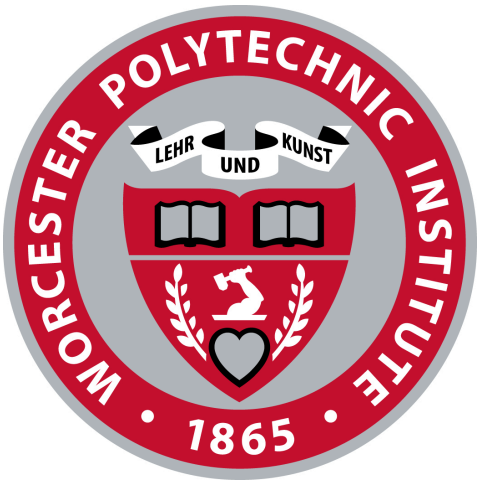
Extension of a gene passed p value calculations to distributed computing



A Major Qualifying Project Report

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science by

Taisiia Yakovenko

1 May 2024

Worcester, MA

Proposal Submitted to:

Professor Zhang Wu

1. Introduction

Amyotrophic lateral sclerosis (ALS) is a rare neurological disease that affects motor function of affected individuals. The disease quickly progresses upon diagnosis which is normally diagnosed between the ages of 55 and 75. There is currently no known cure for it and there is very little that medical professionals can do to delay the rapid progression of the disease or curb some of its side effects. The genetic research in the past years aims to find genetic markers for the disease among affected individuals to set stepping stones for its potential cure. The overarching goal of the project is to extend the implementation of gene based p value computation to handle large genetic data sets and aid in determining markers for ALS.

In order to implement modified GFisher statistic calculation for a single gene, the project utilizes an existing R code base which is then transcribed into a Python code base for singular machine processing and cluster processing. The main motivation behind rewriting the R library into Python is to create more compatibility with the Hail data processing library and provide more implementation for future use. The algorithm was implemented in stages which required comparison of the results from the R code base and the Python code base.

The final product consisted of a ready-to-use Python implementation made for singular machine processing and a detailed proposal of implementation for cluster processing. The Python implementation for singular machine computing utilizes multiple processing libraries and is able to complete some of the necessary calculations on clusters to speed up computational time for larger files. However, single machine processing libraries used in pGFisher implementation are notably slower than their R counterparts due to the approximation algorithms used. Such an issue in extended processing will be resolved by modifying the entire pipeline to use cluster computing. Future works can focus on extending the cluster processing implementation by adding onto the existing algorithms, optimizing the singular machine implementation, and comparing modified GFisher outputs with existing association study outputs.

2. Background

2.1 Genetics Background

Identifying genes contributing to human disease greatly depends on the type of diseases being studied. This project focuses on complex traits and common risk factors which require Genome-Wide Association Studies (GWAS) and thousands of cases and controls.

From the Human Genome Project, we learned that a human genome contains approximately 25,000 genes, with only about 2% of the code being for proteins. The Human Genome is about 3 billion base pairs of DNA and contains instructions to make every protein needed for the body.

2.1.1 DeoxyriboNucleic Acid (DNA)

All genetic information is encoded in our DNA. For any two individuals, the DNA is about 99.9% identical. DNA is composed of four main chemicals (or nucleotides) known as adenosine (A), thymidine (T), guanosine (G), and cytidine (C). These nucleotides form base pairs, A with T and C with G, to form the DNA sequence. The DNA sequence itself is defined by the order of nucleotides. Each cell in the body contains two copies of the human genome (one from the mother and one from the father), which contains around 6 billion base pairs. Human DNA is organized into 23 chromosomes, where each chromosome is millions of base pairs long. DNA uses three-letter codes to describe how a protein should be built, reflecting its function in the body. There are a total of 20 amino acids that can be combined in different ways to make various proteins.

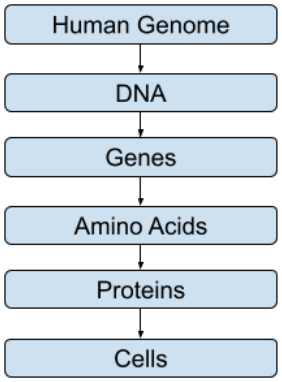


Figure 2.1.1a: The workflow of the genetic process relating the human genome to the cells

2.1.2 DNA variants

The differences in DNA can be described as variants, polymorphisms, or mutations. A variant is an umbrella term to describe any difference in the DNA sequence. A polymorphism occurs when a DNA variant does not cause a disease. A mutation, on the other hand, is a DNA variant that causes a disease. The two main categories that contribute to human disease are alterations to the sequence of the human genome (gene mutation) and alterations to the structure of a chromosome (chromosome alteration). A gene mutation causes a disturbance in the order of base pairs, causing incorrect instructions for protein creation. There are three main types of variants associated with gene mutations: missense variants, nonsense variants, and deletion variants. A missense variant causes a substitution of a single amino acid in a protein. A nonsense variant causes the amino acid sequence to end prematurely. A deletion variant typically deletes a portion of the protein. Chromosome abnormalities are an alteration in the number of chromosomes. The three main types of chromosome abnormalities include part of one chromosome being swapped with another, multiple rearrangements of chromosomes, and duplication or deletion of portions of a chromosome.

2.1.3 Amyotrophic lateral sclerosis (ALS)

ALS, also known as Lou Gehrig's Disease, is a rare and progressive neurological disease that affects the nerve cells responsible for controlling voluntary muscle movement. As the disease advances, it leads to muscle weakness, wasting, and the eventual loss of the ability to control voluntary movements. ALS primarily affects people between the ages of 55 and 75, with a slightly higher prevalence in males and Caucasians. Fifty percent of ALS patients die within 3 years, and 75% die within 5 years. Diagnosis involves a thorough medical evaluation, including electromyography and nerve conduction studies, while other diseases are ruled out.

There are currently two FDA-approved drugs that treat ALS, but the survival benefit is only three months. Unfortunately, there is no cure for ALS, but various treatments and supportive care can help manage its symptoms and improve the quality of life. Current research efforts aim to understand the cellular mechanisms involved in ALS, explore genetics and epigenetics, identify biomarkers for diagnosis and progression monitoring, and develop new treatment options, including drug-like compounds and gene therapy. ALS remains a challenging disease with no definitive cure, but ongoing research offers hope for better understanding and management.

The disease can be sporadic, occurring randomly, or familial, linked to specific genetic mutations. Ninety percent of all ALS cases are sporadic ALS where there is no family history of the disease and it is caused by a combination of genes and the environment. On the other hand, familial ALS is typically caused by a single mutation in a single gene with a family history of ALS present. Two-thirds of the genes for familial ALS have been identified, and this study mainly focuses on identifying a portion of the remaining one-third. Rare variant analysis identifies TBK1 mutations in ALS as the top hit. Two thousand eight hundred forty-three variants within sporadic ALS patients were compared to 4,310 controls. The top genes were tested on 1,318 patients with familial ALS against 2,371 controls to identify. The combination of those two analyses revealed TBK1 mutation mutations as the top hit.

2.2 Association Studies

The goal of an association study is to identify whether particular single nucleotide polymorphisms (SNPs) are more common in the disease population when compared to the control population, known as disease association. More than 90% of SNPs are observed in all populations. To this day, about 113 million SNPs have been identified, describing various mutations causing diseases. SNP stands for "Single Nucleotide Polymorphism" in genetics. It refers to a common type of genetic variation that occurs when a single nucleotide (the building blocks of DNA) at a specific position in the genome differs among individuals.

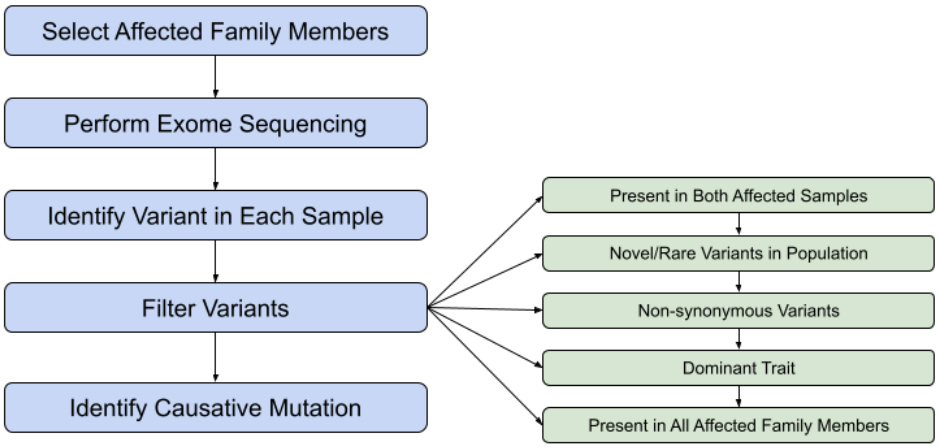


Figure 2.2a: Flow of association wide study to determine a mutation causing a particular disease.

An important note is that the fact that an SNP is associated with a particular disease does not imply causation, but disease association indicates that a variant in some particular region has functional significance. Some of the advantages of gene association studies are that they are relatively cheap, simple, quick, and publishable. However, a major disadvantage is that most of the published candidate gene association studies cannot be replicated, which is essential for validating scientific findings. Some of the

reasons that make association studies unsuccessful include publication bias, inherent statistical bias, matching of case and control samples, and lack of replication.

2.2.1 Genome Wide Association Study (GWAS)

While there are some common flaws in association studies, many can be addressed by performing Genome-Wide Association Studies (GWAS)[1] rather than an association study on a singular gene. GWAS hits can be useful for the following reasons: they may lead to new insights into disease mechanisms and targets, may identify new pathways involved with the disease, and may lead to diagnostic tests. GWAS uses a polygenic risk score to determine how strongly a particular combination of SNPs, weighed by each of their effects, contributes to a particular disease.

2.3 Mathematical Background

A common method to provide association is to use Fisher's combination of independent p-value statistics which is common in bio-medical research. In essence, Fisher's combination allows for combining independent p-value calculations falling under the same hypothesis[2]. One of the main limitations of Fisher's combined statistics is the assumption of independence which is often not the case in genetic applications.

The implementation of the GFisher algorithm, a modification of traditional Fisher's combination which allows to increase statistical power at controlled type I error[16]. Notice that the proposed p_GFisher calculation accounts for the relationship between various presented variables. More specifically, GFisher represents the general family of Fisher type statistics which covers Fisher's combination, Good's statistic, Lancaster's statistic, weighted Z-score combination, etc. The overarching goal is to provide alternative methods for p-value calculations and combinations to showcase a novel approach to detect novel disease genes.

