Contents

1 Variant Calling

1.1 Define Functions

1.1.1 Imports

```
#### Imports ####

import subprocess as sp
import os

gatk = '/home/lucas/Programs/gatk-4.1.9.0/gatk'
wd = '/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/'
os.chdir(wd)
```

1.1.2 Step -1: Create known sites of variation BED

Since we have no previous information on variants, we use an empty file.

```
### FUNTIONS ####

first = True
with open('./known_SNP.txt', 'r+') as infile:
    with open('known_SNPs.bed', 'w+') as outfile:
        for line in infile:
            if first:
                first = False
            else:
                linelist = line.strip().split('\t')
                chrompos = linelist[1]
                minorallel = linelist[3]
                #print(chrompos)
                chrom, pos = chrompos.split(': ')
                start = int(pos.replace(',', ''))
                outfile.write('\t'.join([chrom,
                                         str(start),
                                         str(start+1),
                                          minorallel])+'\n')

def index_feature_file(feat_file):
    cmd = '{} IndexFeatureFile -I {}' .format(gatk, feat_file)
    sp.call(cmd, shell = True)
```

```
index_feature_file('./empty.bed')
```

1.1.3 Step 0: Add read-groups

```
def AddOrReplaceReadGroups(bam):
    output = bam.replace('.bam', '_withRG.bam')
    name = bam.rsplit('.')[0]
    cmd = ("java -jar /home/lucas/Programs/picard.jar "
           "AddOrReplaceReadGroups "
           "-INPUT {} "
           "-OUTPUT {} "
           "-RGID group_{} "
           "-RGLB lib_{} "
           "-RGPL illumina "
           "-RGPU unit1 "
           "-RGSM {}_sample") .format(bam, output, name, name, name)
    sp.call(cmd, shell=True)
```

1.1.4 Step 1: Mark Duplicates (MarkDuplicates, samtools sort)

This second processing step is performed per-sample and consists of identifying read pairs that are likely to have originated from duplicates of the same original DNA fragments through some artifactual processes. These are considered to be non-independent observations, so the program tags all but a single read pair within each set of duplicates, causing the marked pairs to be ignored by default during the variant discovery process. At this stage the reads also need to be sorted into coordinate-order for the next step of the pre-processing. MarkDuplicatesSpark performs both the duplicate marking step and the sort step for this stage of pre-processing. This phase of the pipeline has historically been a performance bottleneck due to the large number of comparisons made between read pairs in a sample so MarkDuplicatesSpark utilizes Apache Spark in order to parallelize the process to better take advantage all available resources. This tool can be run locally even without access to a dedicated Spark cluster.

```
def mark_duplicates(bam):
    outfile = bam.replace('.bam', '_markedDuplicates.bam')
    mtrcsfile = bam.replace('.bam', '_metrics.txt')
    args = [gatk, bam, outfile, mtrcsfile]
    cmd = '{} MarkDuplicates -I {} -O {} -M {}' .format(*args)
```

```
    sp.call(cmd, shell=True)

    cmd = 'samtools sort {} -o {}' .format(outfile, outfile)
    sp.call(cmd, shell=True)
```

1.1.5 Step 2: Base Recalibration (BaseRecalibrator, ApplyRecalibration)

This third processing step is performed per-sample and consists of applying machine learning to detect and correct for patterns of systematic errors in the base quality scores, which are confidence scores emitted by the sequencer for each base. Base quality scores play an important role in weighing the evidence for or against possible variant alleles during the variant discovery process, so it's important to correct any systematic bias observed in the data. Biases can originate from biochemical processes during library preparation and sequencing, from manufacturing defects in the chips, or instrumentation defects in the sequencer. The recalibration procedure involves collecting covariate measurements from all base calls in the dataset, building a model from those statistics, and applying base quality adjustments to the dataset based on the resulting model. The initial statistics collection can be parallelized by scattering across genomic coordinates, typically by chromosome or batches of chromosomes but this can be broken down further to boost throughput if needed. Then the per-region statistics must be gathered into a single genome-wide model of covariation; this cannot be parallelized but it is computationally trivial, and therefore not a bottleneck. Finally, the recalibration rules derived from the model are applied to the original dataset to produce a recalibrated dataset. This is parallelized in the same way as the initial statistics collection, over genomic regions, then followed by a final file merge operation to produce a single analysis-ready file per sample.

```
def base_recalibration(bam):

    outfile = bam.replace('.bam', '_baserecal_table.table')
    known = './empty.bed'
    ref = './ref.fasta'
    args = [gatk, bam, ref, known, outfile]

    cmd = ('{} BaseRecalibrator '
           '-I {} -R {} '
           '--known-sites {} '
           '-O {}') .format(*args)
```

```
    sp.call(cmd, shell=True)

def applyBQSR(bam):

    outfile = bam.replace('.bam', '_BQSR.bam')
    recal_table = bam.replace('.bam', '_baserecal_table.table')
    ref = './ref.fasta'
    args = [gatk, bam, ref, recal_table, outfile]

    cmd = ('{} ApplyBQSR '
           '-I {} -R {} '
           '--bqsr-recal-file {} '
           '-O {}') .format(*args)

    sp.call(cmd, shell=True)
```

