1kGP SNP Browser – Documentation

Contents

1 Software structure and functionality

The 1kGP SNP browser provides a simple web-based interface to explore and analyze bi-allelic single nucleotide polymorphism (SNP) data, sourced from the IGSR.

It was constructed using:

- Flask for web deployment
- zarr for data processing and storage
- sci-kit allel for processing and calculations
- Dash for visualisations.

The following documentation will focus on each file and folder in the project to explain key functionality and justify the design choices made.

1.1 Data

The "1000 Genomes 30x on GRCh38" dataset is the most up-to-date and most representative publicly available genomic data resource and is our dataset chosen for for 1kGP SNP Browser. The data, along with more information about its collection, can be found at: https://www.internationalgenome.org/data-portal/data-collection/30x-grch38

When compared to the 2015 Phase 3 release, the 30x dataset benefits from higher mean coverage (34x vs 7.4x), large improvements in variant discovery (94.8m SNVs vs 78.2m SNVs and ~4m more rare SNVs) and lower estimated FDR (Byrska-Bishop, et al., 2021)

Included within the dataset are 3,202 individual samples, including 698 who are related, sourced from 26 populations. For the purposes of 1kGP, the related samples have not been included when calculating population statistics, as this would not give a true reflection of the diversity within a given cohort.

Chromosome 22 has been chosen as an initial test case to use for this project because it requires the lowest amount of data storage due to it being the smallest autosome. The software has been designed with scaling in mind where possible, so that an extension of its capabilities to all chromosomes would not require major changes to the codebase.

The data has been filtered to contain only bi-allelic SNPs and non-indels - meaning all indels and multi-allelic SNPs were excluded. Indels were removed as the web application is designed to focus on SNPs alone.

Biallelic SNPs were used because, in the human population, the percentage of triallelic SNPs is ~0.2% (Hodgkinson and Eyre-Walker, 2010). This biallelic assumption will therefore likely not have a substantial impact on the overall statistics or charts produced.

Focusing on biallelic SNPs also reduces the dimensionality of the data and the demands on the software of processing and displaying multiple alternate alleles – it was therefore considered a worthwhile tradeoff. Furthermore, the Hardy-Weinberg equilibrium used for the genotype frequency calculations hinges upon the assumption that the SNP is bi-allelic. (The Hardy-Weinberg Principle | Learn Science at Scitable, 2010)

1.1.1 Data pipeline

The dataset was downloaded in .vcf format from the IGSR website. For this initial development phase just the Chromosome 22 genotypes data was included, the exact file name being:

http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data_collections/1000G_2504_high_coverage/working/20201028_3202_raw_GT_with_annot/20201028_CCDG_14151_B01_GRM_WGS_2020-08-05_chr22.recalibrated_variants.vcf.gz

The original compressed VCF file is 27.9Gb in size and was kept in a compressed format as it was processed from VCF into Zarr format, using the scikit allel *vcf_to_zarr* function. The following arguments were selected during this process:

- A *group* of '22' was assigned to save this Zarr file into a separate folder for chromosome 22. Future chromosomes would be also assigned to their own folders.
- *Fields* was set to '*', I.e., all, to retain all the original data from the VCF file.
- The *alt_number* was set to '1' to keep only the first alternate allele, since we will only use biallelic SNPs.
- Finally, Blosc was selected as the *compressor*. Blosc is optimized for binary data, and therefore functions extremely well with the mix of floating point and integer data included within the genotypes, phased genotypes, and variants data. It is designed to both "reduce the size of size of large datasets on-disk or in-memory" as well as accelerating "memory-bound computations" - essential considerations when working with the large data in this project (Blosc – Read the docs).

The conversion process took approximately 150 minutes, working within a Jupyter Notebook on a personal computer with 16Gb of RAM. The output format is a zarr file, which when used in combination with scikit allel, provides functionality that allows us to work with N-dimensional arrays that function similarly to NumPy arrays, but they stay chunked and compressed throughout. Using the following command ensures that whenever data is loaded or sliced, it remains on disk, rather than be stored in local memory.

```
allel.chunked.storage_registry["default"] =
allel.chunked.storage_zarr.ZarrTmpStorage()
```

The Zarr arrays also benefit from lazy-loading – meaning data is only loaded when called, further reducing memory constraints when dealing with large datasets.

Overall, this approach enabled efficient storage of the Chromosome 22 data, with a total file size of 932MB. A typical query (e.g., searching for a single Gene) uses around 100-200Mb of RAM, and loads query information in under 2 seconds, showing that this method could be economical to externally host the webserver even if all chromosomes were included.

