Skipping."
        fi
    done

    # --- STEP B: Merge the new, clean, single-sample VCFs ---
    if [ ${#VCF_LIST[@]} -eq 4 ]; then # Check if we have all 4 files
        echo "Step B: Merging the 4 clean single-sample VCFs..."
        bcftools merge \
          --threads "${THREADS}" \
          -Oz -o "${MERGED_VCF_MUTECT2}" \
          "${VCF_LIST[@]}"

        bcftools index --threads "${THREADS}" "${MERGED_VCF_MUTECT2}"
        echo "Mutect2 VCFs merged successfully into: ${MERGED_VCF_MUTECT2}"
    else
        echo "Error: Could not find all 4 required PASS VCFs to merge. Found
only ${#VCF_LIST[@]}."
        exit 1
    fi

    # --- STEP C: Cleanup ---
    echo "Step C: Removing temporary directory..."
    rm -rf "${TEMP_DIR}"

else
    echo "Merged Mutect2 VCF already exists at ${MERGED_VCF_MUTECT2}.
Skipping."
fi
```

4 GATK HaplotypeCaller Filtering Workflow

run_final_iterative_filtering.sh

- **Purpose:** The purpose of this script is to perform a systematic filtering experiment on the joint-called GATK VCF file. As discussed with the project supervisor, a key challenge is to apply a technical filter that is strict enough to remove artifacts but not so strict that it eliminates true variants in samples with lower sequencing coverage. This script automates the process of testing multiple filtering thresholds, allowing for an empirical evaluation of how filter stringency affects the final variant count. The final output is a set of VCF files, each filtered with a different stringency, which provides the necessary data to justify the selection of an optimal filtering threshold for the main analysis.

- **Logic:** The script is built around a main loop that iterates through a predefined list of percentages (e.g., 3%, 5%, 10%). For each percentage, it executes a multi-step filtering workflow.

    1. **Automated Coverage Parsing:** Before the loop, the script reads the all_samples.mosdepth_summary_combined.txt file and uses awk to automatically parse and store the mean sequencing coverage for each of the 8 samples.
    2. **Proportional Threshold Calculation:** Inside the loop, for the current percentage, the script calculates a specific minimum read depth (DP) threshold for each of the 8 samples. This threshold is directly proportional to the sample's mean coverage (e.g., for the 10% run, a sample with 103x coverage gets a DP threshold of 10).
    3. **Chained Filtering Command:** The script then constructs a single, powerful command string that pipes together a series of bcftools filter commands. This chain first applies the 8 individual proportional DP filters (one for each sample) and then applies a final filter for standard quality metrics (QD, MQ, FS). This "chained" approach ensures that a variant must pass all conditions sequentially to be kept.
    4. **Execution and Reporting:** The complete command string is executed using eval, and the final filtered output is saved to a descriptively named VCF file (e.g., final_bcftools_propDP_15pct.filtered.vcf.gz). Finally, the script counts the number of variants in the new file and prints the result, allowing for a clear comparison between the different percentage thresholds.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Performs iterative filtering by piping a series of simple bcftools
commands.
# This is the most robust method to avoid all shell parsing errors.

set -euo pipefail

## CONFIGURATION ##
EXP_DIR="/data/training2/analisis_TFM_Bulida_Precoz/08_parameter_change_analys
is"
INPUT_VCF="${EXP_DIR}/01_proportional_filtering/all_samples_8.joint.vcf.gz"
COVERAGE_SUMMARY_FILE="${EXP_DIR}/all_samples.mosdepth_summary_combined.txt"
OUTPUT_DIR="${EXP_DIR}/01_proportional_filtering"

PERCENTAGES=(3 5 7 10 15)
VCF_SAMPLE_ORDER=("mt1" "mt2" "mt3" "mt4" "wt1" "wt2" "wt3" "wt4")
THREADS=8

echo "--- Starting Final, Step-by-Step Iterative Filtering Workflow ---"

# --- 1. Extract mean coverages ---
declare -A COVERAGES
```

```
while read -r sample coverage; do
    COVERAGES[$sample]=$coverage
done < <(awk '/^=== Content of: / {split($4, a, "."); sample=a[1]} /^[t]otal/
{print sample, $4}' "${COVERAGE_SUMMARY_FILE}")

# --- 2. Main loop to process each percentage ---
for percent in "${PERCENTAGES[@]}"; do
    echo -e "\n============================================================"
    echo "--- Processing threshold: ${percent}% ---"

    OUTPUT_VCF_FILTERED="${OUTPUT_DIR}/final_bcftools_propDP_${percent}pct.fil
tered.vcf.gz"

    if [ -f "${OUTPUT_VCF_FILTERED}" ]; then
        echo "Output file for ${percent}% already exists. Skipping."
        continue
    fi

    # Calculate DP thresholds
    DP_THRESHOLDS=()
    for sample in "${VCF_SAMPLE_ORDER[@]}"; do
        coverage=${COVERAGES[$sample]}
        threshold=$(LC_NUMERIC=C printf "%.0f" $(echo "${coverage} *
${percent} / 100" | bc -l))
        DP_THRESHOLDS+=($threshold)
    done
    echo "Calculated DP thresholds for ${percent}%: ${DP_THRESHOLDS[*]}"

    # --- 3. Build and execute the command chain ---
    # Start with the input file
    CMD="bcftools view ${INPUT_VCF}"

    # Add a separate filter for EACH sample's DP
    for i in "${!VCF_SAMPLE_ORDER[@]}"; do
        CMD+=" | bcftools filter --threads ${THREADS} -i 'FORMAT/DP[${i}] >=
${DP_THRESHOLDS[${i}]}'"
    done

    # Add the final quality filter and redirect to output file
    CMD+=" | bcftools filter --threads ${THREADS} -i 'QD >= 2.0 && MQ >= 40.0
&& FS <= 60.0' -Oz -o ${OUTPUT_VCF_FILTERED}"

    echo "Applying chained filters..."
    # Use eval to execute the command string with all its pipes
    eval ${CMD}

    bcftools index --threads "${THREADS}" "${OUTPUT_VCF_FILTERED}"

    COUNT=$(bcftools view -H "${OUTPUT_VCF_FILTERED}" | wc -l)
    echo "Filtering for ${percent}% complete. Found ${COUNT} variants."
    echo "Output file: ${OUTPUT_VCF_FILTERED}"
done

echo -e "\n### Iterative filtering workflow finished. ###"
```

find_strict_candidates_iteratively.sh

- **Purpose:** The purpose of this script is to apply the final and most stringent biological filter to each of the VCF files generated by the previous proportional filtering experiment. By systematically testing each technically filtered dataset (3%, 5%, 15%, etc.), this script's primary goal is to determine the optimal technical filtering threshold that yields a robust yet manageable number of high-confidence candidate variants. The final output of this script is a summary table that quantitatively demonstrates the relationship between technical filter stringency and the number of biologically relevant candidates.
- **Logic:** The script operates by iterating through a predefined list of percentages that correspond to the previously generated VCF files. For each percentage, it performs the following actions:
    1. **Input File Selection:** It defines the path to the input VCF file that has been filtered with the current percentage (e.g., final_bcftools_propDP_15pct.filtered.vcf.gz).
    2. **Strict Biological Filtering:** It then uses bcftools view to apply the core biological filter. The include expression (--include) is 'N_PASS(GT[0-3]="alt")==4 && N_PASS(GT[4-7]="alt")==0'. This expression selects only variants that meet two simultaneous conditions: they must be present in all four mutant samples (N_PASS(GT[0-3]="alt")==4) and be completely absent from all four wild-type samples (N_PASS(GT[4-7]="alt")==0).
    3. **Output Generation:** The variants that pass this strict filter are saved to a new, descriptively named VCF file (e.g., strict_candidates_15pct.vcf.gz). The script then counts the number of variants in this new file and appends the result (the percentage and the variant count) to a summary text file, creating a final report of the experiment.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Iterates through proportionally filtered VCFs to find candidates
# that match the strict biological criteria (4/4 in mutants, 0/4 in wild-
types).

set -euo pipefail

## CONFIGURATION ##
EXP_DIR="/data/training2/analisis_TFM_Bulida_Precoz/08_parameter_change_analys
is"
INPUT_DIR="${EXP_DIR}/01_proportional_filtering"
# A new directory to store the final results of this analysis
OUTPUT_DIR="${EXP_DIR}/03_strict_candidate_search"
mkdir -p "${OUTPUT_DIR}"

PERCENTAGES=(3 5 7 10 15)
VCF_SAMPLE_ORDER=("mt1" "mt2" "mt3" "mt4" "wt1" "wt2" "wt3" "wt4")
THREADS=8

# --- Define the strict biological filter expression ---
# Include variants where ALL 4 mutants have the variant AND ALL 4 wild-types
do NOT.
STRICT_FILTER_EXP='N_PASS(GT[0-3]="alt")==4 && N_PASS(GT[4-7]="alt")==0'

## SCRIPT LOGIC ##
echo "--- Starting Workflow: Searching for Strict 4/4 Candidates ---"
```

```
echo "Filter to be applied: ${STRICT_FILTER_EXP}"

# Create a summary file header
SUMMARY_FILE="${OUTPUT_DIR}/strict_candidate_summary.txt"
echo -e "Threshold_Percent\tFinal_Candidate_Count" > "${SUMMARY_FILE}"

for percent in "${PERCENTAGES[@]}"; do
    echo -e "\n============================================================"
    echo "--- Analyzing file from threshold: ${percent}% ---"

    INPUT_VCF="${INPUT_DIR}/final_bcftools_propDP_${percent}pct.filtered.vcf.g
z"
    OUTPUT_VCF="${OUTPUT_DIR}/strict_candidates_${percent}pct.vcf.gz"

    if [ ! -f "${INPUT_VCF}" ]; then
        echo "Input file ${INPUT_VCF} not found. Skipping."
        continue
    fi

    # Apply the strict biological filter
    bcftools view \
      --threads "${THREADS}" \
      --samples "${VCF_SAMPLE_ORDER}" \
      --include "${STRICT_FILTER_EXP}" \
      -Oz -o "${OUTPUT_VCF}" \
      "${INPUT_VCF}"

    # Count the results
    COUNT=$(bcftools view -H "${OUTPUT_VCF}" | wc -l)

    echo "Result for ${percent}% threshold: Found ${COUNT} strict candidates."

