FinalProjPlink1ipynb

March 11, 2024

[ ]:
```
#!/usr/bin/env python
# coding: utf-8




import logging
import os
from multiprocessing import Pool
import pandas as pd
import numpy as np
import gzip
from io import StringIO
from tqdm import tqdm
import statsmodels.api as sm
import argparse

# Setup argparse for command line arguments
parser = argparse.ArgumentParser(description='Run GWAS analysis')
parser.add_argument('--vcf', type=str, help='Path to VCF file', required=True)
parser.add_argument('--pheno', type=str, help='Path to phenotype file',␣
↪required=True)
parser.add_argument('--out', type=str, help='Output prefix for result files',␣
↪required=True)


args = parser.parse_args()





# Use the arguments
vcf_path = args.vcf
pheno_path = args.pheno
output_prefix = args.out


print("Arguments parsed successfully.")
```

```
# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
 ↪%(message)s')

def log_exception(exception):
    logging.error(f"Error: {exception}", exc_info=True)



# # Data Preprocessing



#Function to parse the vcf file and save to a datafrae
def parse_vcf(vcf_path):
    """
    Parses the VCF file to extract SNP information, with progress indication.

    Parameters:
    vcf_path (str): Path to the VCF file.

    Returns:
    pd.DataFrame: DataFrame with SNP information, one row per SNP.
    """
    print(f"Parsing VCF file from {vcf_path}")
    # Determine if the file is compressed and choose the appropriate opener
    if vcf_path.endswith('.gz'):
        opener = gzip.open
    else:
        opener = open

    # Read the file and filter out the header lines
    with opener(vcf_path, 'rt') as f:

        lines = [l for l in tqdm(f, desc="Reading VCF")]
        data_lines = [l for l in lines if not l.startswith('##')]

    # Create a DataFrame from the filtered lines
    vcf_df = pd.read_csv(StringIO(''.join(data_lines)), delimiter='\t',␣
 ↪dtype={'#CHROM': str, 'POS': int, 'ID': str, 'REF': str, 'ALT': str, 'QUAL':␣
 ↪str, 'FILTER': str, 'INFO': str})
    vcf_df.rename(columns={'#CHROM': 'CHROM'}, inplace=True)
    print("VCF parsing complete.")
    return vcf_df
```

```python
#Parsing the vcf file ps3_gwas.vcf.gz
"""
Note: this function can take quite a while to run,
and can be skipped for now if following dataframes are imported
"""

vcfdf = parse_vcf(vcf_path)
print(f"VCF file parsed. DataFrame shape: {vcfdf.shape}")





#Columns from vcfdf
columns = [
    'CHROM', 'POS', 'ID', 'REF', 'ALT', 'QUAL', 'FILTER', 'INFO', 'FORMAT',␣
 ↪'NA06984', 'NA06989', 'NA12878', 'NA18489',
    'NA18504', 'NA18511', 'NA18516', 'NA18523', 'NA18908', 'NA18910',␣
 ↪'NA18915', 'NA18934', 'NA11832', 'NA11894', 'NA11919',
    'NA11933', 'NA11995', 'NA12006', 'NA12044', 'NA12234', 'NA12272',␣
 ↪'NA12342', 'NA12347', 'NA12400', 'NA12760', 'NA11829',
    'NA12777', 'NA11831', 'NA12828', 'NA11843', 'NA12830', 'NA11881',␣
 ↪'NA12842', 'NA11893', 'NA12873', 'NA11918', 'NA11920',
    'NA11932', 'NA11994', 'NA12005', 'NA12889', 'NA18488', 'NA19095',␣
 ↪'NA18508', 'NA18510', 'NA18522', 'NA18864', 'NA18871',
    'NA18876', 'NA12776', 'NA12815', 'NA12827', 'NA12872', 'NA12043',␣
 ↪'NA12144', 'NA12156', 'NA19153', 'NA19160', 'NA12283',
    'NA19172', 'NA12341', 'NA19184', 'NA07037', 'NA19189', 'NA07051',␣
 ↪'NA07056', 'NA07347', 'NA19204', 'NA19209', 'NA18856',
    'NA18868', 'NA18870', 'NA19099', 'NA19222', 'NA19239', 'NA19223',␣
 ↪'NA19235', 'NA19247', 'NA18907', 'NA18933', 'NA19108',
    'NA19141', 'NA19146', 'NA19152', 'NA19171', 'NA19190', 'NA19210',␣
 ↪'NA19102', 'NA19107', 'NA19114', 'NA19119', 'NA19121',
    'NA19138', 'NA12273', 'NA12348', 'NA12413', 'NA12716', 'NA12761',␣
 ↪'NA12778', 'NA12812', 'NA12829', 'NA12843', 'NA12155',
    'NA12874', 'NA12249', 'NA12275', 'NA12282', 'NA12287', 'NA06985',␣
 ↪'NA12340', 'NA12383', 'NA12489', 'NA10851', 'NA11830',
    'NA10847', 'NA11892', 'NA11931', 'NA11840', 'NA12004', 'NA12045',␣
 ↪'NA06994', 'NA07000', 'NA07048', 'NA18853', 'NA18916',
    'NA18923', 'NA19096', 'NA19093', 'NA18505', 'NA19098', 'NA18517',␣
 ↪'NA12890', 'NA18499', 'NA18502', 'NA18507', 'NA18519',
    'NA19116', 'NA19130', 'NA19147', 'NA19159', 'NA18858', 'NA18865',␣
 ↪'NA18877', 'NA12718', 'NA18909', 'NA12749', 'NA12751',
   'NA19248', 'NA12763', 'NA12775', 'NA12814', 'NA18879', 'NA18881',␣
 ↪'NA19185', 'NA19197', 'NA19113', 'NA19200', 'NA19118',
   'NA19236', 'NA19137', 'NA19144', 'NA19149', 'NA19175', 'NA19207',␣
 ↪'NA19214', 'NA18867', 'NA18874', 'NA19238', 'NA19257',