GFisher follows the below general pipeline displayed in Figure 2.3a.

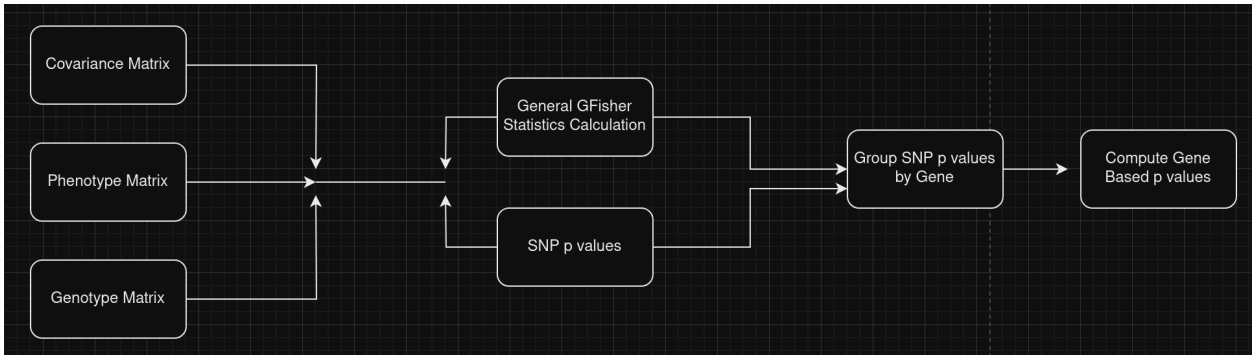


Figure 2.3a: A depiction of the pipeline from the original covariance matrix, phenotype matrix, and genotype matrix to calculate the gene based p values using p_GFisher.

2.4 Tools Used

2.4.1 Spark

Spark[3] is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters. Spark is built on a principle of traditional database management systems required for large scale data processing. Spark leverages distributed computing to process and perform operations on large quantities of data to make computations viable. At its core, Spark relies on primitive operations performed on resilient distributed datasets (RDDs) to provide a high-level interface for more complex calculations.

While Spark offers a multitude of APIs, this project focused on the PySpark library[4] which is an implementation of Spark adapted to Python. The main advantage of using Spark is in the ability to perform computations on large files containing necessary genetic information which may not be viable with the use of only one machine.

2.4.2 Hail

Hail[5] is an open-source Python library built specifically to work with variant call format (VCF)[6] files and perform genetic analysis efficiently. More specifically, Hail is a tool that allows for efficient

processing of large amounts of data by implementing principles of distributed computing through Spark clusters which work on principles similar to Pandas library implemented in Python.

Hail is built to scale and has first-class support for multi-dimensional structured data, like the genomic data in a genome-wide association study (GWAS). Hail is exposed as a Python library, using primitives for distributed queries and linear algebra implemented in Scala, Spark, and increasingly C++.

For the purposes of this project, Hail is used as a uniform processing model of large VCF files to effectively share file processing at various stages of filtering and quality control. All the quality control steps were done outside of the scope of this project using Hail's built in functions. The cleaned data was then further processed to fit the needs of our analysis. Data quality control is outside of the scope of this project and was done using all the standardized procedures.

2.4.3 Terra

Terra[7] is a cloud-native platform for biomedical researchers to access data, run analysis tools, and collaborate. The platform leverages Google Cloud for data processing and storage while providing an easy-to-use interface which allows for full control on the amount of computing engines and methods used. More specifically, Terra works with Jupyter notebooks which rely on a single kernel virtual environment with a wide selection of computational engines and start up settings.

Since this project relies on Hail as the main data processing and analysis library, Terra was used with a Python 3.0 kernel and Spark cluster configuration with various settings depending on the computational power needed to process a particular file.

3. Methodology

The project was done in a step-by-step manner taking into account the necessary set-up, planning, code validation, and analysis. The motivation behind rewriting an existing code base into Python was that Python is more compatible with Hail and allows for scalable algorithms which are necessary for cluster

processing. In order to implement the pipeline, the following high level process was implemented as described in Figure 3a.

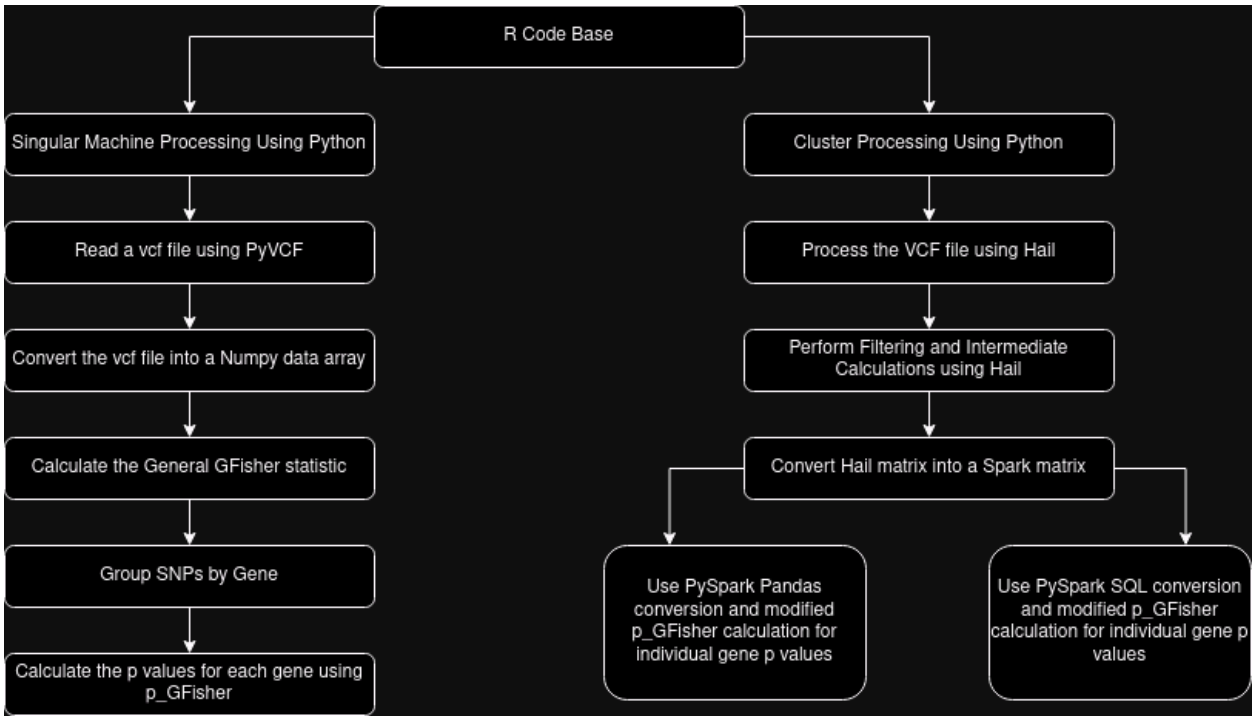


Figure 3a: The high level description of the project which entails looking at the existing R code base with the implementation of p_GFisher calculations followed by Python implementation for single and multi machine computations.

In order to implement the following pipeline to compute the p-values for each individual gene, the project follows the given general process depicted above. In implementing the above pipeline, three main steps were realized: the analysis of an existing R code base, Python implementation for a singular machine processing, and Python implementation for cluster processing.

3. 1 The Analysis of R Code Base

The R code base with the implementation of GFisher algorithm described above is available under the Comprehensive R Archives and was used as the basis for implementation of the Python code. The code base contains functionality for GFisher implementation, comparison with known p-value combination tests, and visualization of the results. For the purposes of this project only the following

functions were implemented and extended using Python: stat_GFisher and p_GFisher along with all the supporting functions descriptions of which are in Figures 7.1a and 7.1b. The implementation model and pipeline proposal used to calculate the necessary p-values for an association study is as follows.

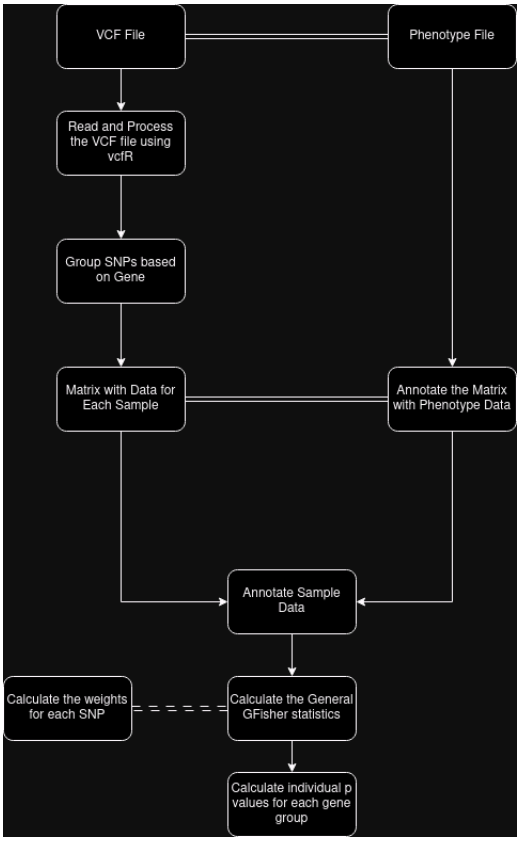


Figure 3.1a: A high level function call to process a VCF file and a phenotype file to obtain final gene based p value calculations.

The above pipeline example was used as a baseline to build single machine processing Python code described above and cross-validate the results.

3. 2 Python Code for Singular Machine Processing Implementation

The motivation in rewriting code from R into Python is to make it compatible with Hail's kernel and requirements to make the code more general purpose. The code implemented for singular machine processing relied on the previously discussed function in R as well as some necessary supporting functions. More specifically, the singular machine processing pipeline relied on numpy which is not

scalable for big data but has been used for cross validation and trial run of the pipeline using the principles similar to Hail's implementation of SKAT[8].

Our model implements logistic skat with the assumption of continuous variables used for comparison. The Python code uses a similar approach to construct the calculation pipeline described in the R code with the tweak of using Hail to process larger VCF files. In this particular implementation Hail is used to build a Matrix table and then performs calculations to group the data by genes. After, the p-value calculations using p-GFisher are called to calculate the p-value for each group similar to SKAT implemented in Hail.

Notice that when the pGFisher function is called it is necessary to convert certain fields of the Matrix Table into numpy arrays which will be passed into the singular machine Python implementation of the pipeline.