1.1.6 Step 3: Merge Bams (MergeSamFiles)

This tool is used for combining SAM and/or BAM files from different runs or read groups into a single file, similar to the \"merge\" function of Samtools (`http://www.htslib.org/doc/samtools.html`).

Note that to prevent errors in downstream processing, it is critical to identify/label read groups appropriately. If different samples contain identical read group IDs, this tool will avoid collisions by modifying the read group IDs to be unique. For more information about read groups, see the GATK Dictionary entry.

```
def mergeBams(*bams, out):
    nbams = len(bams)
    inputs = '-I {} '*nbams
    cmd = 'java -jar /home/lucas/Programs/picard.jar ' \
        'MergeSamFiles ' + \
        inputs .format(*bams) + \
        '-O {}.bam' .format(out)
    sp.call(cmd, shell=True)
```

1.1.7 Step 3: Call Variants (Haplotype caller)

Call germline SNPs and indels via local re-assembly of haplotypes

The HaplotypeCaller is capable of calling SNPs and indels simultaneously via local de-novo assembly of haplotypes in an active region. In other words, whenever the program encounters a region showing signs of variation, it discards the existing mapping information and completely reassembles the reads in that region. This allows the HaplotypeCaller to be more accurate when calling regions that are traditionally difficult to call, for example when they contain different types of variants close to each other. It also makes the HaplotypeCaller much better at calling indels than position-based callers like UnifiedGenotyper.

In the GVCF workflow used for scalable variant calling in DNA sequence data, HaplotypeCaller runs per-sample to generate an intermediate GVCF (not to be used in final analysis), which can then be used in GenotypeGVCFs for joint genotyping of multiple samples in a very efficient way. The GVCF workflow enables rapid incremental processing of samples as they roll off the sequencer, as well as scaling to very large cohort sizes (e.g. the 92K exomes of ExAC).

In addition, HaplotypeCaller is able to handle non-diploid organisms as well as pooled experiment data. Note however that the algorithms used to calculate variant likelihoods is not well suited to extreme allele frequencies (relative to ploidy) so its use is not recommended for somatic (cancer) variant discovery. For that purpose, use Mutect2 instead.

How HaplotypeCaller works

1. Define active regions

The program determines which regions of the genome it needs to operate on (active regions), based on the presence of evidence for variation.

1. Determine haplotypes by assembly of the active region

For each active region, the program builds a De Bruijn-like graph to reassemble the active region and identifies what are the possible haplotypes present in the data. The program then realigns each haplotype against the reference haplotype using the Smith-Waterman algorithm in order to identify potentially variant sites.

1. Determine likelihoods of the haplotypes given the read data

For each active region, the program performs a pairwise alignment of each read against each haplotype using the PairHMM algorithm. This produces a matrix of likelihoods of haplotypes given the read data. These likelihoods are then marginalized to obtain the likelihoods of alleles for each potentially variant site given the read data.

1. Assign sample genotypes

For each potentially variant site, the program applies Bayes' rule, using the likelihoods of alleles given the read data to calculate the likelihoods of each genotype per sample given the read data observed for that sample. The most likely genotype is then assigned to the sample.