   'NA07357', 'NA06986', 'NA18486', 'NA18498', 'NA18501', 'NA18520',␣
 ↪'NA19092', 'NA12154', 'NA12286', 'NA12399', 'NA12414',
   'NA12546', 'NA12717', 'NA12748', 'NA12750', 'NA12762', 'NA12813',␣
 ↪'NA18912', 'NA18917', 'NA18924', 'NA11930', 'NA11992',
  ␣
 ↪'NA12003','NA12046','NA12058','NA18861','NA18873','NA18878','NA19256','NA19198','NA19201','
 'NA19117','NA19129','NA19131','NA19143']




def calculate_allele_counts_and_maf(row):
    try:
        alleles = row[9:].str.extractall(r'(\d)')[0]
        allele_counts = alleles.value_counts()
        total_alleles = 2 * len(row[9:])
        maf = allele_counts.min() / total_alleles if not allele_counts.empty␣
 ↪else np.nan
        return maf
    except Exception as e:
        log_exception(e)
        return np.nan

def parallelize_dataframe(df, func, num_partitions, num_cores):
    try:
        df_split = np.array_split(df, num_partitions)
        pool = Pool(num_cores)

        # Using tqdm in pool.map for progress tracking
        result_list = list(tqdm(pool.imap(func, df_split),␣
 ↪total=num_partitions))
        df = pd.concat(result_list)

        pool.close()
        pool.join()
        return df
    except Exception as e:
        log_exception(e)
        return pd.DataFrame()
```

```
def process_dataframe_slice(df_slice):
    try:
        print("Calculating MAF for a dataframe slice...")
        mafs = df_slice.apply(calculate_allele_counts_and_maf, axis=1)
        df_slice['MAF'] = mafs
        return df_slice
    except Exception as e:
        log_exception(e)
        return pd.DataFrame()

# Assuming vcfdf is your DataFrame after parsing the VCF file



if __name__ == '__main__':
    # Apply parallel processing to calculate MAF and filter
    num_partitions = 10  # Number of partitions to split dataframe
    num_cores = min(4, os.cpu_count() - 1 or 1)
    logging.info(f"Using {num_cores} cores for multiprocessing.")
    print("Starting parallel MAF calculation...")
    result_df = parallelize_dataframe(vcfdf, process_dataframe_slice,␣
 ↪num_partitions, num_cores)

    print(result_df.head())
    print("Parallel MAF calculation completed.")
    filtered_df = result_df[result_df['MAF'] >= 0.05]



#Dropping the first column from the DataFrame
print("Dropping first col in filtered_df")
filtered_df = filtered_df.drop(columns=[filtered_df.columns[0]])



print("Dropping columns that arent SNPs")
columns_to_drop = ['CHROM', 'POS', 'REF', 'ALT', 'QUAL', 'FILTER', 'INFO',␣
 ↪'FORMAT']
filtered_df = filtered_df.drop([col for col in columns_to_drop if col in␣
 ↪filtered_df.columns], axis=1)
print("Non-SNP columns dropped.")




#Reading in phenotype information from ps3_gwas.phen
```

```
phenotype_df = pd.read_csv(pheno_path, sep='\t', header=None,␣
 ↪names=['SampleID', 'PhenotypeValue'])
print("Phenotype information loaded.")



print("Transposing filtered_df to have Sample IDs as the rows")
genotype_transposed = filtered_df.set_index('ID').transpose()



print("Merging the transposed genotype dataframe with the phenotype dataframe␣
 ↪on SampleID")
merged_df = genotype_transposed.merge(phenotype_df, left_index=True,␣
 ↪right_on='SampleID')

sample_col = merged_df.index
print("Merged phenotype_df and  genotype_transposed")
# Setting index to 'SampleID'
merged_df.set_index('SampleID', inplace=True)





#There were some columns from the parse_vcf function that should not have been␣
 ↪included
#(they did not represent SNPs), so we drop them
rs_columns = merged_df.columns[merged_df.columns.str.startswith('rs')]

# Drop columns that do not start with 'rs'
merged_df = merged_df[rs_columns]



# Function to drop duplicate columns, keeping the first
def drop_duplicate_columns(df):
    return df.loc[:, ~df.columns.duplicated()]

#Dropping duplicate columns
merged_df = drop_duplicate_columns(merged_df)




"""
Note: this function can take quite a while to run,
and can be skipped for now if following dataframes are imported
"""
#Batching the dataframe into 4 sections to ensure RAM is not exceeded

#Making sure all columns are strings
merged_df = merged_df.astype(str)

# Calculating the quarter point
quarter_point = len(merged_df.columns) // 4

# List to store processed columns
print("Processing 1st quarter")

processed_columns_first_quarter = []

# Processing the first quarter of the columns
for column in tqdm(merged_df.columns[:quarter_point], desc="Processing First␣
 ↪Quarter"):
    # Split, convert to integers, and sum
    processed_column = merged_df[column].str.split('|', expand=True).