3.3 Python Code for Distributed Computation Implementation

In order to extend the singular machine processing, the project requires the use of tools which support distributive computation with existing implementation of the necessary algorithm to extend the reimplementation of the R libraries. More specifically, this project utilizes Spark and Hail data structures to process and analyze the necessary files to find the final p values.

3.3.1 Tools Used

In order to assure that all the computations could be split up into clusters, the project utilizes the following specific data structures and libraries provided by Spark and Hail.

3.3.1.1 Hail

In order to implement the distributed pipeline Hail was used for processing the VCF files, grouping SNPs by gene, computing individual GFisher statistics, and passing the result onto the gene p-value calculation implemented in p_GFisher.

In order to process the VCF files, a built-in Hail function was used to create a matrix table containing all the necessary information extracted from the VCF files. In our calculations, the genotype information and phenotype information was used for further computations needed to run the p_GFisher algorithm. The phenotype information can include one or more specific fields.

For testing purposes, the following code from the Hail Skat example creates an artificial example of a Hail matrix with 2 genes as described in Figure 3.3.1.1a.

```
hl.reset_global_randomness()
mt = hl.balding_nichols_model( n_populations: 1, n_samples=100, n_variants=20)
mt = mt.annotate_rows(gene=mt.locus.position // 12)
mt = mt.annotate_rows(weight=1)
mt = mt.annotate_cols(phenotype=hl.agg.sum(mt.GT.n_alt_alleles()) - 20 + hl.rand_norm( mean: 0,  sd: 1))
```

Figure 3.3.1.1a: A simple example of a Hail matrix table borrowed from Hail docs to create a table with 100 samples and randomly generated values for each row. The annotation steps are required to group the SNPs into genes and add information necessary for further regression calculations. The example code to generate mt is located in exmpleHt.py file.

In order to calculate the general GFisher statistic, the following steps were implemented with the focus on grouping genes using strategies similar to Hail's Skat example. The genes were grouped using an artificial interval small enough to create two genes from a given generated matrix. In real-world applications, gene sorting would need to be done in the quality control stage.

Furthermore, the GFisher statistics were calculated using the traditional GFisher algorithm and output the necessary p-values for each SNP, Z-scores, and the covariance matrix M all of which are used to call p_GFisher computation.

3.3.1.2 Spark

At its core, Hail runs spark clusters to effectively perform necessary calculations on Hail matrix tables. More specifically, the interactions between Hail and Spark allow for our code to be seamlessly integrated using Spark infrastructure extension for Python. The motivation behind using Spark extension instead of Hail directly is that PySpark Pandas library provides more built-in statistical computations needed for the p_GFisher algorithm.

Moreover, Hail matrix tables and Spark Tables are similar in structure. Thus, in order to insert the modified p_GFisher algorithm to perform the calculations our implementation uses PySpark Pandas library and the built-in functions to represent the existing matrix calculations as shown for the singular machine processing implementation. The PySpark Pandas documentation[9] provides detailed examples of analogous functions which can be used for p_GFisher implementation.

One of the challenges of transferring the implementation from single machine processing to cluster processing is assuring that all the necessary supplementary algorithms have been implemented for cluster processing. For this project, cholesky and nearest positive definite (NPD) matrix calculations have not been implemented, see Section 5.2 for more details on addressing the issues above.

3.3.2 Implementation Details and Examples

As described above the following pipeline was implemented to extend the existing R Code base into the Python implementation for single and cluster machine processing. For easier understanding of the code, all of the Python functions are named the same as their counterparts in R.

Moreover, the function dependencies and libraries necessary are described in Section 7.3 and local dependencies are described in the form of comments for each individual function. In order to obtain those dependencies, use the pyenv[10] package which would describe the structure of the entire environment on which the project is run, see Section 7.2 for package dependencies of the project.

Finally, the R code base contains functionality to test the algorithm implementation and provide supplementary functionality which was partially implemented in Python for testing purposes.

3.4 Testing

The testing of Python implementation was done using smaller datasets and matrices which were recorded into files for consistency. The following matrices were obtained for testing purposes using an R based example pipeline containing 50 samples from Chromosome 21. All the data processing was done using R which were then recorded into text files to be passed into the Python pipeline. Notice that both implementations contain a degree of randomness at all the calculation steps.

4. Results

4. 1 Testing and Verification

The testing process included running all the main and supplementary functions implemented in R and Python to compare the outputs and processing time. The two major functions tested are stat_GFisher (), used to calculate the general GFisher statistic, and p_GFisher(), used to calculate the individual gene p values.

4.1.1 stat_GFisher()

In order to test stat_GFisher() implementation the following inputs were used for both of the implementations. Notice that the rest of the parameters were set to default values described in the function's docstring.

```
p_values = [0.1, 0.05, 0.01]
```

Figure 4.1.1a: A simple array of hypothetical p values to evaluate the outputs and performance of R implementation and Python implementation. The code and examples of the function call are located in p_GFisherNumpy.py file.

The docstring for the function is displayed in Figure 7.1a and the results are as shown below.

Implementation	Value	Percent Error	Processing Time (sec)
R	19.80698	0	0.00263
Python	19.80697	2.4713145289761E-5%	0.0000486

4.1.2 p_GFisher()

The p_GFisher() is the main function used to compute gene based p values and it requires the minimum input of a value for the general GFisher statistic and a correlation matrix. The input correlation matrix is displayed in Figure 4.1.2a and the general GFisher statistic value used was 25.5.

```
[[1.  0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3]
 [0.3 1.  0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3]
 [0.3 0.3 1.  0.3 0.3 0.3 0.3 0.3 0.3 0.3]
 [0.3 0.3 0.3 1.  0.3 0.3 0.3 0.3 0.3 0.3]
 [0.3 0.3 0.3 0.3 1.  0.3 0.3 0.3 0.3 0.3]
 [0.3 0.3 0.3 0.3 0.3 1.  0.3 0.3 0.3 0.3]
 [0.3 0.3 0.3 0.3 0.3 0.3 1.  0.3 0.3 0.3]
 [0.3 0.3 0.3 0.3 0.3 0.3 0.3 1.  0.3 0.3]
 [0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 1.  0.3]
 [0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 1. ]]
```

Figure 4.1.2a: The input correlation matrix for the p_GFisher calculation. Note that the rest of the parameters for the function's input remained default, see Section 7.1 for details. The code for p_GFisher implementation and example is located in p_GFisherNumpy.py file.

The results are displayed in Figure 4.1.2b below.

Implementation	Value	Percent Error	Processing Time (sec)
R	0.1873	0	0.03
Python	0.1515	19.1	0.7

4.1.3 Analysis

Based on the error comparisons for the two main functions, the stat_GFisher calculations have very low percent error between the R and Python implementations. p_GFisher calculations, however, have higher percent error. Such a discrepancy could be due to a degree of randomness used in the supporting functions for p_GFisher calculations, see code comments in p_GFisher() file.

Note that all of the testing procedures were performed on small matrices and did not utilize any parallel processing. Moreover, the input data for both of the functions was artificially created, as opposed to getting the data directly from a VCF file, to ensure that the Python implementation runs correctly.

Overall, the outputs for the two main functions to compute p_GFisher statistics implemented in Python could be used alongside the same R functions to compute the statistics in an environment that requires a Python kernel.

4.2 File Processing

The project utilized three main VCF file processing VCFR, an R based library, VCFlib, a Python based library, and Hail, a Python based library utilizing Spark. While the above three examples are evaluated on the basis of time necessary to process the same file, there are other libraries and resources available depending on the format of output needed after the file has been processed. All the file processing speed was measured on a condensed VCF file of 50 samples on chromosome 21. More specifically, the VCF.gz, a zipped version of the VCF file, was 85.5 Mb and 830.2 MB in an unzipped format.

4.2.1 VCFR

VCFR[11] is a R based library for processing VCF files to extract needed genetic data and express it in a matrix format which can further be processed by the necessary algorithms. The following is an example of a call to process a VCF file using VCFR in Figure 4.2.1a.

```
library(vcfR)
vcfFile = "condensed_vcf_files_chr21_filtered_50samples_sn.vcf.gz"
vcf <- read.vcfR(vcfFile)
```

Figure 4.2.1a : A simple use case of VCFR library to process a vcf file. The code is located in getwts_real_example folder.

The given VCF file has the following structure, as interpreted by VCFR and displayed in Figure 4.2.1b.

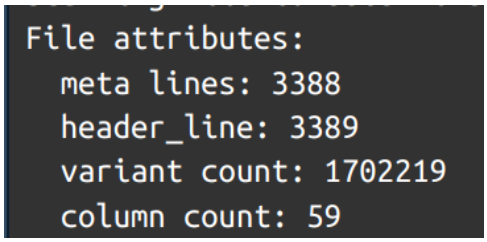


Figure 4.2.1b: The details of VCF file's metadata relevant to the speed of file processing. The code for the given output is located in getwts_real_example folder.

In order to process a condensed file with 50 samples on chromosome 21, [11] VCFR needed 42.29306. seconds.

4.2.3 VCFlib

Similar to VCFR above, VCFlib[12] is a Python based library for processing VCF files and extracting the necessary data for further processing. In order to process the same VCF file with 50 samples on chromosome 21, VCFlib call is displayed in Figure 4.2.3a.

```
startTime = time.time()
vcf_reader = vcf.Reader(filename="condensed_vcf_files_chr21_filtered_50samples_sn.vcf.gz")
endTime = time.time()
```

Figure 4.2.3a: A simple example of processing a VCF file using VCFlib. The code is located in getwts_real_example folder.

The process took 0.0889 seconds.

4.2.4 Hail

Similar to the VCFR and VCFlib, Hail can be used to process VCF files into an appropriate format. The main difference is that Hail's processing utilizes cluster computing. The call to process a VCF file in Hail[13] is displayed in Figure 4.2.4a.

```
ds = hl.import_vcf( path: 'condensed_vcf_files_chr21_filtered_50samples_sn.vcf', reference_genome='GRCh37')
```

Figure 4.2.4a: An example of using Hail to process a VCF file. The code process the vcf file is located in hailVcfProcess.py

Note that .gz files can not be loaded in parallel. In order to load compressed VCF files, the .bgz zip method should be used. Thus, the amount of time it takes to process the same VCF file is 6.72699 seconds with the initialization of Spark clusters..

4.2.5 Analysis

Based on the processing times shown in Figure 4.2.5a , both of the Python implementations - PyVCF and Hail - perform faster than the built in R implementation. While neither VCFR nor PyVCF would scale to large datasets, Hail parallel processing allows to effectively handle larger datasets that would not otherwise be handled by one machine processes.