Input

Input bam file(s) from which to make variant calls

Output

Either a VCF or GVCF file with raw, unfiltered SNP and indel calls. Regular VCFs must be filtered either by variant recalibration (Best Practice) or hard-filtering before use in downstream analyses. If using the GVCF workflow, the output is a GVCF file that must first be run through GenotypeGVCFs and then filtering before further analysis.

```
def call_variants(bam):
    outfile = bam.replace('.bam', '_variants.vcf')
    ref = './ref.fasta'
    args = [gatk, ref, bam, outfile]

    cmd = ('{} --java-options "-Xmx4g" HaplotypeCaller '
           '-R {} -I {} -O {} -ploidy 1') .format(*args)

    sp.call(cmd, shell=True)
```

1.1.8 Step 4: Annotate variants (VEP)

We will use `vep` from `https://www.ensembl.org/info/docs/tools/vep/index.html` `vep` will give us annotations on both the nearest gene to a variant and the genetic effects of it (synonymous/missense/$\text{stop}_{\text{codon}}$...). Since P.Falciparum is not among the available species we will have to use a custom annotation file. For `vep` to be able to use it, the GFF file must be chromosome sorted and tabix indexed and biotype annotations must be added. We choose vcf output to retain it's information (allele freq, read depth...).

1. Create custom GFF (for VEP)

```
import pybedtools as pb
import subprocess as sp
import os
```

```
outfld = "/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/"
os.chdir(outfld)

# Filter out lines that do not correspond to genes
gff = pb.BedTool("../Data/PlasmoDB-52_Pfalciparum3D7.gff")

types = [feat.fields[2] for feat in gff]
set(types)


# Add biotype info to transcripts (it is a VEP requisite)
biotypes = {
    "mRNA": "protein_coding",
    "ncRNA": "ncRNA",
    "rRNA": "rRNA",
    "snoRNA": "snoRNA",
    "snRNA": "snRNA",
    "tRNA": "tRNA",
    "five_prime_UTR": "five_prime_UTR",
    "three_prime_UTR": "three_prime_UTR",
    "pseudogenic_transcript": "pseudogenic_transcript"
}

def add_biotype(feature):
    if feature.fields[2] in biotypes.keys():
        feature.attrs["biotype"] = biotypes[feature.fields[2]]
    return(feature)

added_biotype = gff.each(add_biotype)

# Sort GFF

added_biotype.sort().saveas("PlDB-52_Pfalciparum3D7_vep.gff")

# Change type from "protein_coding_gene" to "gene"

types = ['ncRNA_gene', 'protein_coding_gene']

with open("PlDB-52_Pfalciparum3D7_vep.gff", 'r+') as infile:
    with open("PlDB-52_Pfalciparum3D7_vep_changetypes.gff", 'w+') as outfile:
```

```
            for line in infile:
                linelist = line.strip().split('\t')
                if linelist[2] in types:
                    linelist[2] = 'gene'
                outfile.write('\t'.join(linelist)+'\n')

    # Compress GFF
    cmd = "bgzip PlDB-52_Pfalciparum3D7_vep_changetypes.gff"
    sp.call(cmd, shell=True)

    # Tabix GFF
    cmd = "tabix -p gff PlDB-52_Pfalciparum3D7_vep_changetypes.gff.gz"
    sp.call(cmd, shell=True)
```

2. Run VEP

```
def call_VEP(vcf, gff, fasta):

    out = vcf.replace('.vcf', '_VEPannotated.txt')
    args = [vcf, out, gff, fasta]

    cmd = ("/home/lucas/Programs/ensembl-vep/vep "
           "-i {} "
           "-o {} "
           "--gff {} "
           "--fasta {} "
           "--force_overwrite "
           "--vcf") .format(*args)

    sp.call(cmd, shell=True)
```