    # Append the result to the summary file
    echo -e "${percent}\t${COUNT}" >> "${SUMMARY_FILE}"
done

echo -e "\n### Workflow Complete! ###"
echo "Summary of results:"
cat "${SUMMARY_FILE}"
```

GATK_prepare_notebook_inputs.sh

- **Purpose:** The purpose of this script is to perform all the necessary data preparation steps to generate the final input tables required for the comprehensive Jupyter Notebook analysis of the GATK HaplotypeCaller results. It takes the final, most stringently filtered candidate VCF (from the 15% proportional depth filter) and processes it to create two distinct tables: one containing the full SnpEff annotation for all candidates, and a second one specifically formatted with Allele Depth (AD) and Depth (DP) information for the subsequent Variant Allele Frequency (VAF) analysis.
- **Logic:** The script executes a multi-step data extraction workflow that separates annotation from VAF data preparation.
    1. **Functional Annotation:** The script first calls snpEff ann to annotate the input VCF (strict_candidates_15pct.vcf.gz), creating a temporary VCF file with the ANN field populated.
    2. **Full Annotation Table Extraction:** It then uses bcftools query on this annotated VCF to extract the basic variant information and the full ANN field into a .tsv table, which will be used for the global characterization of all candidates in the notebook.
    3. **VAF Data Extraction:** This is the most critical step. The script uses bcftools query again, but with a different logic. It uses the -R flag to specify the candidate VCF as a list of regions. It then queries the larger, pre-filtered VCF that still contains all 8 samples (final_bcftools_propDP_15pct.filtered.vcf.gz), extracting the per-sample AD and DP fields only for the candidate variants. This robustly ensures that the VAF table contains the necessary genotype information from all samples for the specific variants of interest.
    4. **Cleanup:** Finally, the temporary annotated VCF created in the first step is removed to keep the workspace clean.

Script:

```
#!/usr/bin/env bash
# Prepares all data tables for the GATK analysis notebook.
# It correctly extracts VAF data from the full, pre-filtered VCF.

set -euo pipefail

## CONFIGURATION ##
EXP_DIR="/data/training2/analisis_TFM_Bulida_Precoz/08_parameter_change_analysis"
GATK_WORKFLOW_DIR="${EXP_DIR}/GATK_HaplotypeCaller_workflow"

# Input VCF with the 224 strict candidates
STRICT_VCF="${GATK_WORKFLOW_DIR}/03_strict_candidate_search/strict_candidates_15pct.vcf.gz"
# Input VCF with all samples, filtered by the 15% proportional threshold
FULL_VCF="${GATK_WORKFLOW_DIR}/01_proportional_filtering/final_bcftools_propDP_15pct.filtered.vcf.gz"

# Output directory for the notebook data
NOTEBOOK_DATA_DIR="${GATK_WORKFLOW_DIR}/notebook_data"
mkdir -p "${NOTEBOOK_DATA_DIR}"

SNPEFF_GENOME_NAME="bulida_v1"

# --- Output Files ---
```

```
ANNOTATED_VCF="${NOTEBOOK_DATA_DIR}/gatk_15pct_strict.ann.vcf"
FULL_TABLE_OUTPUT="${NOTEBOOK_DATA_DIR}/gatk_15pct_strict_full_table.tsv"
VAF_TABLE_OUTPUT="${NOTEBOOK_DATA_DIR}/gatk_15pct_vaf_data.tsv"

echo "--- Preparing all data files for final GATK Notebook Analysis ---"

# --- 1. Annotate VCF with SnpEff ---
echo "Step 1: Annotating the strict candidate VCF..."
snpEff ann -v "${SNPEFF_GENOME_NAME}" "${STRICT_VCF}" > "${ANNOTATED_VCF}"

# --- 2. Extract Full Annotation Table ---
echo "Step 2: Extracting full annotation table..."
bcftools query -f '%CHROM\t%POS\t%REF\t%ALT\t%INFO/ANN\n' "${ANNOTATED_VCF}" >
"${FULL_TABLE_OUTPUT}"

# --- 3. Extract VAF Data Table from the correct source ---
echo "Step 3: Extracting Allele Depth data for VAF analysis..."
# The -R flag tells bcftools to only process regions listed in the strict VCF
file,
# but it extracts the data from the FULL VCF that contains all 8 samples.
bcftools query -f '%CHROM\t%POS\t%REF\t%ALT[\t%AD\t%DP]\n' -R "${STRICT_VCF}"
"${FULL_VCF}" > "${VAF_TABLE_OUTPUT}"
echo "VAF data table created successfully."

# --- 4. Cleanup ---
rm "${ANNOTATED_VCF}"

echo -e "\n### Data preparation complete. ###"
```

5 Mutect2 Analysis Workflow

mutect2_run_filtering.sh

- **Purpose:** This script performs the first and most critical technical filtering step for the GATK Mutect2 pipeline. Its purpose is to take the raw variant calls generated by Mutect2 for each mutant sample and apply GATK's sophisticated filtering models. This process is designed to evaluate each potential somatic variant against a set of quality metrics and common sequencing artifacts, providing a "tagged" VCF file where each variant is either marked as PASS or with a specific reason for failure. This is the initial step in cleaning the Mutect2 callset to improve its reliability.
- **Logic:** The script operates by iterating through a predefined list of the four mutant samples (mt1 to mt4). For each sample in the loop, it defines the paths for the corresponding raw input VCF (from the raw_vcfs directory) and the filtered output VCF. It then executes the gatk FilterMutectCalls command. This tool applies its internal statistical models to assess evidence of artifacts, such as strand bias or contamination. It does not physically remove any variants; instead, it annotates the FILTER column of each record in the output VCF with either PASS or a specific filter name (e.g., weak_evidence, triallelic_site), creating a new set of filtered VCFs ready for the next stage of the workflow.

Script:

```
#!/usr/bin/env bash
# Applies the default GATK FilterMutectCalls filters to the raw VCFs.

set -euo pipefail

## CONFIGURATION ##
BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
EXP_DIR="${BASE_DIR}/08_parameter_change_analysis"
RAW_VCF_DIR="${BASE_DIR}/04_mutect2_calling/raw_vcfs"
# A new directory for the outputs of this workflow
OUTPUT_DIR="${EXP_DIR}/mutect2_final_workflow"

# --- CORRECTED DIRECTORY NAME AS PER YOUR SUGGESTION ---
FILTERED_DIR="${OUTPUT_DIR}/01_mutect2_filtered"
mkdir -p "${FILTERED_DIR}"