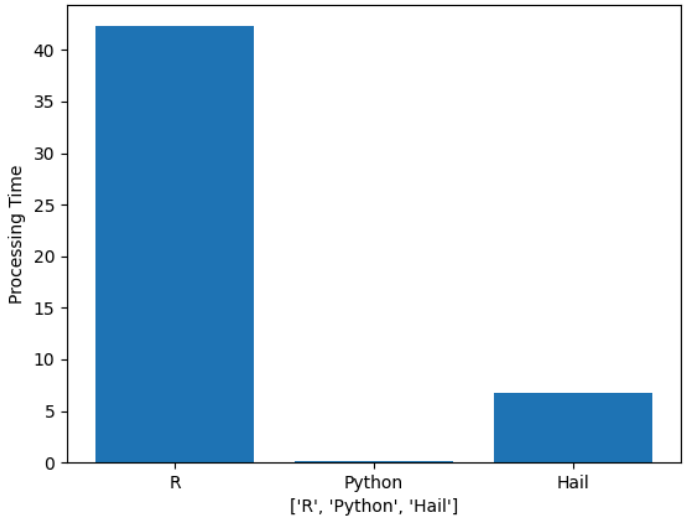


Figure 4.2.5a: A bar chart showing the processing time of the same vcf file using R, Python, and Hail. The code used to generate the graph is located under finalReport/generateCharts.py

4.3 Computational Speed

A covariance matrix calculation was performed on a 10x10, 100x100, and 1,000 and 1,000 matrices in R, Python, and Spark. All the computations were performed 100 times and the average time was taken to account for any error or out of normal usage of the local machine's CPU.

4.3.1 R

A built in R function was used to compute the covariance of a given matrix and the code is displayed in Figure 4.3.1a.

```r
set.seed(123)
# Create a 10 by 10 matrix with random values from 1 to 100
data1 <- matrix(sample(1:100, 100, replace = TRUE), nrow = 1000, ncol = 1000)
num_iterations <- 100
# Loop to run covariance calculation and record execution times
for (i in 1:num_iterations) {
  start_time <- Sys.time()  # Record start time

  # Calculate covariance
  covariance_matrix <- cov(data1)

  end_time <- Sys.time()  # Record end time

  # Calculate execution time in seconds
  execution_time <- end_time - start_time

  # Store execution time in array
  execution_times[i] <- execution_time
}

# Save execution times to a CSV file with a title for the column
output_data <- data.frame(execution_times)
write.csv(output_data, file = "execution_times.csv", row.names = FALSE)
```

Figure 4.3.1a: The code snippet used to find the processing times of the same computation for each of the three example matrix sizes. The array of processing times is recorded into a CSV file for further processing. The code is located in calcRTime.r file under the results folder.

The computations times from each of the runs for each matrix size are displayed in Figure 4.3.1b.

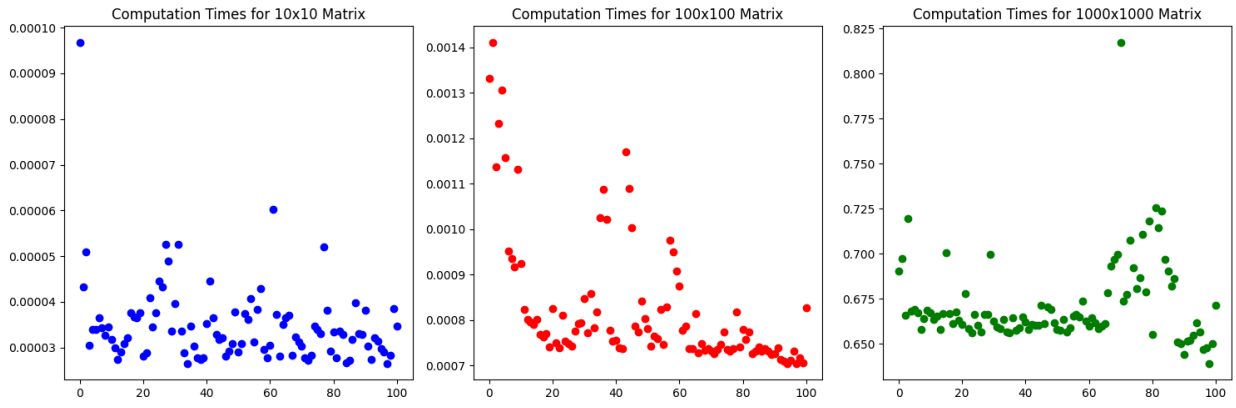


Figure 4.3.1b: Three side-by-side plots of processing times for each of the 100 runs performed for each matrix calculation.

4.3.2 Python Numpy

The code used to calculate covariance using numpy is displayed in Figure 4.3.2a. The matrices were all randomly generated with values between 1 and 100 inclusive using the numpy library.

```
allTimes = np.array([])
for i in range(0,100):
    start = time.time()
    cov = np.cov(arr1)
    end = time.time()
    totalProcessing = end - start
    allTimes = np.append(allTimes, totalProcessing)
return allTimes
```

Figure 4.3.2a: A snippet of code used to calculate the time necessary to compute the covariance matrix given an array of a particular dimension. All the computations are run 100 times and an array of processing time is returned for further computations. The code is located in calcTimes.py file.

The scatter plot of each of the matrix sizes is displayed in Figure 4.3.2b:

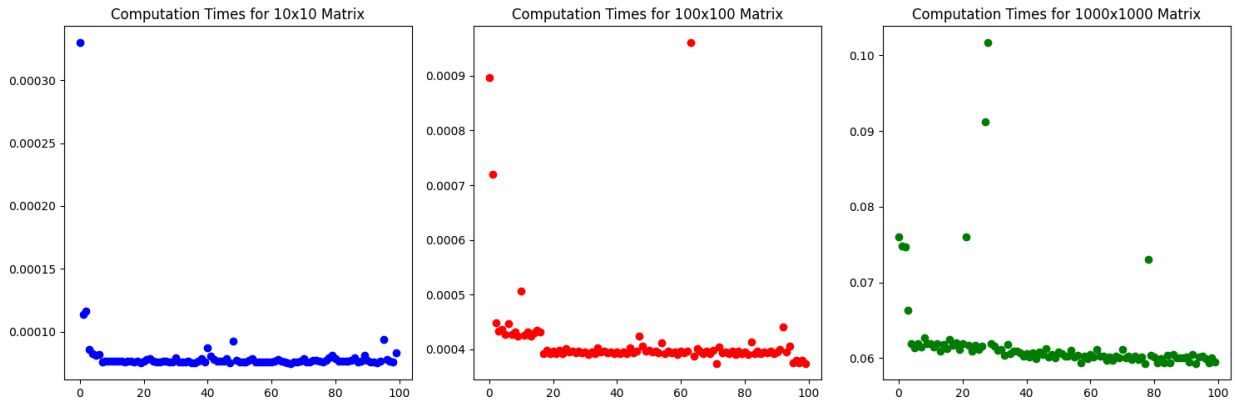


Figure 4.3.2b: Side-by-side scatter plots for processing times of each computation for each of the three evaluated matrix sizes. The code is located in calcTimes.py file to generate the data and code to create the graphs is located in finalReport/generateCharts.py file.

4.3.3 Python Pandas

The code used to calculate the covariance using Pandas is displayed in Figure 4.3.3a.

```
allTimes = np.array([])
for i in range(0,100):
    start = time.time()
    covariance_matrix = df1.cov()
    end = time.time()
    total = end - start
    allTimes = np.append(allTimes, total)
return allTimes
```

Figure 4.3.3a: A snippet of code used to calculate the processing time for 100 runs of covariance matrix calculation.The code is located in calcTimes.py file.

The data frames were created by creating random numpy arrays of the respective sizes and then converted to a Pandas dataframe. The scatter plot of processing times for each of the matrix sizes is displayed in Figure 4.3.3b

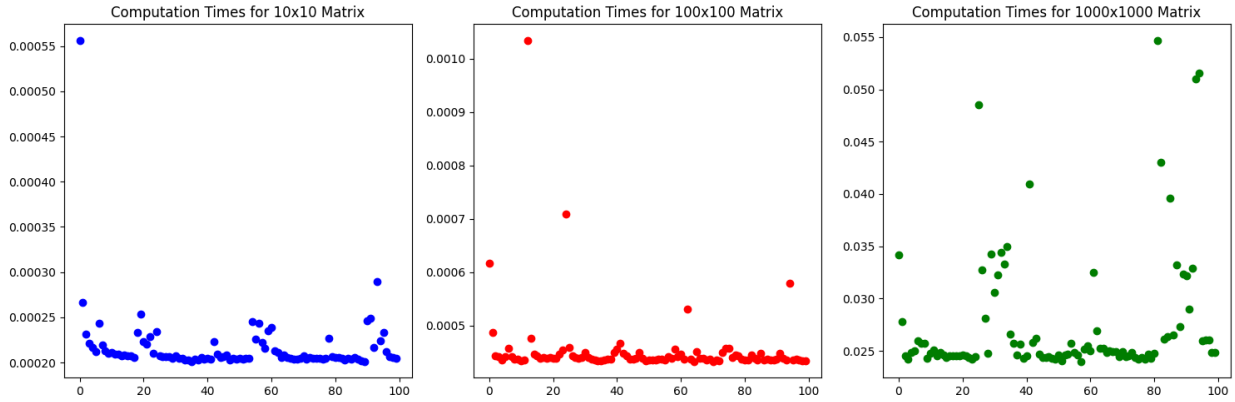


Figure 4.3.3b: Side-by-side scatter plots of each of the 100 processing times for each represented matrix size. The code located to generate the results is located in calcTimes.py and the code to create the graphs is located in finalReport/generateCharts.py

4.3.4 PySpark SQL Using Terra

Terra was used to perform all the covariance calculations on a PySpark SQL matrix. The specific Jupyter Notebook configurations used to launch the Spark cluster are as displayed in Figure 4.3.4a.

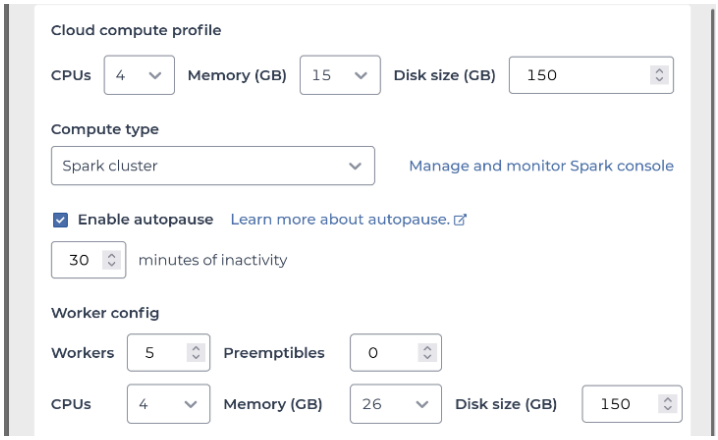


Figure 4.3.4a: Computing environment used to run 5 Spark clusters to compute the covariance.