1.2 Calls

```
#### CALLS ####

import os
import subprocess as sp

wd = '/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/'
os.chdir(wd)
```

```
gatk = '/home/lucas/Programs/gatk-4.1.9.0/gatk'
indir = '/home/lucas/ISGlobal/Projects/Phd_Project/ChIP_Seq/Bams/'

os.listdir(indir)

bams = ['1.2B_in_sort_q5.bam',
        '10G_in_sort_q5.bam',
        'A7K9_in_sort_q5.bam',
        'E5K9_in_sort_q5.bam',
        'B11_in_sort_q5.bam'
        #'NF54_in_renamed_q5_sort.bam',
        ]

for bam in bams:
    bam = indir+bam

    AddOrReplaceReadGroups(bam)
    bam = bam.replace('.bam', '_withRG.bam')

    mark_duplicates(bam)
    bam = bam.replace('.bam', '_markedDuplicates.bam')

    base_recalibration(bam)
    applyBQSR(bam)
    bam = bam.replace('.bam', '_BQSR.bam')


bamlist = [f for f in os.listdir(indir) if f.endswith('_withRG_markedDuplicates_BQSR.ba

os.chdir(indir)
mergeBams(*bamlist, out = 'merged_12B_10G_A7_E5_B11')

bam = 'merged_12B_10G_A7_E5_B11.bam'
sp.call('samtools index {}' .format(bam), shell=True)
call_variants(bam)

os.chdir(wd)
vcf = './merged_12B_10G_A7_E5_B11_variants.vcf'
gff = './PlDB-52_Pfalciparum3D7_vep_changetypes.gff.gz'
fasta = './ref.fasta'
```

```
call_VEP(vcf, gff, fasta)
```

1.3 Parsing and Filtering

1.3.1 Step 5: Parse VEP output and annotate

```
import pybedtools as pb
import pandas as pd
import numpy as np
import os
from itertools import chain

project_path = "/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/"
os.chdir(project_path)

vep = pb.BedTool("merged_12B_10G_A7_E5_B11_variants_VEPannotated.txt")
gff = pb.BedTool("PlDB-52_Pfalciparum3D7_vep_changetypes.gff.gz")

# Create dict for annotation (from GFF)
gff_gene = gff.filter(lambda x: x[2] in ["gene", "pseudogene"])

def getAnnot(gffentry):

    info = gffentry.fields[8].split(";")
    dinfo = {x.split('=')[0]:x.split('=')[1] for x in info}
    gid = dinfo['ID']
    anot = dinfo['description']
    return([gid, anot])

annot = {}
for entry in gff_gene:
    ga = getAnnot(entry)
    annot[ga[0]] = ga[1]

def getRatioDepth(GF):
    if len(GF) <2:
        rf = np.nan
        alt = np.nan
        ratio = np.nan
        dp = 0
```

```
    else:
        rf = int(GF[1].split(",")[0])
        alt = int(GF[1].split(",")[1])
        dp = rf+alt

        if dp == 0:
            ratio = np.nan
        else:
            ratio = round(rf / dp, 2)

    return(rf, alt, ratio, dp)

# Create parsed output

def parse_variant(variant):

    # Parse vcf info
    ref = variant.fields[3]
    alt = variant.fields[4]
    pos = variant.start
    chrom = variant.chrom

    v10G = variant.fields[9].split(":")
    v12B = variant.fields[10].split(":")
    vA7 = variant.fields[11].split(":")
    vB11 = variant.fields[12].split(":")
    vE5 = variant.fields[13].split(":")

    ref_count1, alt_count1, r1, d1 = getRatioDepth(v10G)
    ref_count2, alt_count2, r2, d2 = getRatioDepth(v12B)
    ref_count3, alt_count3, r3, d3 = getRatioDepth(vA7)
    ref_count4, alt_count4, r4, d4 = getRatioDepth(vB11)
    ref_count5, alt_count5, r5, d5 = getRatioDepth(vE5)

    parsed_vcf = [chrom, pos, ref, alt,
                  ref_count1, alt_count1, r1, d1,
                  ref_count2, alt_count2, r2, d2,
                  ref_count3, alt_count3, r3, d3,
                  ref_count4, alt_count4, r4, d4,
                  ref_count5, alt_count5, r5, d5,
```

```
                ]

    # Parse vep info
    info = {}
    for x in variant.fields[7].split(";"):
        feat = x.split("=")
        if len(feat) == 2:
            info[feat[0]] = feat[1]
        else:
            info[feat[0]] = ""

    vep_out = info["CSQ"].split(",")
    effects = [effect.split("|") for effect in vep_out]

    # Add annotation (from GFF)
    for effect in effects:
        gene = effect[4]
        if gene != "":
            gannot = annot[gene]
        else:
            gannot = ""
        effect.append(gannot)

    parsed_variant = [parsed_vcf + effect for effect in effects]

    return(parsed_variant)

# Create DF
colnames = ["Chrom", "Pos", "Ref", "Alt",
            "RefCount_10G", "AltCount_10G", "RefRatio_10G", "depth_10G",
            "RefCount_12B", "AltCount_12B", "RefRatio_12B", "depth_12B",
            "RefCount_A7", "AltCount_A7", "RefRatio_A7", "depth_A7",
            "RefCount_B11", "AltCount_B11", "RefRatio_B11", "depth_B11",
            "RefCount_E5", "AltCount_E5", "RefRatio_E5", "depth_E5",

            "Allele",
            "Consequence",
            "IMPACT",
            "SYMBOL",
            "Gene",
```

```
            "Feature_type",
            "Feature",
            "BIOTYPE",
            "EXON",
            "INTRON",
            "HGVSc",
            "HGVSp",
            "cDNA_position",
            "CDS_position",
            "Protein_position",
            "Amino_acids",
            "Codons",
            "Existing_variation",
            "DISTANCE",
            "STRAND",
            "FLAGS",
            "SYMBOL_SOURCE",
            "HGNC_ID",
            "SOURCE",
            "PlDB-52_Pfalciparum3D7_vep.gff.gz",

            "Annot"]


parsed = [parse_variant(var) for var in vep]
flat = list(chain.from_iterable(parsed))
var_df = pd.DataFrame.from_records(flat, columns=colnames)

var_df.to_csv("parsed_variants_new.tsv", sep = '\t')
```