REF_GENOME="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta"
MUTANT_SAMPLES=("mt1" "mt2" "mt3" "mt4")

echo "--- Running GATK FilterMutectCalls on Raw VCFs ---"
for sample in "${MUTANT_SAMPLES[@]}"; do
    RAW_VCF="${RAW_VCF_DIR}/${sample}_vs_pooled_wt.raw.vcf.gz"
    FILTERED_VCF_OUT="${FILTERED_DIR}/${sample}.filtered.vcf.gz"

    if [ -f "${FILTERED_VCF_OUT}" ]; then
        echo "Filtered file for ${sample} already exists. Skipping."
        continue
    fi

    echo "Processing ${sample}..."
    gatk FilterMutectCalls -R "${REF_GENOME}" -V "${RAW_VCF}" -O
"${FILTERED_VCF_OUT}"
done
echo "GATK filtering complete. Tagged VCFs are in: ${FILTERED_DIR}"
```

mutect2_apply_coverage_filter.sh

- **Purpose:** The purpose of this script is to apply a second, more stringent layer of technical filtering to the Mutect2 variant calls. After the initial filtering with GATK's default models, this script adds a crucial quality control step based on read depth. By establishing a minimum read depth threshold that is proportional to each sample's mean coverage (e.g., 15%), it ensures that the variants retained for final analysis are supported by a robust amount of sequencing evidence, thereby increasing the overall confidence of the callset.
- **Logic:** The script first calculates the specific depth (DP) threshold for each of the four mutant samples based on a predefined percentage. It then iterates through each of the GATK-filtered VCF files and applies a three-stage command chain using piped bcftools commands. For each input file, the chain executes the following:
    1. **Sample Selection:** bcftools view first isolates the data corresponding only to the mutant sample, discarding the pooled normal control.
    2. **GATK Filter Selection:** The output is then piped to bcftools filter, which uses the expression -i 'FILTER="PASS"' to keep only the variants that were successfully passed by the initial FilterMutectCalls step.
    3. **Proportional Depth Filtering:** Finally, the stream is passed to a second bcftools filter command. This one applies the calculated proportional depth threshold (-i "FORMAT/DP >= X") for that specific sample, ensuring that only variants with sufficient read support are retained. The final output is a set of four VCF files, each containing high-confidence variants that have passed both GATK's artifact filters and a rigorous, coverage-aware depth filter.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Applies a proportional DP filter and keeps only PASSing variants for
the mutant sample.

set -euo pipefail

## CONFIGURATION ##
# SET THE DESIRED PERCENTAGE THRESHOLD HERE !
PERCENT_TO_USE=15 # Using 15% as a robust, conservative choice

BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
EXP_DIR="${BASE_DIR}/08_parameter_change_analysis"
# --- Corrected input directory name ---
INPUT_DIR="${EXP_DIR}/mutect2_final_workflow/01_mutect2_filtered"
COVERAGE_SUMMARY_FILE="${EXP_DIR}/all_samples.mosdepth_summary_combined.txt"
OUTPUT_DIR="${EXP_DIR}/mutect2_final_workflow/02_dp_filtered"
mkdir -p "${OUTPUT_DIR}"

MUTANT_SAMPLES=("mt1" "mt2" "mt3" "mt4")
THREADS=8

# --- 1. Calculate DP thresholds ---
declare -A DP_THRESHOLDS
while read -r sample coverage; do
    threshold=$(LC_NUMERIC=C printf "%.0f" $(echo "${coverage} *
${PERCENT_TO_USE} / 100" | bc -l))
    DP_THRESHOLDS[$sample]=$threshold
done < <(awk '/^=== Content of: / {split($4, a, "."); sample=a[1]} /^[t]otal/
{print sample, $4}' "${COVERAGE_SUMMARY_FILE}")

# --- 2. Main loop to filter each sample ---
```

```
echo "--- Applying Proportional DP Filter (${PERCENT_TO_USE}%) ---"
for sample in "${MUTANT_SAMPLES[@]}"; do
    INPUT_VCF="${INPUT_DIR}/${sample}.filtered.vcf.gz"
    OUTPUT_VCF="${OUTPUT_DIR}/${sample}.propDP_${PERCENT_TO_USE}pct.vcf.gz"
    DP_THRESHOLD=${DP_THRESHOLDS[$sample]}

    if [ -f "${OUTPUT_VCF}" ]; then
        echo "Final filtered file for ${sample} already exists. Skipping."
        continue
    fi

    echo "Processing ${sample} with DP threshold >= ${DP_THRESHOLD}..."

    # This command chain does everything:
    # 1. Selects only the mutant sample column.
    # 2. Keeps only variants marked as PASS from the previous GATK step.
    # 3. Keeps only variants where that sample's DP meets the proportional
threshold.
    bcftools view --threads "${THREADS}" --samples "${sample}" "${INPUT_VCF}"
\
    | bcftools filter --threads "${THREADS}" -i 'FILTER="PASS"' \
    | bcftools filter --threads "${THREADS}" -i "FORMAT/DP >= ${DP_THRESHOLD}"
-Oz -o "${OUTPUT_VCF}"

    bcftools index --threads "${THREADS}" "${OUTPUT_VCF}"
done
echo "Proportional DP filtering complete. Final VCFs are in ${OUTPUT_DIR}"
```

mutect2_merge_and_find_candidates.sh

- **Purpose:** This script serves as a comprehensive downstream analysis pipeline designed to process and compare two different sets of pre-filtered Mutect2 variants: one generated with GATK's default filters and another with an additional proportional depth filter. The purpose is to apply a consistent workflow of merging, biological filtering, annotation, and data extraction to both sets. This allows for a direct and fair comparison of the final candidate lists, making it possible to evaluate the impact of the different technical filtering strategies.
- **Logic:** The script is built around a reusable Bash function, run_analysis_pipeline, which is executed twice—once for each input dataset. The logic within this function is a multi-stage pipeline:
    1. **Pre-processing for Merging:** To prevent common errors, the script first performs a two-step cleaning process on each of the four input VCFs. It uses bcftools view to isolate only the mutant sample's data (solving potential "duplicate sample" errors) and then bcftools annotate to remove the problematic AS_FilterStatus INFO field (solving potential "wrong number of fields" errors). This generates four clean, single-sample VCFs.
    2. **Merging and Biological Filtering:** The four cleaned VCFs are then merged into a single multi-sample VCF using bcftools merge. Immediately after, bcftools view applies the strict biological filter ('N_PASS(GT="alt")==4') to this merged file, creating a final VCF containing only the variants present in all four mutant replicates.
    3. **Final Annotation and Data Extraction:** If any candidates pass the strict biological filter, the script proceeds to annotate this final VCF with SnpEff. It then uses bcftools query to extract the results into a tab-delimited table and a cut/sort pipeline to generate a unique list of affected gene IDs, preparing the data for the final interpretation stages.
    4. **Execution:** The main body of the script simply calls the run_analysis_pipeline function twice, providing the correct input directory and output file prefix for each of the two filtering strategies being compared.

Script:

```
#!/usr/bin/env bash
# Runs the full downstream analysis pipeline.
# Includes a step to remove the problematic 'AS_FilterStatus' field before
merging.

set -euo pipefail

##
==============================================================================
==============
## CONFIGURATION
##
==============================================================================
==============

BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
EXP_DIR="${BASE_DIR}/08_parameter_change_analysis"
FINAL_CANDIDATE_DIR="${EXP_DIR}/final_candidate_sets"
mkdir -p "${FINAL_CANDIDATE_DIR}"

# --- Input Directories ---
INPUT_DIR_DEFAULT="${EXP_DIR}/mutect2_final_workflow/01_mutect2_filtered"
INPUT_DIR_PROP_DP="${EXP_DIR}/mutect2_final_workflow/02_dp_filtered"

# --- Output Prefixes ---
OUTPUT_PREFIX_DEFAULT="mutect2_default_filter"
```

```
OUTPUT_PREFIX_PROP_DP="mutect2_plus_propDP_15pct"

# --- General Parameters ---
MUTANT_SAMPLES=("mt1" "mt2" "mt3" "mt4")
THREADS=8
SNPEFF_GENOME_NAME="bulida_v1"
STRICT_FILTER_EXP='N_PASS(GT="alt")==4'

##
================================================================================
==============
## ANALYSIS FUNCTION
##
================================================================================
==============

run_analysis_pipeline() {
    local input_dir="$1"
    local output_prefix="$2"
    local temp_dir="${FINAL_CANDIDATE_DIR}/temp_${output_prefix}"
    mkdir -p "${temp_dir}"

    echo "============================================================"
    echo "--- Starting Downstream Analysis for prefix: ${output_prefix} ---"
    echo "--- Input directory: ${input_dir} ---"

    # --- Step 1: Pre-processing before merge ---
    local VCF_LIST_FINAL=()
    for sample in "${MUTANT_SAMPLES[@]}"; do
        local original_vcf=$(find "${input_dir}" -name "${sample}.*.vcf.gz" |
head -n 1)
        local single_sample_vcf="${temp_dir}/${sample}.single.vcf.gz"
        local clean_vcf="${temp_dir}/${sample}.clean.vcf.gz"

        if [ -f "$original_vcf" ]; then
            echo "Processing ${sample}:"
            # 1a: Extract only the mutant sample to solve duplicate name issue
            echo "  - Extracting mutant sample..."
            bcftools view --threads "${THREADS}" --samples "${sample}" -Oz -o
"${single_sample_vcf}" "${original_vcf}"

            # 1b: Remove the problematic INFO field to solve merge error
            echo "  - Removing AS_FilterStatus field..."
            bcftools annotate --threads "${THREADS}" -x INFO/AS_FilterStatus -
Oz -o "${clean_vcf}" "${single_sample_vcf}"

            bcftools index --threads "${THREADS}" "${clean_vcf}"
            VCF_LIST_FINAL+=("${clean_vcf}")
        else
            echo "ERROR: Could not find VCF for ${sample} in ${input_dir}"
            exit 1
        fi
    done

    # 2. Merge the super-clean, single-sample VCFs
    echo "Step 2: Merging 4 clean single-sample VCFs..."
    local MERGED_VCF="${temp_dir}/merged.vcf.gz"
```

```
    bcftools merge --threads "${THREADS}" -Oz -o "${MERGED_VCF}"
"${VCF_LIST_FINAL[@]}"
    bcftools index --threads "${THREADS}" "${MERGED_VCF}"

    # 3. Apply the strict biological filter (4 out of 4)
    local
STRICT_CANDIDATES_VCF="${FINAL_CANDIDATE_DIR}/${output_prefix}_strict_candidat
es.vcf.gz"
    echo "Step 3: Applying strict biological filter (${STRICT_FILTER_EXP})..."
    bcftools view --threads "${THREADS}" --include "${STRICT_FILTER_EXP}" -Oz
-o "${STRICT_CANDIDATES_VCF}" "${MERGED_VCF}"
    bcftools index --threads "${THREADS}" "${STRICT_CANDIDATES_VCF}"
    local COUNT=$(bcftools view -H "${STRICT_CANDIDATES_VCF}" | wc -l)
    echo "Found ${COUNT} strict (4/4) candidates for this filter set."

    # 4. Annotate and Extract Results
    if [ "${COUNT}" -gt 0 ]; then
        local
ANNOTATED_VCF="${FINAL_CANDIDATE_DIR}/${output_prefix}_final.ann.vcf"
        local
TABLE_OUTPUT="${FINAL_CANDIDATE_DIR}/${output_prefix}_final_table.tsv"
        local
GENE_LIST_OUTPUT="${FINAL_CANDIDATE_DIR}/${output_prefix}_final_gene_list.txt"

        echo "Step 4: Annotating and extracting final results..."
        snpEff ann -v "${SNPEFF_GENOME_NAME}" "${STRICT_CANDIDATES_VCF}" >
"${ANNOTATED_VCF}"
        bcftools query -f '%CHROM\t%POS\t%REF\t%ALT\t%INFO/ANN\n'
"${ANNOTATED_VCF}" > "${TABLE_OUTPUT}"
        cut -f 5 "${TABLE_OUTPUT}" | cut -d '|' -f 5 | sort -u | grep -v
'^\s*$' > "${GENE_LIST_OUTPUT}"
    else
        echo "No candidates found, skipping annotation and extraction."
    fi

    # 5. Cleanup
    rm -rf "${temp_dir}"
    echo "Analysis for prefix '${output_prefix}' is complete."
}

##
================================================================================
==============
## SCRIPT EXECUTION
##
================================================================================
==============

# Execute the pipeline for the first filter strategy
run_analysis_pipeline "${INPUT_DIR_DEFAULT}" "${OUTPUT_PREFIX_DEFAULT}"

# Execute the pipeline for the second filter strategy
run_analysis_pipeline "${INPUT_DIR_PROP_DP}" "${OUTPUT_PREFIX_PROP_DP}"

echo -e "\n### Both analysis pipelines have finished successfully! ###"
echo "Final results are in: ${FINAL_CANDIDATE_DIR}"
```