The scatter plots for processing time for each of the matrices is displayed in Figure 4.3.4b.

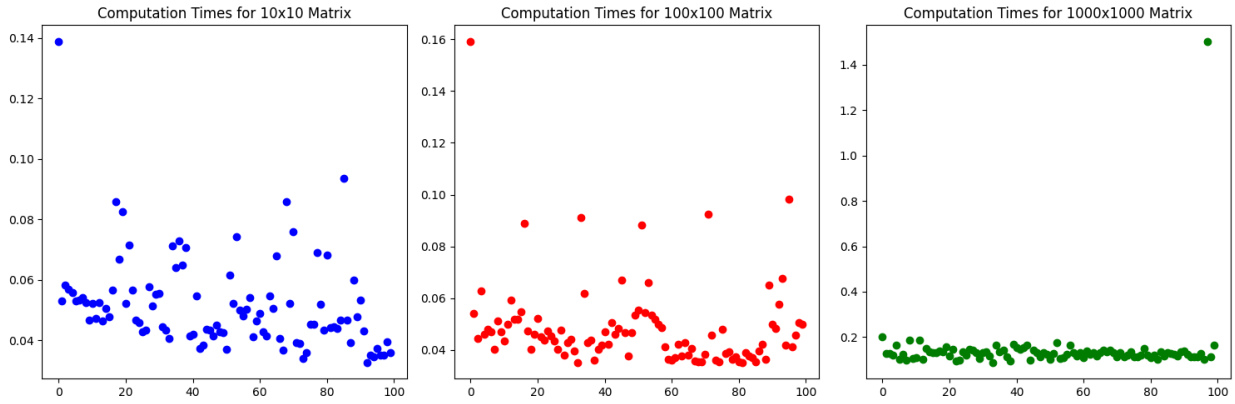


Figure 4.3.4b: Side-by-side scatter plot for processing times for each of the 100 runs of computations. Notice that in the first two plots there is more variation than in the last one which is the result of varying availability of cloud resources. The code to generate the data is located in terraSpark.py and the code to generate the graphs is located in sparkResults.py

4.3.5 Analysis

Notice that for each of the libraries the average computational time was relatively similar for the 10x10 and 100x100 matrix with a steep increase for the 1000x1000 matrix calculation.

The average computation times for each of the libraries used are displayed in the table below. The code to generate the table is located in results/calculations/calcTimes.py file and similar calculations are performed in a Terra notebook, see Figure 7.1f.

Matrix Size	R	Python Numpy	Python Pandas	Spark 5 Workers
10	3.48E-05	0.0000807499885	0.0002171111107	0.04230956554
100	0.0008273530006	0.000415687561	0.0004545044899	0.05197875261
1000	0.6713336945	0.0620933342	0.02765007496	0.1368969536

More specifically, Figure 4.3.5a displays the bar plots of average computation time for each of the libraries and each of the matrix sizes.

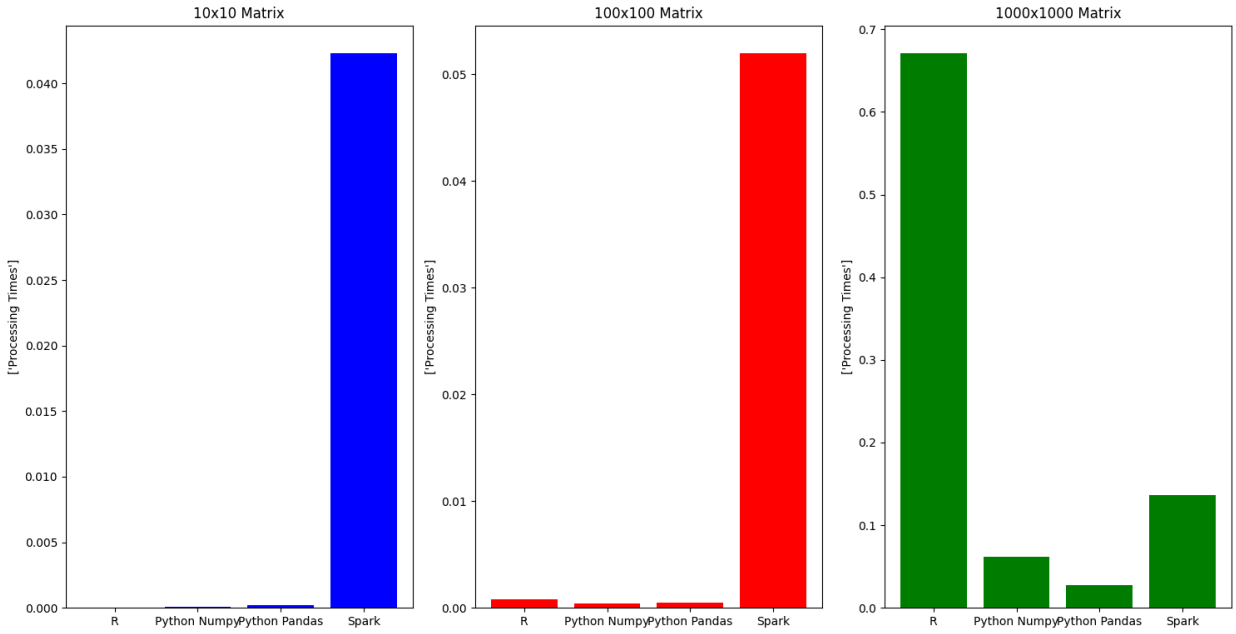


Figure 4.3.5a: Average processing time across all mentioned libraries. Notice that Pandas and PySpark clusters have relatively similar performance which would be explained by PySpark's dependency on Pandas. The code to create the graphs is located in the results folder along with all the necessary csv files.

Notice that for the smallest matrix, 10x10, Pandas was the slowest average computation time while for the largest matrix, 1000x1000, it was the fastest computation library. The R library behaved in the opposite manner. One possible explanation is that Pandas is aimed to work efficiently with larger datasets while the built in R libraries may not be as efficient.

Moreover, using parallelized computations supported by Spark and ran on Terra used 5 worker clusters. Notice that there may be significant variation in processing time necessary for each computation performed using Terra due to availability of cloud resources and remote machines. While the computational times for smaller matrices are generally greater when compared to the other libraries, Spark has a relatively small increase in the processing time needed as the matrix sizes increase exponentially. The processing times for smaller matrices can be accounted for by the fact that Spark parallelizes all computations which accounts for the major portion of processing time of any input.

Overall, single machine processing libraries, Python and R, support all the necessary computations for smaller data inputs and provide more time efficient calculations. On the larger scale, however, Spark will have an advantage of processing the data by using distributed computing.

4.4 Integration of Hail with Spark

In order to address data scalability issues that come up with large genetic data and assure integration with Hail, the following simple pipeline displayed in Figure 4.4a proposes a method to effectively combine the two libraries. See how the calculations compare in Figures 7.1d and 7.1e.

```
hl.init(log="hailSparkCalculation.log")
#create a hail matrix

t = hl.utils.range_table(10)
t = t.annotate(a = t.idx, b = t.idx * t.idx, c = hl.str(t.idx))
t.describe()

#perform the stats calculation
print("Row a values are as follows")
t.a.show()
stats = t.aggregate(hl.agg.stats(t.a))
print("The Hail statistics for row  a are as follows: ")
print(stats)

#convert a Hail matrix into pySpark
print("The PySpark statistics for row a are as follows: ")
psA = ps.DataFrame(t.a.collect())
mean = psA[0].mean()
std = psA[0].std(ddof=0)
min = psA[0].min()
max = psA[0].max()
sum = psA[0].sum()
psStats = { "mean": mean,
            "stdev":std,
            "min":min,
            "max":max,
            "sum": sum
}
print(psStats)
```

Figure 4.4a: A simple example of comparing computations implemented for Hail matrix tables and PySpark matrices. In this particular example, a Hail matrix is created and converted into PySpark Pandas DataFrame to perform the same basic statistical calculations.The detailed code is located in hailSparkCalculations.py file.

Since the pipeline was able to convert the data types, further p_GFisher implementations would rely on Spark data frames which would be compatible with Hail's outputs. See proposed pipeline in Figure 4.4b.

```
hl.init(log="sparkPipeline.log")
ht = hl.read_table("ht.mt")
ht.show()
ht.describe()
sp = ht.to_spark(flatten=True) #returns a spark sql data frame
spPandas = ps.DataFrame(sp)

#get general GFisher statistics
pvals, gf, M = eht.findGFM(ht)

#convert all the necessary information in PySparkPandas dataframe
pvalsPandas = ps.DataFrame(pvals.collect())
gfPandas = ps.DataFrame(gf.collect())
```

Figure 4.4b: An example of integration of Hail, to process files and calculate general statistics, with Spark, to provide another data structure for further processing by p_GFisher. The detailed code is located in sparkPipeline.py

In this manner, all of the file processing is done by Hail's optimized system and more fine grained calculations are done using PySpark Pandas or PySpark SQL. Notice that PySpark Pandas is an API built on top of PySpark SQL which may sometimes cause dependency issues with version compatibility for Python, PyArrow, Pandas, Scala and Java. One possible method to avoid those issues is to use PySpark SQL directly since Hail matrices and PySpark SQL matrices have the same structure and require less conversion than the conversions necessary to work with PySpark Pandas.

In order to fully implement p_GFisher such that it is compatible with distributive computing, a similar pipeline for data conversion and communication would need to be used. For further details on future necessary p_GFisher modifications necessary see Section 5.2.

4.5 R Implementation of p_GFisher on a Smaller VCF file

In order to calculate gene based p-values using the R code base the following steps were implemented: reading the VCF file, processing the data, filtering by SNP, calculating general statistics, calculating the covariance matrix, and calculating the final p values. See getwts_real_example folder which contains the pipeline code as well as all the necessary files to perform the calculations. It took

about 50 seconds to run the entire pipeline including the original processing of the genotype and phenotype files.

Notice that this particular example processes the vcf and then performs all the necessary calculations to compute the p values. More specifically, the majority of computation power is required to do the initial file processing and gene filtering. All the necessary statistical computations take a relatively small amount of time. More specifically, the amount of time to process the individual VCF file and time to run the entire pipeline are almost analogous. Such similar processing times could be explained by the relatively small size of the genetic data input. Given larger files, the time to perform all necessary preparatory steps and the p values calculations will increase along with the file processing time.