1.3.2 Step 6: Subset Variants DF (dplyr)

```
library(tidyverse)

wd <- '/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/'
setwd(wd)

variants <- read_tsv('parsed_variants_new.tsv') %>%
  mutate(Var_id = paste0('Variant_', `...1`)) %>%
  select(Var_id, everything(), `...1`)
```

```
parse_variants_bystrain <- function(strain, depth_filter, refratio_filter, impact_filte
  depthcol <- paste0('depth_', strain)
  ratiocol <- paste0('RefRatio_', strain)
  outname <- paste0('./Parsed_by_Strain/',
                    strain,
                    '_variants_depth_', depth_filter,
                    '_refratio_', refratio_filter,
                    '_impactfilter_', impact_filter,
                    '.tsv'
                    )

  if (impact_filter){
    variants <- variants %>%
      filter(IMPACT == 'HIGH')
  }

  variants %>%
    filter(get(depthcol) >= depth_filter &
           get(ratiocol) <= refratio_filter) %>%
    select(Var_id, contains(strain), Gene, Annot, Consequence,
           Chrom, Pos, Ref, Alt, everything()) %>%
    mutate(Annot = gsub('\"', '', Annot)) %>%
    write_tsv(outname)
}

#### Parse variants per strain

depth_filter <- 20
refratio_filter <- 0.5
impact_filter <- F

strains <- c('12B', '10G', 'A7', 'E5', 'B11')
for (strain in strains){
  parse_variants_bystrain(
    strain, depth_filter, refratio_filter, impact_filter
  )
}

#### Parse variants, all strains together
```

```
## depth_filter <- 20
## refratio_filter <- 0.5
## impact_filter <- F

## outname <- paste0(
##   'allstrains_variants_depth_', depth_filter,
##   '_refratio_', refratio_filter,
##   '_impactfilter_', impact_filter,
##   '.tsv'
## )

## filtered_vars <- variants
## if (impact_filter){
##   filtered_vars <- variants %>%
##     filter(IMPACT == 'HIGH')
## }

## filtered_vars <- variants %>%
##   filter(
##   (RefRatio_12B <= refratio_filter & depth_12B >= depth_filter) |
##   (RefRatio_10G <= refratio_filter & depth_10G >= depth_filter) |
##   (RefRatio_A7 <= refratio_filter & depth_A7 >= depth_filter) |
##   (RefRatio_E5 <= refratio_filter & depth_E5 >= depth_filter) |
##   (RefRatio_B11 <= refratio_filter & depth_B11 >= depth_filter)
##   ) %>%
##   select(Chrom, Pos, Ref, Alt, everything()) %>%
##   mutate(Annot = gsub('\"', '', Annot)) %>%
##   write_tsv(outname)
```