mutect2_annotate_and_extract.sh

- **Purpose:** This script serves as the final data processing stage for a given set of high-confidence Mutect2 candidates. Its objective is to take a VCF file of biologically filtered variants, perform functional annotation, and then reformat the data into two final, user-friendly files: a comprehensive tab-delimited table (.tsv) and a clean, unique list of affected gene IDs. These files are the definitive outputs of the pipeline, ready for interpretation and downstream analysis, such as Gene Ontology.
- **Logic:** The script executes a sequential, three-step workflow to process the input candidate VCF.
    1. **Functional Annotation with SnpEff:** First, the script calls snpEff ann, using the custom bulida_v1 database to annotate each variant. This process adds the critical ANN field to the VCF, which contains predictions about the variant's functional effect and impact on gene models.
    2. **Extraction to Table Format:** Next, the script uses bcftools query to parse the newly annotated VCF file. It extracts the essential variant information (chromosome, position, alleles) along with the full ANN annotation string, writing these results into a structured .tsv table.
    3. **Creation of Unique Gene List:** In the final step, the script processes the table generated in the previous stage. It uses a chain of cut and sort -u commands to isolate the fifth sub-field of the ANN string, which corresponds to the Gene ID, thereby producing a clean and unique list of all genes affected by the candidate variants.

Script:

```
#!/usr/bin/env bash
# Annotate and extract the final list of independent Mutect2 candidates.

set -euo pipefail

## CONFIGURATION ##
BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
EXP_DIR="${BASE_DIR}/08_parameter_change_analysis"
# --- Input/Output Directories for the independent Mutect2 analysis ---
CANDIDATE_DIR="${EXP_DIR}/mutect2_independent_analysis/candidates"
FINAL_RESULTS_DIR="${EXP_DIR}/mutect2_independent_analysis/final_results"
mkdir -p "${FINAL_RESULTS_DIR}"

# --- SnpEff Database Name ---
SNPEFF_GENOME_NAME="bulida_v1"

# --- Input File ---
# This is the VCF containing the strict 4-out-of-4 candidates from the M2
script
INPUT_VCF="${CANDIDATE_DIR}/mutect2_strict_4of4_candidates.vcf.gz"

# --- Output Files ---
ANNOTATED_VCF="${FINAL_RESULTS_DIR}/mutect2_final.ann.vcf"
TABLE_OUTPUT="${FINAL_RESULTS_DIR}/mutect2_final_table.tsv"
GENE_LIST_OUTPUT="${FINAL_RESULTS_DIR}/mutect2_final_gene_list.txt"

echo "--- Final Annotation and Extraction for Mutect2 Candidates ---"

if [ ! -f "${INPUT_VCF}" ]; then
    echo "ERROR: Input file ${INPUT_VCF} not found. Please run script M2
first."
    exit 1
```

```
fi

# Step 1: Functional Annotation with SnpEff
echo "Step 1: Annotating final candidates..."
snpEff ann -v "${SNPEFF_GENOME_NAME}" "${INPUT_VCF}" > "${ANNOTATED_VCF}"
echo "Annotation complete."

# Step 2: Extraction to a TSV table
# We extract all candidates, as they have already passed the strict biological
filter.
echo "Step 2: Extracting all candidates into a table..."
bcftools query -f '%CHROM\t%POS\t%REF\t%ALT\t%INFO/ANN\n' "${ANNOTATED_VCF}" >
"${TABLE_OUTPUT}"
echo "Final Mutect2 table created: ${TABLE_OUTPUT}"

# Step 3: Create final gene list
echo "Step 3: Creating final gene list..."
cut -f 5 "${TABLE_OUTPUT}" | cut -d '|' -f 5 | sort -u | grep -v '^\s*$' >
"${GENE_LIST_OUTPUT}"
echo "Final Mutect2 gene list created: ${GENE_LIST_OUTPUT}"

echo -e "\n### Independent Mutect2 analysis workflow finished! ###"
echo "Final results are located in: ${FINAL_RESULTS_DIR}"
```

6 Final Analysis & Interpretation

final_prepare_for_notebook.sh

- **Purpose:** The purpose of this script is to act as a centralized data preparation engine for the final comparative analysis. It is designed to process the final candidate VCF files from all three separate analytical pipelines (GATK 15% DP, Mutect2 Default, and Mutect2 15% DP). For each pipeline, it generates the two essential tables required as input for the main Jupyter Notebook: a comprehensive table with all functional annotations and a second table specifically formatted with Allele Depth (AD) and read Depth (DP) information for VAF analysis.
- **Logic:** The script is built around a reusable Bash function, process_vcf_for_notebook, to ensure a consistent workflow is applied to each of the three candidate sets. The main body of the script calls this function three times, once for each pipeline. The function itself executes a sequential three-step process:
    1. **Functional Annotation:** It first calls snpEff ann on the input candidate VCF to generate a temporary, annotated version of the file with the ANN field populated.
    2. **Data Table Extraction:** It then processes the annotated VCF with bcftools query twice. The first call extracts the variant coordinates and the full ANN field into the main data table (..._all_variants.tsv). The second call extracts the per-sample AD and DP fields into the VAF data table (..._vaf_data.tsv). A key detail in this step is that for the GATK candidates, it intelligently queries the larger, pre-filtered VCF containing all 8 samples to ensure the VAF data is correctly retrieved.
    3. **Cleanup:** Finally, the temporary annotated VCF created in the first step is removed, leaving only the final data tables ready for the notebook.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Prepares all necessary data tables for the comprehensive analysis
notebook.
# It processes the final candidate VCFs from all three pipelines.

set -euo pipefail

##
================================================================================
==============
## CONFIGURATION
##
================================================================================
==============
BASE_DIR="/data/training2/analisis_TFM_Bulida_Precoz"
EXP_DIR="${BASE_DIR}/08_parameter_change_analysis"
SNPEFF_GENOME_NAME="bulida_v1"
THREADS=8

# --- Define Input & Output Pairs ---

# Pair 1: GATK (15% Proportional Filter)
GATK_INPUT_VCF="${EXP_DIR}/GATK_HaplotypeCaller_workflow/03_strict_candidate_s
earch/strict_candidates_15pct.vcf.gz"
GATK_OUTPUT_TABLE="${EXP_DIR}/GATK_HaplotypeCaller_workflow/notebook_data/gatk
_15pct_all_variants.tsv"
GATK_VAF_TABLE="${EXP_DIR}/GATK_HaplotypeCaller_workflow/notebook_data/gatk_15
pct_vaf_data.tsv"
```