4.6 Single Machine p_GFisher Calculation Using Hail

The existing Python implementation for p_GFisher relies on a number of libraries which do not support distributed computing, similar to the existing R implementation. One strategy to optimize computation is to use built-in Hail functions to process the data before it is passed along into the p_GFisher implementation. The code for these results is in hailNumpyPipeline.py.

In order to measure the computation time of singular machine calculation using Python and Hail the following steps were implemented: generate a Hail matrix, calculate the general GFisher statistics, and finally compute the individual gene p values. Note that the matrix table used is of comparable size to the VCF file processed by the R script.

Notice that the pipeline described below uses logic similar to Hail SKAT implementation[8] to process the matrix and calculate the general statistics. The main portion borrowed from Hail's implementation was gene grouping and the idea that Hail matrix table structure allows the application of the same calculation to every single gene. Hail's Skat is structured as described in the function's documentation.

After generating the matrix, further calculations are performed to obtain a Hail table which contains global variables necessary to compute p values for individual genes. The table structure and example content are displayed in Figure 7.1c. Notice that now the table contains all the data that is sorted by the gene represented with the original SNPs. Due to this separation, it would be more effective to perform p_GFisher calculations.

More specifically, p_GFisher uses the values for G, weight, Q, and group. Next, use built in Hail functions to calculate GFisher statistics followed by conversion to numpy and call to p_GFisher. In order to calculate the p values, p_vals, and the covariance matrix, M, a general GFisher statistic calculation is used.

The entire process to output two p values for the two represented genes takes 11.92384934425354 seconds. Notice that due to the details of original p_GFisher implementation, the Python implementation only works with one gene at a time and thus a for loop was necessary to compute the statistics for both of the genes. In this particular instance, a for loop significantly slows down the computation causing extended computation time. In order to improve computational time, see Section 5.2 for notes on optimization.

5. Future Work

Some of the potential outlets for future work include performing optimization and quality control on the code for single machine processing as well as finalizing the algorithms necessary for cluster computing.

5.1 Optimization and Quality Control for Single Machine Processing

To further optimize single machine processing code in Python, a number of alternative algorithms and libraries can be implemented. More specifically, the integrations and nearest positive definite (NPD) matrix calculations can be improved upon. The integration calculations used in getGFisherGM() take up a lot of computational resources and can be replaced by approximation functions. A potential source of those approximations could come from libraries such as scipy, sklearn, or pytorch.

Moreover, the NPD calculation could be improved with the use of another built in library or another approximation approach. The current implementation utilizes an existing NPD class[14], which provides functionality to determine whether or not any computations need to be performed and functions to perform those computations. Another NPD implementation may be more efficient if it uses lower level calculations and relies on more approximations.

5.2 Continuation of Implementation for Cluster Computing

The final results of Python implementation described above utilized distributive computing up until the main p_GFisher function was called. One optimization strategy would be to rewrite the p_GFisher function such that it is compatible with Spark or Hail data structures. The main advantage of extending the pipeline to include distributive computation all across the board would be scalability of calculations meaning that large data structures can be processed and reduced via the p_GFisher approach.

In order to expand the pipeline, one approach would be to follow the implementation of Hail SKAT function. More specifically, focus on pgenchisq function which is used to perform the gene based p value calculation inside logistic skat implementation. Notice that all the computations are passed along to code in Scala which then utilizes existing computations with some low level additions to come up with the Hail's output p value. Such a low level approach would guarantee the compatibility of p_GFisher computations with clusters, however, there is another approach which would utilize PySpark Pandas.

The proposed strategy includes using existing Hail code to compute general GFisher statistics, convert a Hail matrix into a PySpark Pandas Matrix, and modify p_GFisher such that it accepts PySpark Pandas matrix as an input. The main difficulty with extending p_GFisher to work with PySpark Pandas matrix is that all the calculations currently described in Numpy would need to be rewritten such that they are compatible with the new data structure. The implementation would call for transition to PySpark Pandas API since it contains more functionality for statistical calculation than Hail does. Moreover, since Hail works directly with Spark, the transition to PySpark Pandas would be relatively inexpensive and would perform similar functionality.

More specifically, the Cholesky calculations, used in p_GFisher() function, and NPD calculations, used in p_GFisher() function, do not currently have an implementation which supports parallel computing. However, Cholesky calculation is mathematically described and can be used as a stepping stone for implementation. The NPD calculation is not currently described in literature but, as discussed earlier, it may be replaced by an approximation which is compatible with PySpark Pandas.

6. Conclusion

ALS is a rare neurological disorder which is caused by gene mutations and is often diagnosed at the late stages of the illness. In order to improve diagnostic capabilities, this project leverages the idea of gene association to discover novel genes associated with ALS. More specifically, the project utilizes existing genetic data to calculate general GFisher statistic and use p_GFisher modifications to calculate gene based statistics. There is an existing implementation in R and Python which allows efficient processing of smaller datasets. However, genetic datasets are often large and require distributed computation to perform any type of calculations. Thus, the proposed extension of p_GFisher utilizes Hail, Terra, and Spark to assure parallel computations which would be able to handle larger datasets.

7. Code Samples

7.1 Functions

```
Compute the GFisher test statistics.

:param p: A vector of input p-values of the GFisher test.
:param df: Degrees of freedom for inverse chi-square transformation for each p-value.
          It can be a vector of the same length as p, indicating each transformation function
          might have a different df, or a single number, indicating the same degrees of freedom for all.
:param w: Vector of non-negative weights for each p-value. Default is 1 (equal weights).
:return: The GFisher test statistics.
:rtype: float
:details: The statistic is calculated based on normalized weights, i.e., w/mean(w).
:author: Hong Zhang
:examples:
    n = 10
    pval = np.random.rand(n)
    fisher_stat(pval, df=2, w=1)
    fisher_stat(pval, df=[2]*n, w=[1]*n)
    fisher_stat(pval, df=list(range(1, n+1)), w=list(range(1, n+1)))
:export:
     # Example 1:
p_values = [0.1, 0.05, 0.01]
fisher_statistic = stat_GFisher(p_values)
print("Fisher Statistic (Default arguments):", fisher_statistic)
```

Figure 7.1a: Function description for stat_GFisher with examples. The code for the function is located in p_GFisherNumpy.py file.

```
"""
Calculates p-value using Fisher's exact test with various methods.
Depends: ....

Args:
    q: Test statistic (float).
    df: Degrees of freedom (array-like of ints).
    w: Weights for observations (array-like of floats).
    M: correlation matrix of the Zscores from which the input p-values were obtained.
    p_type: Tail of the test ("two" for two-sided, other for one-sided).
    method: Method for calculating p-value ("MR" for Monte Carlo, "HYB" for hybrid, "Brown" for Brown's method).
    nsim: Number of simulations for Monte Carlo method (int, default=None).
    seed: Random seed for Monte Carlo simulation (int, default=None).

Returns:
    The calculated p-value (float).

  Examples:
  np.random.seed(123)
  # Define the dimensions
  n = 10
  # Create the matrix M
  M = 0.3 * np.ones((n, n)) + np.diag(0.7 * np.ones(n))
  # Generate zscore -> zscore; pval, gf1 will not always match the r code due to random generation
  zscore = np.random.normal(size=n) @ np.linalg.cholesky(M)
  # Calculate p-values
  pval = 2 * (1 - norm.cdf(np.abs(zscore)))

  gf1 = stat_GFisher(pval, df=2, w=1)
  gf2 = stat_GFisher(pval, df=np.arange(1, n + 1), w=np.arange(1, n + 1))

  result1 = p_GFisher(gf1, df=2, w=1, M=M, method="HYB")
  result2 = p_GFisher(gf1, df=2, w=1, M=M, method="MR", nsim=5e4)
  result3 = p_GFisher(gf2, df=list(range(1, n + 1)), w=list(range(1, n + 1)), M=M, method="HYB")
  result4 = p_GFisher(gf2, df=11, w=11, M=M, method="MR")
```

Figure 7.1b: Description and example usage of p_GFisher Python implementation. The code for the function is located in p_GFisherNumpy.py file.

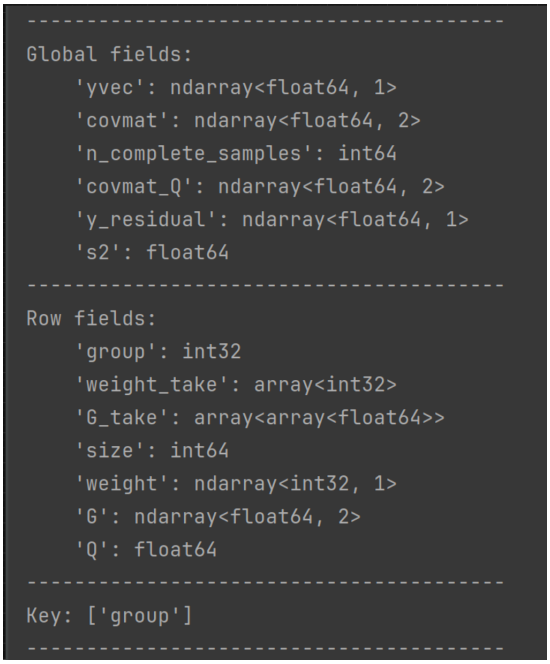


Figure 7.1c: The structure and fields of a Hail table used to calculate p_GFisher statics with numpy and Hail integration. The code to obtain this snippet is in exampleHt.py file

```
The statistics of Row a are as follows:
Struct(mean=4.5, stdev=2.8722813232690143, min=0.0, max=9.0, n=10, sum=45.0)
```

Figure 7.1d: The general statistical information using Hail's built-in aggregation stats. The detailed code for these calculations is in hailSparkCalculations.py file.

```
{'mean': 4.5, 'stdev': 2.8722813232690143, 'min': 0, 'max': 9, 'sum': 45}
```

Figure 7.1e: The general statistics calculation using PySpark Pandas built in functions. Notice that the values are identical to those calculated in Figure 7.1d on the same dataset. The detailed code for these calculations is in hailSparkCalculations.py file.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import rand
import time

# Create a SparkSession
spark = SparkSession.builder \
    .appName("RandomDataFrame") \
    .getOrCreate()

# Define the number of rows and columns
num_rows = 1000
num_cols = num_rows

# Create a list of column names
columns = [f"col{i}" for i in range(1, num_cols+1)]

# Create a DataFrame with random numbers in range 1 to 100
df = spark.range(num_rows).select(*[(rand() * 100).cast("int").alias(col) for col in columns])