1.3.3 Step 7: Collapse variants and keep only nearest annotation per SNP

```
import os
import pybedtools as pb
from collections import defaultdict

## FUNCTIONS

## Get all 'Consequences'
```

```
wd = '/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/'
os.chdir(wd)
raw_vcf = 'parsed_variants.tsv'

firstline = True
consequences = []
cds_consequences = []
with open(raw_vcf, 'r+') as infile:
    for line in infile:
        ll = line.strip().split('\t')
        if firstline:
            colnames = ll
            firstline = False
        else:
            ll = line.strip().split('\t')[1:] #Skip first col (index)
            ld = {k:v for k, v in zip(colnames, ll)}
            consequences.append(ld['Consequence'])
            if ld.get('DISTANCE') == '':
                cds_consequences.append(ld['Consequence'])
colnames
set(consequences)
set(cds_consequences)
cds_vars = set(cds_consequences)
non_cds_vars = set(consequences) - set(cds_consequences)

## Parse variants

def variants_to_dict(var_file):
    variants = {}
    firstline = True
    i = 1
    with open(var_file, 'r+') as infile:
        for line in infile:
            ll = line.strip().split('\t')
            if firstline:
                colnames = ll
                firstline = False
            else:
                ld = {k:[v] for k, v in zip(colnames, ll)}
```

```
                var_id = '_'.join(ld['Chrom']+ld['Pos']+ld['Ref']+ld['Alt'])
                if var_id not in variants.keys():
                    variants[var_id] = ld
                    variants[var_id]['Unique_ID'] = f'Variant_{i}'
                    i += 1
                else:
                    ld = {k:v for k, v in zip(colnames, ll)}
                    for k, v in ld.items():
                        variants[var_id][k].append(v)
    return(variants)

## Filter variants


def filter_variants(var_dict, out_file):
    with open(out_file, 'w+') as outfile:

        ## Header
        outfile.write('Var_ID\t'+'\t'.join(colnames)+'\n')

        for k, v in var_dict.items():
            lmask = [vc in cds_vars for vc in v['Consequence']]
            true_idx = [i for i, x in enumerate(lmask) if x]
            if any(lmask):
                for idx in true_idx:
                    row = []
                    for cn in colnames:
                        row.append(v[cn][idx].replace('\"', ''))
                    outfile.write(v['Unique_ID']+'\t'+'\t'.join(row)+'\n')

            elif v['Consequence'] == ['intergenic_variant']:
                row = [v[cn][0].replace('\"', '') for cn in colnames]
                outfile.write(v['Unique_ID']+'\t'+'\t'.join(row)+'\n')

            elif 'upstream_gene_variant' in v['Consequence'] or 'downstream_gene_varia
                if 'upstream_gene_variant' in v['Consequence']:
                    up_mask = [vc == 'upstream_gene_variant' for vc in v['Consequence'
                    true_idx = [i for i, x in enumerate(up_mask) if x]
                    to_sort_up = [int(v['DISTANCE'][idx]) for idx in true_idx]
                    for idx, val in enumerate(v['DISTANCE']):
```

```
                        if int(val) == min(to_sort_up):
                            up_idx = idx
                    row = [v[cn][up_idx].replace('\"', '') for cn in colnames]
                    outfile.write(v['Unique_ID']+'\t'+'\t'.join(row)+'\n')

                if 'downstream_gene_variant' in v['Consequence']:
                    down_mask = [vc == 'downstream_gene_variant' for vc in v['Conseque
                    true_idx = [i for i, x in enumerate(down_mask) if x]
                    to_sort_down = [int(v['DISTANCE'][idx]) for idx in true_idx]
                    for idx, val in enumerate(v['DISTANCE']):
                        if int(val) == min(to_sort_down):
                            down_idx = idx
                    row = [v[cn][down_idx].replace('\"', '') for cn in colnames]
                    outfile.write(v['Unique_ID']+'\t'+'\t'.join(row)+'\n')

## CALLS

wd = '/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/'
os.chdir(wd)

var_fld = './Parsed_by_Strain/'
var_fls = [f for f in os.listdir(var_fld) if f.endswith('FALSE.tsv')]

for f in var_fls:
    vars = variants_to_dict(var_fld+f)
    filter_variants(vars, var_fld+f.replace('.tsv', '_nearest_only.tsv'))

# var_file = 'allstrains_variants_depth_20_refratio_0.5_impactfilter_FALSE.tsv'
# filter_variants(
#     variants_to_dict(var_file),
#     var_file.replace('.tsv', '_nearest_only.tsv')
# )
```