```
GATK_VCF_FOR_VAF="${EXP_DIR}/GATK_HaplotypeCaller_workflow/01_proportional_filtering/final_bcftools_propDP_15pct.filtered.vcf.gz"

# Pair 2: Mutect2 (Default Filter)
MUTECT2_DEFAULT_INPUT_VCF="${EXP_DIR}/mutect2_final_workflow/final_candidate_sets/mutect2_default_filter_strict_candidates.vcf.gz"
MUTECT2_DEFAULT_OUTPUT_TABLE="${EXP_DIR}/mutect2_final_workflow/final_candidate_sets/mutect2_default_filter_all_variants.tsv"
MUTECT2_DEFAULT_VAF_TABLE="${EXP_DIR}/mutect2_final_workflow/final_candidate_sets/mutect2_default_vaf_data.tsv"

# Pair 3: Mutect2 (15% Proportional DP Filter)
MUTECT2_PROPDP_INPUT_VCF="${EXP_DIR}/mutect2_final_workflow/final_candidate_sets/mutect2_plus_propDP_15pct_strict_candidates.vcf.gz"
MUTECT2_PROPDP_OUTPUT_TABLE="${EXP_DIR}/mutect2_final_workflow/final_candidate_sets/mutect2_plus_propDP_15pct_all_variants.tsv"
MUTECT2_PROPDP_VAF_TABLE="${EXP_DIR}/mutect2_final_workflow/final_candidate_sets/mutect2_plus_propDP_15pct_vaf_data.tsv"

##
================================================================================
==============
## ANALYSIS FUNCTION
##
================================================================================
==============

process_vcf_for_notebook() {
    local input_vcf="$1"
    local output_table="$2"
    local vaf_table="$3"
    local vcf_for_vaf_query="$4" # VCF containing all samples for VAF
extraction
    local pipeline_name="$5"

    local temp_ann_vcf="${output_table%.tsv}.ann.vcf"

    echo "============================================================"
    echo "--- Preparing data for: ${pipeline_name} ---"

    if [ ! -f "$input_vcf" ]; then
        echo "ERROR: Input VCF not found: ${input_vcf}"
        return 1
    fi

    # 1. Annotate with SnpEff
    echo "Annotating with SnpEff..."
    snpEff ann -v "${SNPEFF_GENOME_NAME}" "${input_vcf}" > "${temp_ann_vcf}"

    # 2. Extract full annotation table
    echo "Extracting full annotation table..."
    bcftools query -f '%CHROM\t%POS\t%REF\t%ALT\t%INFO/ANN\n'
"${temp_ann_vcf}" > "${output_table}"

    # 3. Extract VAF data table
    echo "Extracting VAF data..."
```

```
    # If a specific VCF for VAF query is provided, use it. Otherwise, use the
input VCF.
    local source_vcf_for_vaf=${vcf_for_vaf_query:-$input_vcf}
    bcftools query -f '%CHROM\t%POS\t%REF\t%ALT[\t%AD\t%DP]\n' --samples
"mt1,mt2,mt3,mt4" -R "${input_vcf}" "${source_vcf_for_vaf}" > "${vaf_table}"

    rm "${temp_ann_vcf}" # Cleanup
    echo "Preparation for ${pipeline_name} complete."
}

##
==============================================================================
==============
## SCRIPT EXECUTION
##
==============================================================================
==============

# The GATK VAF query needs the original multi-sample VCF
process_vcf_for_notebook "${GATK_INPUT_VCF}" "${GATK_OUTPUT_TABLE}"
"${GATK_VAF_TABLE}" "${GATK_VCF_FOR_VAF}" "GATK HaplotypeCaller"

# The Mutect2 VAF queries can use their own strict candidate files, as they
already contain the 4 mutant samples
process_vcf_for_notebook "${MUTECT2_DEFAULT_INPUT_VCF}"
"${MUTECT2_DEFAULT_OUTPUT_TABLE}" "${MUTECT2_DEFAULT_VAF_TABLE}" "" "Mutect2
Default"

process_vcf_for_notebook "${MUTECT2_PROPDP_INPUT_VCF}"
"${MUTECT2_PROPDP_OUTPUT_TABLE}" "${MUTECT2_PROPDP_VAF_TABLE}" "" "Mutect2
Proportional DP"

echo -e "\n### All data files have been prepared for the final notebook
analysis. ###"
```

Variant_analysis_HaplotypeCaller_vs_Mutect2.ipynb

- **Purpose:** This Jupyter Notebook serves as the final, integrative analysis of the entire project. Its primary purpose is to synthesize and compare the candidate gene lists generated from the three separate analytical pipelines (GATK HaplotypeCaller, Mutect2 with default filtering, and Mutect2 with proportional depth filtering). The goal is to identify the most robust, high-confidence candidate genes by finding the intersection between these different methodologies and to generate the final summary figures and tables for the thesis report.
- **Logic:** The notebook executes a systematic workflow to compare and summarize the results.
    1. **Data Loading:** It begins by loading the final, clean gene lists from each of the three pipelines. These lists are read into three distinct Python set objects, which is an efficient data structure for performing comparisons and finding intersections.
    2. **Comparative Visualization:** The notebook then uses the matplotlib-venn library to generate a series of Venn diagrams. It systematically creates all pairwise comparisons between the three sets, as well as a final three-way Venn diagram. This provides a clear visual representation of the degree of concordance and discordance between the different variant calling and filtering strategies.
    3. **Identification of Common Genes:** Using Python's set intersection operations, the script programmatically identifies the list of "gold-standard" genes, those that are present in the results of all three analytical pipelines.
    4. **Final Summary Table Generation:** In the final step, the notebook takes this list of highest-confidence genes. It then goes back to the full data tables from each pipeline to find the specific variants located within these common genes. It compiles this information into a single, consolidated summary table that details each high-confidence variant, the pipeline(s) that detected it, and its predicted functional impact.

You can find this script on the GitHub repository for this project.

1_extract_sequences.sh

- **Purpose:** The purpose of this script is to generate a FASTA file containing the protein sequences for a specific list of candidate genes. This is a critical data preparation step that creates the necessary input file for downstream functional analysis tools like BLASTp and InterProScan, which operate on protein sequences to infer biological function.
- **Logic:** The script performs a two-stage process to first generate a complete proteome and then select the sequences of interest.
    1. **Full Proteome Extraction:** The script first calls the gffread utility. This tool reads the genome-wide GFF3 annotation file and, using the reference genome FASTA file (-g flag), translates all annotated coding sequences (CDS) into their corresponding protein sequences. The result is a single FASTA file (all_proteins.fasta) containing every predicted protein for the organism.
    2. **Targeted Sequence Selection:** Next, the script filters this large proteome file to isolate only the proteins corresponding to the gene IDs in the provided list (final_gene_list.txt). It uses the grep command with the -Fwf options to find all FASTA headers that match the gene IDs. The -A 1 option then prints each matching header along with the single line of sequence that follows it. A final sed command is used to remove the separator lines (--) that grep inserts between matches, producing a clean, final FASTA file (target_proteins_for_analysis.fasta).

Script:

```
#!/usr/bin/env bash
# SCRIPT: Extracts protein sequences for a target list of genes.

set -euo pipefail

## CONFIGURATION ##
WORKSPACE_DIR="/data/training2/analisis_TFM_Bulida_Precoz/10_functional_annota
tion"
GENOME_FASTA="/data/training2/info/assemblies/BUL_cur_guided.v1.0.fasta"
ANNOTATION_GFF3="/data/training2/info/assemblies/Bulida_annotationRNAseq.gff3"
GENE_LIST_FILE="${WORKSPACE_DIR}/final_gene_list.txt"

# Output files
ALL_PROTEINS_FASTA="${WORKSPACE_DIR}/all_proteins.fasta"
TARGET_PROTEINS_FASTA="${WORKSPACE_DIR}/target_proteins_for_analysis.fasta"

# --- Step 1: Extract ALL protein sequences from the genome ---
if [ ! -s "${ALL_PROTEINS_FASTA}" ]; then # -s checks if file exists AND is
not empty
    echo "--- Extracting ALL protein sequences from the genome ---"
    gffread "${ANNOTATION_GFF3}" -g "${GENOME_FASTA}" -y
"${ALL_PROTEINS_FASTA}"
else
    echo "--- File with all proteins already exists. Skipping extraction. ---"
fi

# --- Step 2: Select the protein sequences for the genes of interest ---
echo "--- Selecting protein sequences for the target genes ---"
# Use grep to find the headers matching our gene list (-Fwf)
# and print the header line plus the sequence that follows (-A 1)
# This is more robust than seqtk subseq for this task.
grep -A 1 -Fwf "${GENE_LIST_FILE}" "${ALL_PROTEINS_FASTA}" >
"${TARGET_PROTEINS_FASTA}"
```