 ↪astype(int).sum(axis=1)
    processed_columns_first_quarter.append(processed_column)

#Save as df
numeric_df_first_quarter = pd.concat(processed_columns_first_quarter, axis=1)
numeric_df_first_quarter.columns = merged_df.columns[:quarter_point]

#Create a copy to defragment
numeric_df_first_quarter = numeric_df_first_quarter.copy()

numeric_df_first_quarter['Sample ID'] = sample_col




"""
Note: this function can take quite a while to run,
and can be skipped for now if following dataframes are imported
"""

# List to store processed columns
print("Processing 2nd quarter")

processed_columns_second_quarter = []

# Calculating the midpoint, avoiding 'PhenotypeValue' if it's the last column
midpoint = len(merged_df.columns) // 2
```

```
if 'PhenotypeValue' in merged_df.columns[-1]:
    midpoint = (len(merged_df.columns) - 1) // 2

# Processing the second quarter of the columns (from quarter_point to midpoint)
for column in tqdm(merged_df.columns[quarter_point:midpoint], desc="Processing␣
  ↪Second Quarter"):
    # Split, convert to integers, and sum
    processed_column = merged_df[column].str.split('|', expand=True).
  ↪astype(int).sum(axis=1)
    processed_columns_second_quarter.append(processed_column)

# Concatenate into a new dataframe for the second quarter
numeric_df_second_quarter = pd.concat(processed_columns_second_quarter, axis=1)
numeric_df_second_quarter.columns = merged_df.columns[quarter_point:midpoint]

numeric_df_second_quarter = numeric_df_second_quarter.copy()

numeric_df_second_quarter['Sample ID'] = sample_col




"""
Note: this function can take quite a while to run,
and can be skipped for now if following dataframes are imported
"""

# List to store processed columns for the 3rd quarter
print("Processing 3rd quarter")
processed_columns_third_quarter = []

# Processing the 3rd quarter of the columns
for column in tqdm(merged_df.columns[quarter_point*2:quarter_point*3],␣
  ↪desc="Processing Third Quarter"):
    # Split, convert to integers, and sum
    processed_column = merged_df[column].str.split('|', expand=True).
  ↪astype(int).sum(axis=1)
    processed_columns_third_quarter.append(processed_column)

# Save as df
numeric_df_third_quarter = pd.concat(processed_columns_third_quarter, axis=1)
numeric_df_third_quarter.columns = merged_df.columns[quarter_point*2:
  ↪quarter_point*3]

# Create a copy to defragment
numeric_df_third_quarter = numeric_df_third_quarter.copy()
```

```
numeric_df_third_quarter['Sample ID'] = sample_col




"""
Note: this function can take quite a while to run,
and can be skipped for now if following dataframes are imported
"""

# List to store processed columns for the 4th quarter
print("Processing 4th quarter")

processed_columns_fourth_quarter = []

# Processing the 4th quarter of the columns
for column in tqdm(merged_df.columns[quarter_point*3:], desc="Processing Fourth␣
 ↪Quarter"):
    # Split, convert to integers, and sum
    processed_column = merged_df[column].str.split('|', expand=True).
 ↪astype(int).sum(axis=1)
    processed_columns_fourth_quarter.append(processed_column)

# Save as df
numeric_df_fourth_quarter = pd.concat(processed_columns_fourth_quarter, axis=1)
numeric_df_fourth_quarter.columns = merged_df.columns[quarter_point*3:]

# Create a copy to defragment
numeric_df_fourth_quarter = numeric_df_fourth_quarter.copy()

numeric_df_fourth_quarter['Sample ID'] = sample_col


print("Finished processing entire df!")

"""
Note: reading in the csvs can take quite a while to run,
and can be skipped for now if the final dataframes are imported
"""

df1 = numeric_df_first_quarter
df2 = numeric_df_second_quarter
df3 = numeric_df_third_quarter
df4 = numeric_df_fourth_quarter

#Save to csv
print("Saving df1 to csv")
```

```python
df1.to_csv(f"{output_prefix}_df1.csv", index=False)
print("Saving df2 to csv")
df2.to_csv(f"{output_prefix}_df2.csv", index=False)
print("Saving df3 to csv")
df3.to_csv(f"{output_prefix}_df3.csv", index=False)
print("Saving df4 to csv")
df4.to_csv(f"{output_prefix}_df4.csv", index=False)

print("Intermediate DataFrames saved.")
```