# Record processing times
processing_times = []

# Run the process 100 times
for _ in range(100):
    start_time = time.time()

    # Calculate the covariance matrix
    cov = df.cov("col1", "col2")
    end_time = time.time()
    elapsed_time = end_time - start_time
    processing_times.append(elapsed_time)

# Show the processing times
print("Processing times (seconds):", processing_times)
print("The average processing time is", sum(processing_times)/len(processing_times))

# Stop the SparkSession
spark.stop()
```

Figure 7.1f: A code snipper from a Terra notebook used to generate the processing times for covariance calculations using a 5 worker Spark Cluster and PySpark SQL commands. Copy of the code is located in terraSpark.py file.

7.2 Package Dependencies

hail==0.2.130

```
├── aiodns [required: >=2.0.0,<3, installed: 2.0.0]
│   └── pycares [required: >=3.0.0, installed: 4.4.0]
│       └── cffi [required: >=1.5.0, installed: 1.16.0]
│       └── pycparser [required: Any, installed: 2.22]
├── aiohttp [required: >=3.9,<4, installed: 3.9.5]
│   ├── aiosignal [required: >=1.1.2, installed: 1.3.1]
│   │   └── frozenlist [required: >=1.1.0, installed: 1.4.1]
│   ├── async-timeout [required: >=4.0,<5.0, installed: 4.0.3]
│   ├── attrs [required: >=17.3.0, installed: 23.2.0]
│   ├── frozenlist [required: >=1.1.1, installed: 1.4.1]
│   ├── multidict [required: >=4.5,<7.0, installed: 6.0.5]
│   └── yarl [required: >=1.0,<2.0, installed: 1.9.4]
│       ├── idna [required: >=2.0, installed: 3.7]
│       └── multidict [required: >=4.0, installed: 6.0.5]
├── avro [required: >=1.10,<1.12, installed: 1.11.3]
├── azure-identity [required: >=1.6.0,<2, installed: 1.16.0]
│   ├── azure-core [required: >=1.23.0, installed: 1.30.1]
│   │   ├── requests [required: >=2.21.0, installed: 2.31.0]
│   │   │   ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│   │   │   ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│   │   │   ├── idna [required: >=2.5,<4, installed: 3.7]
│   │   │   └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
│   │   ├── six [required: >=1.11.0, installed: 1.16.0]
```

```
│   │   └── typing-extensions [required: >=4.6.0, installed: 4.11.0]
│   ├── cryptography [required: >=2.5, installed: 42.0.5]
│   │   └── cffi [required: >=1.12, installed: 1.16.0]
│   │       └── pycparser [required: Any, installed: 2.22]
│   ├── msal [required: >=1.24.0, installed: 1.28.0]
│   │   ├── cryptography [required: >=0.6,<45, installed: 42.0.5]
│   │   │   └── cffi [required: >=1.12, installed: 1.16.0]
│   │   │       └── pycparser [required: Any, installed: 2.22]
│   │   ├── PyJWT [required: >=1.0.0,<3, installed: 2.8.0]
│   │   └── requests [required: >=2.0.0,<3, installed: 2.31.0]
│   │       ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│   │       ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│   │       ├── idna [required: >=2.5,<4, installed: 3.7]
│   │       └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
│   └── msal-extensions [required: >=0.3.0, installed: 1.1.0]
│       ├── msal [required: >=0.4.1,<2.0.0, installed: 1.28.0]
│       │   ├── cryptography [required: >=0.6,<45, installed: 42.0.5]
│       │   │   └── cffi [required: >=1.12, installed: 1.16.0]
│       │   │       └── pycparser [required: Any, installed: 2.22]
│       │   ├── PyJWT [required: >=1.0.0,<3, installed: 2.8.0]
│       │   └── requests [required: >=2.0.0,<3, installed: 2.31.0]
│       │       ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│       │       ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│       │       ├── idna [required: >=2.5,<4, installed: 3.7]
│       │       └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
│       ├── packaging [required: Any, installed: 24.0]
```

```
│       └── portalocker [required: >=1.0,<3, installed: 2.8.2]
├── azure-mgmt-storage [required: ==20.1.0, installed: 20.1.0]
│   ├── azure-common [required: ~=1.1, installed: 1.1.28]
│   ├── azure-mgmt-core [required: >=1.3.1,<2.0.0, installed: 1.4.0]
│   │   └── azure-core [required: >=1.26.2,<2.0.0, installed: 1.30.1]
│   │       ├── requests [required: >=2.21.0, installed: 2.31.0]
│   │       │   ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│   │       │   ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│   │       │   ├── idna [required: >=2.5,<4, installed: 3.7]
│   │       │   └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
│   │       ├── six [required: >=1.11.0, installed: 1.16.0]
│   │       └── typing-extensions [required: >=4.6.0, installed: 4.11.0]
│   └── msrest [required: >=0.6.21, installed: 0.7.1]
│       ├── azure-core [required: >=1.24.0, installed: 1.30.1]
│       │   ├── requests [required: >=2.21.0, installed: 2.31.0]
│       │   │   ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│       │   │   ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│       │   │   ├── idna [required: >=2.5,<4, installed: 3.7]
│       │   │   └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
│       │   ├── six [required: >=1.11.0, installed: 1.16.0]
│       │   └── typing-extensions [required: >=4.6.0, installed: 4.11.0]
│       ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│       ├── isodate [required: >=0.6.0, installed: 0.6.1]
│       │   └── six [required: Any, installed: 1.16.0]
│       ├── requests [required: ~=2.16, installed: 2.31.0]
│       │   ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
```

```
│       │   ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│       │   ├── idna [required: >=2.5,<4, installed: 3.7]
│       │   └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
│       └── requests-oauthlib [required: >=0.5.0, installed: 2.0.0]
│       ├── oauthlib [required: >=3.0.0, installed: 3.2.2]
│       └── requests [required: >=2.0.0, installed: 2.31.0]
│               ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│               ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│               ├── idna [required: >=2.5,<4, installed: 3.7]
│               └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
├── azure-storage-blob [required: >=12.11.0,<13, installed: 12.19.1]
│   ├── azure-core [required: >=1.28.0,<2.0.0, installed: 1.30.1]
│   │   ├── requests [required: >=2.21.0, installed: 2.31.0]
│   │   │   ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│   │   │   ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│   │   │   ├── idna [required: >=2.5,<4, installed: 3.7]
│   │   │   └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
│   │   ├── six [required: >=1.11.0, installed: 1.16.0]
│   │   └── typing-extensions [required: >=4.6.0, installed: 4.11.0]
│   ├── cryptography [required: >=2.1.4, installed: 42.0.5]
│   │   └── cffi [required: >=1.12, installed: 1.16.0]
│   │       └── pycparser [required: Any, installed: 2.22]
│   ├── isodate [required: >=0.6.1, installed: 0.6.1]
│   │   └── six [required: Any, installed: 1.16.0]
│   └── typing-extensions [required: >=4.3.0, installed: 4.11.0]
├── bokeh [required: >=3,<4, installed: 3.4.1]
```

```
│   ├── contourpy [required: >=1.2, installed: 1.2.1]
│   │   └── numpy [required: >=1.20, installed: 1.26.4]
│   ├── Jinja2 [required: >=2.9, installed: 3.1.3]
│   │   └── MarkupSafe [required: >=2.0, installed: 2.1.5]
│   ├── numpy [required: >=1.16, installed: 1.26.4]
│   ├── packaging [required: >=16.8, installed: 24.0]
│   ├── pandas [required: >=1.2, installed: 1.5.3]
│   │   ├── numpy [required: >=1.21.0, installed: 1.26.4]
│   │   ├── Python-dateutil [required: >=2.8.1, installed: 2.9.0.post0]
│   │   │   └── six [required: >=1.5, installed: 1.16.0]
│   │   └── pytz [required: >=2020.1, installed: 2024.1]
│   ├── pillow [required: >=7.1.0, installed: 10.3.0]
│   ├── PyYAML [required: >=3.10, installed: 6.0.1]
│   ├── tornado [required: >=6.2, installed: 6.4]
│   └── xyzservices [required: >=2021.09.1, installed: 2024.4.0]
├── boto3 [required: >=1.17,<2.0, installed: 1.34.87]
│   ├── botocore [required: >=1.34.87,<1.35.0, installed: 1.34.87]
│   │   ├── jmespath [required: >=0.7.1,<2.0.0, installed: 1.0.1]
│   │   ├── Python-dateutil [required: >=2.1,<3.0.0, installed: 2.9.0.post0]
│   │   │   └── six [required: >=1.5, installed: 1.16.0]
│   │   └── urllib3 [required: >=1.25.4,<3,!=2.2.0, installed: 2.2.1]
│   ├── jmespath [required: >=0.7.1,<2.0.0, installed: 1.0.1]
│   └── s3transfer [required: >=0.10.0,<0.11.0, installed: 0.10.1]
│       └── botocore [required: >=1.33.2,<2.0a.0, installed: 1.34.87]
│           ├── jmespath [required: >=0.7.1,<2.0.0, installed: 1.0.1]
│           ├── Python-dateutil [required: >=2.1,<3.0.0, installed: 2.9.0.post0]
```

```
│       │   └── six [required: >=1.5, installed: 1.16.0]
│       └── urllib3 [required: >=1.25.4,<3,!=2.2.0, installed: 2.2.1]
├── botocore [required: >=1.20,<2.0, installed: 1.34.87]
│   ├── jmespath [required: >=0.7.1,<2.0.0, installed: 1.0.1]
│   ├── Python-dateutil [required: >=2.1,<3.0.0, installed: 2.9.0.post0]
│   │   └── six [required: >=1.5, installed: 1.16.0]
│   └── urllib3 [required: >=1.25.4,<3,!=2.2.0, installed: 2.2.1]
├── decorator [required: <5, installed: 4.4.2]
├── Deprecated [required: >=1.2.10,<1.3, installed: 1.2.14]
│   └── wrapt [required: >=1.10,<2, installed: 1.16.0]
├── dill [required: >=0.3.6,<0.4, installed: 0.3.8]
├── frozenlist [required: >=1.3.1,<2, installed: 1.4.1]
├── google-auth [required: >=2.14.1,<3, installed: 2.29.0]
│   ├── cachetools [required: >=2.0.0,<6.0, installed: 5.3.3]
│   ├── pyasn1-modules [required: >=0.2.1, installed: 0.4.0]
│   │   └── pyasn1 [required: >=0.4.6,<0.7.0, installed: 0.6.0]
│   └── rsa [required: >=3.1.4,<5, installed: 4.9]
│       └── pyasn1 [required: >=0.1.3, installed: 0.6.0]
├── google-auth-oauthlib [required: >=0.5.2,<1, installed: 0.8.0]
│   ├── google-auth [required: >=2.15.0, installed: 2.29.0]
│   │   ├── cachetools [required: >=2.0.0,<6.0, installed: 5.3.3]
│   │   ├── pyasn1-modules [required: >=0.2.1, installed: 0.4.0]
│   │   │   └── pyasn1 [required: >=0.4.6,<0.7.0, installed: 0.6.0]
│   │   └── rsa [required: >=3.1.4,<5, installed: 4.9]
│   │       └── pyasn1 [required: >=0.1.3, installed: 0.6.0]
│   └── requests-oauthlib [required: >=0.7.0, installed: 2.0.0]
```