```
# Remove the "--" separator lines that grep adds between matches
sed -i '/^--$/d' "${TARGET_PROTEINS_FASTA}"

echo -e "\n### Process complete. ###"
# Verify that the output file now has content
COUNT=$(grep -c ">" "${TARGET_PROTEINS_FASTA}")
echo "Found and extracted ${COUNT} sequences."
echo "The file with the proteins to analyze is: ${TARGET_PROTEINS_FASTA}"
```

2_run_interproscan.sh

- **Purpose:** The purpose of this script is to perform a comprehensive functional analysis of the candidate protein sequences using InterProScan. This tool is a powerful software suite that scans protein sequences against a large collection of signature databases (such as Pfam, SUPERFAMILY, and CDD). The primary goal is to identify conserved protein domains and families, which provides critical insight into the potential biological function of the candidate genes. The script is configured to also retrieve associated Gene Ontology (GO) terms and pathway information for each protein.

- **Logic:** The script is designed to be robust and self-contained, first preparing the necessary environment and then executing the main analysis.

    1. **Database Indexing Pre-check:** The script begins with a crucial pre-flight check. It verifies if the HMMER-based databases required by InterProScan (like SuperFamily and PIRSF) have been properly indexed. If it detects that the binary index files (e.g., .h3f) are missing, it automatically runs the hmmpress command on the necessary library files. This step ensures that the local analysis will not fail due to an incomplete setup, a problem identified in previous runs.

    2. **Main InterProScan Execution:** The script then calls the main interproscan.sh command. It specifies the input FASTA file of target proteins (-i) and requests the output in a single, easy-to-parse tab-separated format (-f tsv). Key flags are included to enrich the analysis: --goterms and --pathways explicitly tell the program to include Gene Ontology and pathway annotations in the results. The --disable-precalc flag is used to force all analyses to run locally, ensuring reproducibility and avoiding potential issues with external lookup services. Finally, the -cpu flag is used to specify the number of threads to parallelize and speed up this computationally intensive task.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Runs InterProScan on the target protein sequences.

set -euo pipefail

# 1) Determine Conda and InterProScan installation paths
PREFIX="${CONDA_PREFIX}"
IPS_HOME="${PREFIX}/share/InterProScan"

# 2) Index HMMER libraries if the .h3* files are missing
for lib in \
  "data/superfamily/1.75/hmmlib_1.75" \
  "data/pirsf/3.10/sf_hmm_all"; do

  full_path="${IPS_HOME}/${lib}"
  if [[ -f "${full_path}" && ! -f "${full_path}.h3f" ]]; then
    echo "Indexing ${full_path} with hmmpress..."
    "${PREFIX}/bin/hmmpress" "${full_path}"
  fi
done

## CONFIGURATION ##
WORKSPACE_DIR="/data/training2/analisis_TFM_Bulida_Precoz/10_functional_annota
tion"
INPUT_FASTA="${WORKSPACE_DIR}/target_proteins_for_analysis.fasta"
OUTPUT_FILE_PREFIX="${WORKSPACE_DIR}/interproscan_results"
```

```
# Number of CPUs to use
CPU=8

echo "--- Starting InterProScan analysis ---"
echo "Input FASTA: ${INPUT_FASTA}"
echo "This may take several hours..."

# Run InterProScan with TSV output, GO terms, and pathway annotation
interproscan.sh \
    -i "${INPUT_FASTA}" \
    -f tsv \
    -o "${OUTPUT_FILE_PREFIX}.tsv" \
    --iprlookup \
    --goterms \
    --pathways \
    --disable-precalc \
    -cpu "${CPU}"

echo -e "\n### InterProScan analysis completed. ###"
echo "Results written to: ${OUTPUT_FILE_PREFIX}.tsv"
```

3_run_blast_annotation.sh

- **Purpose:** The purpose of this script is to perform a local protein BLAST (BLASTp) search to functionally characterize the candidate protein sequences. This step is essential for inferring the potential function of the uncharacterized candidate genes by identifying homologous proteins in a well-annotated reference proteome. For this analysis, the script uses the complete set of reference proteins from *Prunus armeniaca* (apricot) downloaded from the UniProt database, ensuring the most relevant possible comparison.
- **Logic:** The script automates a complete, multi-step BLAST workflow, from data acquisition to final result processing.
    1. **Dependency and Data Check:** The script first ensures that all necessary external tools (wget, makeblastdb, blastp) are available. It then checks if the *Prunus armeniaca* reference proteome has already been downloaded, and if not, it uses wget to fetch it from the UniProt database.
    2. **BLAST Database Creation:** Next, it checks for the existence of a BLAST database. If one is not found, it uses the makeblastdb command to convert the downloaded reference protein FASTA file into a properly formatted and indexed database, which is required for the BLAST search.
    3. **BLASTp Execution:** The script then executes the main blastp command. It takes the user's candidate proteins (target_proteins_for_analysis.fasta) as the query and searches them against the newly created *Prunus armeniaca* database. The search is configured to use a specific E-value threshold (1e-5) and a custom output format that provides key information for each hit, including sequence identity, alignment length, and the descriptive title of the subject protein.
    4. **Top-Hit Extraction:** Since BLASTp can report multiple hits for each query, the final stage of the script processes the full results table. It uses a combination of sort (to order the hits by bitscore, a measure of alignment quality) and awk to select only the single best hit for each query protein, creating a clean and summarized final output table.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Local BLASTp of Bulida Precoz proteins vs. Prunus armeniaca
reference proteome
# USAGE: ./blastp_prunus_local.sh

set -euo pipefail

#### 1) CONFIGURATION ####
WORKDIR="/data/training2/analisis_TFM_Bulida_Precoz/10_functional_annotation"
INPUT_FASTA="${WORKDIR}/target_proteins_for_analysis.fasta"

# Where we'll put the prunus proteome and BLAST db
REF_FASTA="${WORKDIR}/prunus_armeniaca_refprot.fasta"
DB_PREFIX="${WORKDIR}/prunus_armeniaca_db"

# Output
OUT_FULL="${WORKDIR}/blastp_prunus_armeniaca_local.tsv"
OUT_TOP1="${WORKDIR}/blastp_prunus_armeniaca_local_top1.tsv"

# BLAST parameters
EVALUE="1e-5"
THREADS=8
```

```
#### 2) CHECK DEPENDENCIES ####
for tool in wget makeblastdb blastp; do
  if ! command -v ${tool} &> /dev/null; then
    echo "ERROR: ${tool} not found; please install BLAST+ and wget" >&2
    exit 1
  fi
done

#### 3) DOWNLOAD Prunus armeniaca PROTEOME ####
if [ ! -s "${REF_FASTA}" ]; then
  echo "--- Fetching Prunus armeniaca proteome from UniProt ---"
  wget -O "${REF_FASTA}" \
    "https://rest.uniprot.org/uniprotkb/stream?query=organism_id:36596&format=
fasta"
else
  echo "--- Reference proteome already present: ${REF_FASTA} ---"
fi

#### 4) MAKE BLAST DATABASE ####
if [ ! -f "${DB_PREFIX}.pin" ]; then
  echo "--- Building BLAST database: ${DB_PREFIX} ---"
  makeblastdb \
    -in "${REF_FASTA}" \
    -dbtype prot \
    -out "${DB_PREFIX}"
else
  echo "--- BLAST DB already exists: ${DB_PREFIX}.* ---"
fi

#### 5) RUN BLASTp LOCALLY ####
echo "--- Running BLASTp (local) of ${INPUT_FASTA} vs ${DB_PREFIX} ---"
blastp \
  -query "${INPUT_FASTA}" \
  -db "${DB_PREFIX}" \
  -evalue "${EVALUE}" \
  -num_threads "${THREADS}" \
  -outfmt "6 qseqid sseqid pident length evalue bitscore stitle" \
  -out "${OUT_FULL}"

echo "Full BLASTp results saved to ${OUT_FULL}"

#### 6) EXTRACT TOP-HIT PER QUERY ####
echo "--- Extracting single best hit per query (highest bitscore) ---"
sort -k1,1 -k6,6nr "${OUT_FULL}" | awk '!seen[$1]++' > "${OUT_TOP1}"
echo "Top-hit table saved to ${OUT_TOP1}"