1.3.4 Step 8: Remove variants that overlap deleted regions

```
import os
import pybedtools as pb
from collections import defaultdict

## FUNTIONS
```

```
def filter_by_deletions(del_file, vars_file):

    dd = pb.BedTool(del_file)
    dd_del = dd.filter(lambda x: 'deletion' in x.fields[3])

    dels = defaultdict(list)
    for feat in dd_del:
        dels[feat.chrom ].append((feat.start, feat.stop))

    firstline = True
    out = vars_file.replace('.tsv', '_deletions_filtered.tsv')
    with open(vars_file, 'r+') as infile:
        with open(out, 'w+') as outfile:
            for line in infile:
                if firstline:
                    firstline = False
                    outfile.write(line)
                else:
                    linelist = line.split('\t')
                    chr = linelist[8]
                    pos = int(linelist[9])
                    del_regions = dels[chr]
                    in_del = [pos > _del[0] and pos < _del[1] for _del in del_regions]
                    if not any (in_del):
                            outfile.write('\t'.join(linelist))


## CALLS

wd = '/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/'
os.chdir(wd)

dupl_del_dir = '../Paper/Paper_Analysis/Data_Files/Duplication_Deletion_Regions_Mean_Se
v_dir = './Parsed_by_Strain/'

dd_files = [    '1.2B_in_sort_q5_noDup_rpkm_normInput_bs10_smth200_bymean_dupl_del_fact
            ]

v_files = [
```

```
    '12B_variants_depth_20_refratio_0.5_impactfilter_FALSE_nearest_only.tsv',
    '10G_variants_depth_20_refratio_0.5_impactfilter_FALSE_nearest_only.tsv',
    'A7_variants_depth_20_refratio_0.5_impactfilter_FALSE_nearest_only.tsv',
    'B11_variants_depth_20_refratio_0.5_impactfilter_FALSE_nearest_only.tsv',
    'E5_variants_depth_20_refratio_0.5_impactfilter_FALSE_nearest_only.tsv'
]

for d, v in zip(dd_files, v_files):
    filter_by_deletions(dupl_del_dir+d, v_dir+v)
```

1.3.5 Step 9: Merge Strains and drop columns

```
library(tidyverse)

wd <- '/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/Parsed_by_Strain/'
setwd(wd)
var_files <- list.files(wd, pattern = '_deletions_filtered.tsv')
df_names <- sapply(var_files, function(x) str_split(x, '_')[[1]][1])
var_dfs <- lapply(var_files, function(x) read_tsv(x) %>% select(-Var_ID))
names(var_dfs) <- df_names


colnames(var_dfs[[1]])

all_vars <- var_dfs %>%
  reduce(bind_rows) %>%
  distinct() %>%
  select(
    -HGVSc,
    -HGVSp,
    -Existing_variation,
    -FLAGS,
    -SYMBOL_SOURCE,
    -HGNC_ID,
    -SOURCE,
    -`PlDB-52_Pfalciparum3D7_vep.gff.gz`
  ) %>%
  select(
    Chrom, Pos, Ref, Alt,
    Gene, SYMBOL, Annot,
```

```
    Consequence, IMPACT,
    everything()
  ) %>%
  mutate(Annot = gsub('+', ' ', Annot, fixed = T)) %>%
  write_tsv('all_strains_merged_selected_cols.tsv')
```

1.3.6 Step 10: Column filtering for final table

```
library(tidyverse)

wd <- '/mnt/Disc4T/Projects/PhD_Project/Variant_Calling/Parsed_by_Strain/'
setwd(wd)

var_files <- list.files(wd, pattern = 'filtered.tsv')

filter_table <- function(vars_file){
  vars_file %>%
    read_tsv() %>%
    select(
      -HGVSc,
      -HGVSp,
      -Existing_variation,
      -FLAGS,
      -SYMBOL_SOURCE,
      -HGNC_ID,
      -SOURCE,
      -`PlDB-52_Pfalciparum3D7_vep.gff.gz`
    ) %>%
    select(Var_ID, Chrom, Pos, Ref, Alt, Gene, Annot, Consequence, everything()) %>%
    mutate(Annot = gsub('+', ' ', Annot, fixed = T)) %>%
    write_tsv(gsub('.tsv', '_col_filter.tsv', vars_file, fixed = T))
}

for (f in var_files){filter_table(f)}
```