```
│         ├── oauthlib [required: >=3.0.0, installed: 3.2.2]
│         └── requests [required: >=2.0.0, installed: 2.31.0]
│         ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│         ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│         ├── idna [required: >=2.5,<4, installed: 3.7]
│         └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
├── humanize [required: >=1.0.0,<2, installed: 1.1.0]
├── janus [required: >=0.6,<1.1, installed: 1.0.0]
│   └── typing-extensions [required: >=3.7.4.3, installed: 4.11.0]
├── jproperties [required: >=2.1.1,<3, installed: 2.1.1]
│   └── six [required: ~=1.13, installed: 1.16.0]
├── nest-asyncio [required: >=1.5.8,<2, installed: 1.6.0]
├── numpy [required: <2, installed: 1.26.4]
├── orjson [required: >=3.6.4,<3.9.11, installed: 3.9.10]
├── pandas [required: >=2,<3, installed: 1.5.3]
│   ├── numpy [required: >=1.21.0, installed: 1.26.4]
│   ├── Python-dateutil [required: >=2.8.1, installed: 2.9.0.post0]
│   │   └── six [required: >=1.5, installed: 1.16.0]
│   └── pytz [required: >=2020.1, installed: 2024.1]
├── parsimonious [required: <1, installed: 0.10.0]
│   └── regex [required: >=2022.3.15, installed: 2024.4.16]
├── plotly [required: >=5.18.0,<6, installed: 5.21.0]
│   ├── packaging [required: Any, installed: 24.0]
│   └── tenacity [required: >=6.2.0, installed: 8.2.3]
├── pyspark [required: >=3.3.2,<3.4, installed: 3.3.3]
│   └── py4j [required: ==0.10.9.5, installed: 0.10.9.5]
```

```
├── Python-json-logger [required: >=2.0.2,<3, installed: 2.0.7]
├── PyYAML [required: >=6.0,<7.0, installed: 6.0.1]
├── requests [required: >=2.31.0,<3, installed: 2.31.0]
│   ├── certifi [required: >=2017.4.17, installed: 2024.2.2]
│   ├── charset-normalizer [required: >=2,<4, installed: 3.3.2]
│   ├── idna [required: >=2.5,<4, installed: 3.7]
│   └── urllib3 [required: >=1.21.1,<3, installed: 2.2.1]
├── rich [required: >=12.6.0,<13, installed: 12.6.0]
│   ├── commonmark [required: >=0.9.0,<0.10.0, installed: 0.9.1]
│   └── pygments [required: >=2.6.0,<3.0.0, installed: 2.17.2]
├── scipy [required: >1.2,<1.12, installed: 1.11.4]
│   └── numpy [required: >=1.21.6,<1.28.0, installed: 1.26.4]
├── sortedcontainers [required: >=2.4.0,<3, installed: 2.4.0]
├── tabulate [required: >=0.8.9,<1, installed: 0.9.0]
├── typer [required: >=0.9.0,<1, installed: 0.12.3]
│   ├── click [required: >=8.0.0, installed: 8.1.7]
│   ├── rich [required: >=10.11.0, installed: 12.6.0]
│   │   ├── commonmark [required: >=0.9.0,<0.10.0, installed: 0.9.1]
│   │   └── pygments [required: >=2.6.0,<3.0.0, installed: 2.17.2]
│   ├── shellingham [required: >=1.3.0, installed: 1.5.4]
│   └── typing-extensions [required: >=3.7.4.3, installed: 4.11.0]
└── uvloop [required: >=0.19.0,<1, installed: 0.19.0]
pipenv==2023.12.1
├── certifi [required: Any, installed: 2024.2.2]
├── setuptools [required: >=67, installed: 68.2.0]
└── virtualenv [required: >=20.24.2, installed: 20.25.3]
```

```
        ├── distlib [required: >=0.3.7,<1, installed: 0.3.8]
        ├── filelock [required: >=3.12.2,<4, installed: 3.13.4]
        └── platformdirs [required: >=3.9.1,<5, installed: 4.2.0]
pyarrow==16.0.0
└── numpy [required: >=1.16.6, installed: 1.26.4]
scikit-learn==1.4.2
├── joblib [required: >=1.2.0, installed: 1.4.0]
├── numpy [required: >=1.19.5, installed: 1.26.4]
├── scipy [required: >=1.6.0, installed: 1.11.4]
│   └── numpy [required: >=1.21.6,<1.28.0, installed: 1.26.4]
└── threadpoolctl [required: >=2.0.0, installed: 3.4.0]
statsmodels==0.14.2
├── numpy [required: >=1.22.3, installed: 1.26.4]
├── packaging [required: >=21.3, installed: 24.0]
├── pandas [required: >=1.4,!=2.1.0, installed: 1.5.3]
│   ├── numpy [required: >=1.21.0, installed: 1.26.4]
│   ├── Python-dateutil [required: >=2.8.1, installed: 2.9.0.post0]
│   │   └── six [required: >=1.5, installed: 1.16.0]
│   └── pytz [required: >=2020.1, installed: 2024.1]
├── patsy [required: >=0.5.6, installed: 0.5.6]
│   ├── numpy [required: >=1.4, installed: 1.26.4]
│   └── six [required: Any, installed: 1.16.0]
└── scipy [required: >=1.8,!=1.9.2, installed: 1.11.4]
        └── numpy [required: >=1.21.6,<1.28.0, installed: 1.26.4]
```

8. References

1. U.S. National Library of Medicine. (n.d.). *What are genome-wide association studies?: Medlineplus Genetics*. MedlinePlus. https://medlineplus.gov/genetics/understanding/genomicresearch/gwastudies/#:~:text=Genome%2Dwide%20association%20studies%20(GWAS,(pronounced%20%E2%80%9Csnips%E2%80%9D).
2. Yoon, S., Baik, B., Park, T., & Nam, D. (2021). Powerful p-value combination methods to detect incomplete association. Scientific Reports, 11(1). https://doi.org/10.1038/s41598-021-86465-y
3. Overview - Spark 3.0.0 Documentation. (n.d.). Spark.apache.org. https://spark.apache.org/docs/latest/index.html
4. PySpark Documentation — PySpark 3.3.1 documentation. (n.d.). Spark.apache.org. https://spark.apache.org/docs/3.3.1/api/Python/index.html#:~:text=PySpark%20is%20an%20interface%20for
5. Hail. (n.d.). GitHub. Retrieved May 1, 2024, from https://github.com/hail-is
6. What is a Variant Call Format (VCF) file? (n.d.). Precision Oncology Solutions | GenomOncology. https://www.genomoncology.com/blog/what-is-a-variant-call-format-vcf-file
7. Terra. (n.d.). App.terra.bio. Retrieved May 1, 2024, from https://app.terra.bio/
8. Hail | hail.methods.statgen. (n.d.). Hail.is. Retrieved May 1, 2024, from https://hail.is/docs/0.2/_modules/hail/methods/statgen.html#_logistic_skat

9. pyspark.pandas.DataFrame — PySpark 3.5.0 documentation. (n.d.). Spark.apache.org. https://spark.apache.org/docs/latest/api/Python/reference/pyspark.pandas/api/pyspark.pandas.DataFrame.html
10. Python, R. (n.d.). Managing Multiple Python Versions With pyenv – Real Python. RealPython.com. Retrieved May 1, 2024, from https://realPython.com/intro-to-pyenv/
11. Preliminaries. (n.d.). R-Packages. Retrieved May 1, 2024, from https://cran.r-project.org/web/packages/vcfR/vignettes/intro_to_vcfR.html
12. PyVCF - A Variant Call Format Parser for Python — PyVCF 0.6.8 documentation. (n.d.). Pyvcf.readthedocs.io. https://pyvcf.readthedocs.io/en/latest/
13. Hail | Import / Export. (n.d.). Hail.is. Retrieved May 1, 2024, from https://hail.is/docs/0.2/methods/impex.html
14. bocpdms/nearestPD.py at master · alan-turing-institute/bocpdms. (n.d.). GitHub. Retrieved May 1, 2024, from https://github.com/alan-turing-institute/bocpdms/blob/master/nearestPD.py
15. National Institute of Neurological Disorders and Stroke. (2023, March 8). Amyotrophic lateral sclerosis (ALS). Www.ninds.nih.gov. https://www.ninds.nih.gov/health-information/disorders/amyotrophic-lateral-sclerosis-als
16. Wu, Michael C., Lee, S., Cai, T., Li, Y., Boehnke, M., & Lin, X. (2011). Rare-Variant Association Testing for Sequencing Data with the Sequence Kernel Association Test. The American Journal of Human Genetics, 89(1), 82–93. https://doi.org/10.1016/j.ajhg.2011.05.029
17. Querying gnomad using hail table by gene symbol. (2020, June 10). Hail Discussion. https://discuss.hail.is/t/querying-gnomad-using-hail-table-by-gene-symbol/1452

18. Wu, Michael C., Lee, S., Cai, T., Li, Y., Boehnke, M., & Lin, X. (2011). Rare-Variant Association Testing for Sequencing Data with the Sequence Kernel Association Test. The American Journal of Human Genetics, 89(1), 82–93. https://doi.org/10.1016/j.ajhg.2011.05.029
19. University of California Santa Cruz. (2019). UCSC Genome Browser Home. Ucsc.edu. https://genome.ucsc.edu/
20. Hail | MatrixTable Tutorial. (n.d.). Hail.is. Retrieved May 1, 2024, from https://hail.is/docs/0.2/tutorials/07-matrixtable.html#MatrixTable-Anatomy
21. pyspark.pandas.DataFrame — PySpark master documentation. (n.d.). Spark.apache.org. Retrieved May 1, 2024, from https://spark.apache.org/docs/latest/api/Python/reference/pyspark.pandas/api/pyspark.pandas.DataFrame.html#pyspark.pandas.DataFrame