#### 7) DONE ####
echo "===== COMPLETE ====="
echo "Full results: ${OUT_FULL}"
echo "Best hits   : ${OUT_TOP1}"
```

4_annotate_with_GO.sh

- **Purpose:** The purpose of this script is to enrich the local BLASTp results by appending Gene Ontology (GO) terms. The initial BLAST output provides protein descriptions but lacks the standardized functional annotations necessary for downstream enrichment analysis. This script automates the process of querying the UniProt database to retrieve the corresponding GO terms for each BLAST hit and then integrates this information back into the main results table.
- **Logic:** The script performs a multi-stage data retrieval and integration workflow using a combination of awk, curl, and join.
    1. **Accession Extraction:** The script first uses awk to parse the full BLASTp output table. It extracts the UniProt accession number from the subject ID column for each hit, creating a clean, unique list of all accession numbers found.
    2. **GO Term Retrieval:** It then uses curl to query the UniProt REST API. The entire list of unique accession numbers is submitted in a single request to the /uniprotkb/accessions endpoint, specifically requesting the go_id field. The results are saved to a new mapping file.
    3. **Data Formatting:** The script then uses awk to process this mapping file, collapsing multiple GO terms for a single protein into one comma-separated line. Simultaneously, it prepares the original BLAST results table for merging.
    4. **Final Join:** Finally, it uses the Unix join command to merge the BLAST results with the GO term information based on the common UniProt accession number. This produces the final, comprehensive output table (blast_with_GO.tsv) that contains both the BLAST hit details and the associated GO annotations.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Annotate BLASTp hits with UniProt GO terms (accessions endpoint +
timeouts)

set -euo pipefail

#### 1) CONFIGURATION ####
WORKDIR="/data/training2/analisis_TFM_Bulida_Precoz/10_functional_annotation"
BLAST_IN="${WORKDIR}/blastp_prunus_armeniaca_local.tsv"
ACCESSIONS_LIST="${WORKDIR}/subject_accessions.list"
GO_MAPPING="${WORKDIR}/uniprot_GO_mapping.tsv"
GO_COLLAPSED="${WORKDIR}/uniprot_GO_mapping.collapsed.tsv"
TMP_BLAST2="${WORKDIR}/blast_for_join.tsv"
FINAL_OUT="${WORKDIR}/blast_with_GO.tsv"

#### 2) EXTRACT UNIQUE ACCESSIONS ####
awk '{
  acc=$2;
  sub(/^tr\|/,"",acc);
  sub(/\|.*$/,"",acc);
  print acc
}' "${BLAST_IN}" | sort -u > "${ACCESSIONS_LIST}"

num_acc=$(wc -l < "${ACCESSIONS_LIST}")
echo "Found ${num_acc} unique UniProt accessions in BLAST hits."

#### 3) FETCH GO IDs VIA ACCESSIONS ENDPOINT ####
accessions=$(paste -sd, "${ACCESSIONS_LIST}")
```

```
echo "Querying UniProt accession endpoint for GO_IDs…"

curl -s \
     --connect-timeout 10 \
     -m 30 \
     "https://rest.uniprot.org/uniprotkb/accessions?accessions=${accessions}&f
ields=accession,go_id&format=tsv" \
  | tail -n +2 > "${GO_MAPPING}"

if [ ! -s "${GO_MAPPING}" ]; then
  echo "No GO IDs returned. Your proteins may lack UniProt GO cross-
references."
  exit 0
fi
echo "Downloaded GO mapping to ${GO_MAPPING}"

#### 4) COLLAPSE MULTIPLE GO_IDs PER ACCESSION ####
awk 'BEGIN{FS=OFS="\t"}{
  acc=$1; gid=$2;
  arr[acc]=(arr[acc]?arr[acc]",":"")gid:gid)
}
END{
  for(a in arr) print a, arr[a]
}' "${GO_MAPPING}" > "${GO_COLLAPSED}"
echo "✓ Collapsed GO_IDs per accession into ${GO_COLLAPSED}"

#### 5) PREPARE BLAST TABLE FOR JOIN ####
awk 'BEGIN{FS=OFS="\t"}{
  acc=$2; sub(/^tr\|/,"",acc); sub(/\|.*$/,"",acc);
  printf("%s\t%s\t%s\t%s\t%s\t%s\t%s\n",
    $1, acc, $3, $4, $5, $6, $7)
}' "${BLAST_IN}" > "${TMP_BLAST2}"
echo "✓ Prepared BLAST join file ${TMP_BLAST2}"

#### 6) JOIN GO_IDs INTO BLAST HITS ####
echo -e "qseqid\taccession\tpident\tlength\tevalue\tbitscore\ttitle\tGO_IDs" >
"${FINAL_OUT}"
join -t $'\t' -1 2 -2 1 \
  <(sort -k2,2 "${TMP_BLAST2}") \
  <(sort -k1,1 "${GO_COLLAPSED}") \
  >> "${FINAL_OUT}"

echo "Annotation complete! See: ${FINAL_OUT}"
```

6.1 Functional_analysis.ipynb

- **Purpose:** This Jupyter Notebook serves as the primary tool for the descriptive functional analysis of the candidate genes. The purpose of this notebook is to load, process, and visualize the annotation data obtained from the previous BLASTp and InterProScan analyses. It is designed to transform the raw output tables into structured data frames and generate initial summary statistics and plots, such as identifying the most frequently occurring Gene Ontology (GO) terms associated with the candidate variants.
- **Logic:** The notebook executes a series of data manipulation and visualization steps in a sequential manner using the pandas and matplotlib libraries.
    1. **Data Loading:** The notebook begins by loading the two main annotation files: blast_with_GO.tsv, which contains the BLASTp hits and their associated GO IDs, and interproscan_results.tsv, containing the domain and signature annotations.
    2. **GO Term Processing:** It then processes the BLAST results table (df_blast). A key operation is the parsing of the semi-colon separated GO_IDs string; the script "explodes" this column to create a new data frame where each row represents a single protein-to-GO_ID association. It also extracts the clean protein descriptions from the title column.
    3. **Frequency Analysis and Visualization:** The script calculates the frequency of each unique GO term found among the candidate proteins and generates a bar plot to visualize the top 10 most frequent terms. This provides a first look at the most prominent functions within the candidate set.
    4. **Data Consolidation and Export:** Finally, the notebook merges the processed GO term data with the protein descriptions and saves several new, cleaned tables (exploded_go_terms.tsv, go_terms_wide.tsv, accession_descriptions.tsv) back to the disk for use in subsequent enrichment analyses.

You can find this script on the GitHub repository for this project.

5_prepare_for_GO_enrichment.sh

- **Purpose:** The purpose of this script is to prepare the final, essential input file required for the Gene Ontology (GO) enrichment analysis in R. The clusterProfiler package requires a specific "universe" file that maps every gene in the background set to its corresponding GO terms. This script takes the comprehensive annotation table generated previously and reformats it into the simple, two-column structure needed by clusterProfiler.
- **Logic:** The script uses a single awk command to process the exploded_go_terms.tsv file. awk reads the input table line by line and, for every line after the header (NR>1), it extracts the first column (the protein/gene identifier) and the third column (the GO term identifier). It then prints these two columns, separated by a tab, to a new output file named gene2go_background.tsv. This file serves as the TERM2GENE mapping in the subsequent R analysis script.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Prepares the necessary input files for the R enrichment analysis.

set -euo pipefail

## CONFIGURATION ##
WORKSPACE_DIR="/data/training2/analisis_TFM_Bulida_Precoz/10_functional_annota
tion"
# This is the file generated by your script 4_annotate_with_GO.sh
ANNOTATION_FILE="${WORKSPACE_DIR}/blast_annotation/exploded_go_terms.tsv"

# Output file: a two-column map of GENE_ID -> GO_TERM
GENE2GO_MAP_FILE="${WORKSPACE_DIR}/gene2go_background.tsv"

echo "--- Preparing gene-to-GO mapping file ---"

# We use awk to format the input into the two-column format required by
clusterProfiler
# We take the first column (Gene ID) and the third column (GO ID)
awk 'BEGIN{FS="\t"; OFS="\t"} NR>1 {print $1, $3}' "${ANNOTATION_FILE}" >
"${GENE2GO_MAP_FILE}"

echo "Preparation complete. The following files are ready for R:"
echo "1. Gene list of interest: ${WORKSPACE_DIR}/final_gene_list.txt"
echo "2. Background universe map: ${GENE2GO_MAP_FILE}"
```

create_full_go_universe.sh

- **Purpose:** The purpose of this script is to generate a comprehensive, genome-wide annotation file (gene2go_universe_full.tsv). This file serves as the statistical background or "universe" required for the Gene Ontology (GO) enrichment analysis performed by the R package clusterProfiler. To ensure statistical validity, the enrichment test must compare the small list of candidate genes against all annotated genes in the genome; this script is responsible for creating that complete background annotation map.

- **Logic:** The script executes a multi-stage bioinformatic pipeline to perform a homology-based annotation of the entire predicted proteome of the organism.

    1. **FASTA File Sanitization:** The script first performs a critical data cleaning step. It reads the initial protein FASTA file (all_proteins.fasta) and uses awk to remove any non-standard amino acid characters (such as * or X) that may have been generated by gffread from imperfect gene models. This produces a clean FASTA file (all_proteins.cleaned.fasta) that is compatible with downstream tools.

    2. **Whole-Proteome BLASTp:** It then runs blastp, using the entire cleaned proteome as a query against the *Prunus armeniaca* reference protein database. This step identifies the closest known homolog for every protein in the genome.

    3. **UniProt Accession Extraction:** The script parses the full BLASTp output to extract a unique list of all UniProt accession numbers corresponding to the identified homologs.

    4. **Batch GO Term Retrieval:** To handle the large number of accessions without exceeding command-line length limits, the script fetches the corresponding GO terms from the UniProt REST API in batches. It uses a for loop to process the accession list in chunks of 400, submitting a separate curl request for each batch and appending the results to a single mapping file.

    5. **Final Universe File Creation:** In the final step, the script uses a series of awk commands to join the initial BLAST results (mapping your internal gene IDs to UniProt accessions) with the downloaded GO term data (mapping UniProt accessions to GO IDs). This creates the definitive two-column gene2go_universe_full.tsv file, which maps every original protein ID in your genome to its inferred GO terms.

Script:

```
#!/usr/bin/env bash
# SCRIPT: Creates a comprehensive gene -> GO mapping file for the entire
proteome.

set -euo pipefail

#### CONFIGURATION ####
WORKDIR="/data/training2/analisis_TFM_Bulida_Precoz/10_functional_annotation"
ALL_PROTEINS_FASTA="${WORKDIR}/all_proteins.fasta"
ALL_PROTEINS_CLEANED_FASTA="${WORKDIR}/all_proteins.cleaned.fasta"
DB_PREFIX="${WORKDIR}/prunus_armeniaca_db"
UNIVERSE_FILE_OUT="${WORKDIR}/gene2go_universe_full.tsv"

# Temporary files
BLAST_OUT_FULL="${WORKDIR}/temp_blast_all_proteins.tsv"
ACCESSIONS_LIST_FULL="${WORKDIR}/temp_all_accessions.list"
GO_MAPPING_FULL="${WORKDIR}/temp_all_uniprot_go_mapping.tsv"

THREADS=8
BATCH_SIZE=400 # Number of accessions to query per request
```

```
#### SCRIPT LOGIC ####
echo "--- Creating Full GO Universe for Enrichment Analysis ---"

# --- Step 0: Clean the protein FASTA file ---
if [ ! -s "${ALL_PROTEINS_CLEANED_FASTA}" ]; then
    echo "Step 0: Cleaning the protein FASTA file..."
    awk '/^>/ {print; next} {gsub(/\*|X/,""); print}' "${ALL_PROTEINS_FASTA}"
> "${ALL_PROTEINS_CLEANED_FASTA}"
    echo "Cleaned FASTA created."
else
    echo "Step 0: Cleaned FASTA file already exists. Skipping."
fi

# --- Step 1: Run BLASTp ---
if [ ! -s "${BLAST_OUT_FULL}" ]; then
    echo "Step 1: Running BLASTp for all proteins..."
    blastp \
        -query "${ALL_PROTEINS_CLEANED_FASTA}" \
        -db "${DB_PREFIX}" \
        -evalue 1e-5 \
        -num_threads "${THREADS}" \
        -outfmt "6 qseqid sseqid" \
        -out "${BLAST_OUT_FULL}"
else
    echo "Step 1: BLASTp output already exists. Skipping."
fi

# --- Step 2: Extract UniProt accessions ---
if [ ! -s "${ACCESSIONS_LIST_FULL}" ]; then
    echo "Step 2: Extracting UniProt accessions..."
    awk '{ acc=$2; sub(/^tr\|/,"",acc); sub(/\|.*$/,"",acc); print acc }'
"${BLAST_OUT_FULL}" \
    | sort -u > "${ACCESSIONS_LIST_FULL}"
else
    echo "Step 2: Accession list already exists. Skipping."
fi

# --- Step 3: Fetch GO terms from UniProt in batches (THE FIX) ---
if [ ! -s "${GO_MAPPING_FULL}" ]; then
    echo "Step 3: Fetching GO terms from UniProt in batches..."
    # Create the output file so we can append to it
    touch "${GO_MAPPING_FULL}"

    # Read the accession list file line by line and process in batches
    mapfile -t accessions_array < "${ACCESSIONS_LIST_FULL}"
    total_accessions=${#accessions_array[@]}

    for (( i=0; i<total_accessions; i+=BATCH_SIZE )); do
        # Get a batch of accessions
        batch_array=("${accessions_array[@]:i:BATCH_SIZE}")
        batch_string=$(IFS=,; echo "${batch_array[*]}")

        echo "  - Fetching batch starting at accession #${i}..."

        # Run curl for the batch and append (>>) to the output file
        curl -s --connect-timeout 20 -m 60 \
```

```
            "https://rest.uniprot.org/uniprotkb/accessions?accessions=${batch
_string}&fields=accession,go_id&format=tsv" \
        | tail -n +2 >> "${GO_MAPPING_FULL}"

        sleep 1 # Be nice to the UniProt server and wait 1 second
    done
else
    echo "Step 3: GO mapping file already exists. Skipping."
fi

# --- Step 4: Create the final two-column gene2go file ---
if [ ! -s "${UNIVERSE_FILE_OUT}" ]; then
    echo "Step 4: Creating the final universe file..."
    awk 'BEGIN{FS=OFS="\t"} FNR==NR{go[$1]=$2; next} { acc=$2;
sub(/^tr\|/,"",acc); sub(/\|.*$/,"",acc); if(acc in go) print $1, go[acc] }' \
        <(awk 'BEGIN{FS=OFS="\t"}{split($2, go_list, "; "); for(i in go_list)
print $1, go_list[i]}' "${GO_MAPPING_FULL}") \
        "${BLAST_OUT_FULL}" \
    | awk 'BEGIN{FS=OFS="\t"}{split($2, go_list, ","); for(i in go_list) print
$1, go_list[i]}' \
    | sort -u > "${UNIVERSE_FILE_OUT}"
else
    echo "Step 4: Final universe file already exists. Skipping."
fi

echo -e "\n### Universe file creation workflow finished. ###"
echo "Check the final output at: ${UNIVERSE_FILE_OUT}"
```

6_run_GO_enrichment.Rmd

- **Purpose:** This R Markdown script performs the final and most important biological analysis of the project: a statistical Gene Ontology (GO) enrichment analysis. The purpose is to take the final list of high-confidence candidate genes and determine if any biological functions, molecular processes, or cellular components are significantly over-represented within this set compared to the entire genome. The script is designed to produce a self-contained HTML report that includes the full data processing, statistical results, and publication-quality visualizations, providing the key biological conclusions for the thesis.
- **Logic:** The notebook executes a complete and robust workflow for a custom GO enrichment analysis using the clusterProfiler package.
    1. **Environment Setup:** The first code chunk ensures a reproducible environment by checking for, installing, and loading all necessary R and Bioconductor packages, including clusterProfiler, dplyr for data manipulation, ggplot2 for plotting, and GO.db for annotation.
    2. **Data Loading and Preparation:** The script then loads the two essential input files: the list of candidate genes (final_gene_list.txt) and the comprehensive background file containing all gene-to-GO mappings for the organism (gene2go_universe_full.tsv). It processes this data to create the two key data frames required by clusterProfiler for a custom analysis: TERM2GENE (mapping GO terms to genes) and TERM2NAME (mapping GO IDs to their official, human-readable descriptions fetched from the GO.db database).
    3. **Iterative Enrichment Analysis:** The core of the analysis is performed in a loop that iterates through the three main GO ontologies: Biological Process (BP), Cellular Component (CC), and Molecular Function (MF). For each ontology, it first filters the background universe to use only the relevant terms. It then runs the statistical over-representation test using the enricher() function, comparing the user's list of genes against the corresponding background.
    4. **Results and Visualization:** Finally, for each of the three ontologies, the script checks if any statistically significant terms were found. If so, it generates and displays both a formatted table of the enriched terms using knitr::kable and a dotplot visualization, which clearly shows the most significant results, the number of genes involved, and their statistical p-value.

You can find this script on the GitHub repository for this project.

6_run_GO_enrichment_v2.Rmd

- **Purpose:** The purpose of this R Markdown script is to perform a descriptive functional analysis of the final candidate genes. It is designed to load the annotation data generated from previous BLAST and UniProt queries, process it into structured tables, and generate summary statistics and visualizations. The key outputs are a comprehensive summary table for each candidate gene and a frequency analysis of the most common Gene Ontology (GO) terms associated with the candidate set, providing an initial characterization of their potential biological roles.
- **Logic:** The script executes a multi-stage data processing and visualization workflow using the R packages dplyr for data manipulation and ggplot2 for plotting.
    1. **Environment Setup:** The script begins by ensuring all necessary R and Bioconductor packages (including clusterProfiler, dplyr, and GO.db) are installed and loaded into the session.
    2. **Data Loading:** It then loads the three primary data files: blast_with_GO.txt (containing the BLAST hits and associated GO IDs), accession_descriptions.txt (containing clean protein descriptions), and gene2go_background.txt (the full GO universe for the organism).
    3. **Candidate Summary Table Generation:** The script processes the main BLAST results table. It uses functions from dplyr and tidyr to parse the semi-colon separated GO_IDs field, creating a long-format data frame. It then joins this with the clean descriptions and groups the data by the original query ID (qseqid) to produce and display a comprehensive summary table (summary_df) for each candidate gene.
    4. **GO Term Frequency Analysis:** Finally, the script performs a descriptive frequency analysis. It counts the occurrences of each unique GO term across all candidate hits, joins these counts with official term names fetched from the GO.db package, and uses ggplot2 to generate a bar plot visualizing the top 10 most frequent GO terms found in the candidate set.

You can find this script on the GitHub repository for this project.