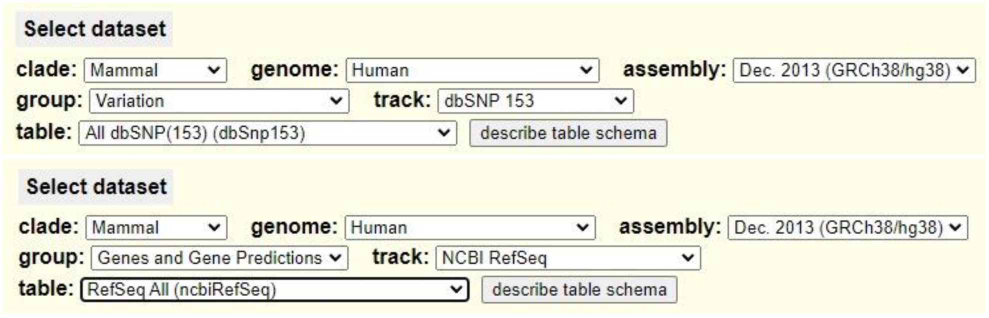
Auxiliary datasets were merged into the main variants table to include population information, dbSNP Refence SNP (RS), as well as gene names and aliases, explained in more detail below.

Given that the 698 related samples are excluded from the data, population data was sourced from samples in the initial phase 3 release and merged on sample ID. Aggregate 'superpopulations' representing 5 continents were chosen as the level of population granularity. Those continents are Africa, America (North and South), East Asia, Europe, and South Asia.

Superpopulations were chosen as the test case to use for this development server as there is a multitude of prior research performed at this level that we could benchmark our results against to ensure that the statistics being displayed were credible. If this server were to be fully rolled out, then granularity at a country-level would also benefit the user – this is discussed in more detail in Section 2.2.3.

dbSNP build 153 RS numbers and NCBI RefSeq gene names with corresponding positions were obtained from the UCSC table browser. The UCSC search configurations for obtaining these files are shown in Figure 1. RS numbers were simply matched by chromosome position to the SNPs extracted from the 1000 Genomes VCF file.

Gene names were mapped to chromosome position accounting for double overlapped genes (see Section 1.1.2.7 for more detail). Both gene names and RS numbers were stored as chunked, compressed Zarr folders under 'GENE1', 'GENE2' and 'RS_VAL'.


Select dataset

clade: Mammal genome: Human assembly: Dec. 2013 (GRCh38/hg38)
group: Variation track: dbSNP 153
table: All dbSNP(153) (dbSnp153)

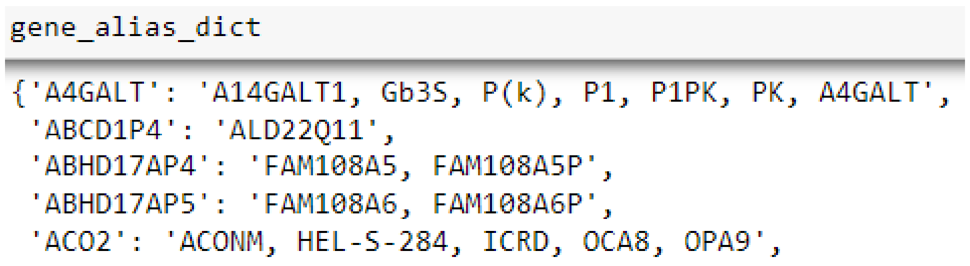
Select dataset

clade: Mammal genome: Human assembly: Dec. 2013 (GRCh38/hg38)
group: Genes and Gene Predictions track: NCBI RefSeq
table: RefSeq All (ncbiRefSeq)

Figure 1: Search configuration in UCSC Table browser for dbSNP 153 and NCBI RefSeq genes.

Gene aliases for chromosome 22 were downloaded from the NCBI Gene data portal. Obtaining a second source of genes allowed us to sense check the data that had been downloaded from UCSC to check the completeness of the list. A list of unique genes in our database was obtained, showing that the method described above assigns 816 unique genes across the SNP positions. Of those, 472 (58%) were found to have at least one alias.

A dictionary was then created using the *json* module – assigning the gene names in the database (stored as 'GENE1' and 'GENE2') as keys, and their aliases as values. An extract of the first five rows of this dictionary can be seen in Figure 2. In the case where a gene name entered by the user does not match the Zarr database, the *filter_data* function within data_proc.py (1.5.5) will search for that name amongst values within the dictionary and return a key in case of a match. The creation of this dictionary can be found on the Github repository - [Generating Gene Aliases.]()



```
gene_alias_dict
```

```
{'A4GALT': 'A14GALT1, Gb3S, P(k), P1, P1PK, PK, A4GALT',
 'ABCD1P4': 'ALD22Q11',
 'ABHD17AP4': 'FAM108A5, FAM108A5P',
 'ABHD17AP5': 'FAM108A6, FAM108A6P',
 'ACO2': 'ACONM, HEL-S-284, ICRD, OCA8, OPA9',
```

Figure 2: First 5 keys and associated values stored within the gene alias dictionary stored as a json file within website/data.

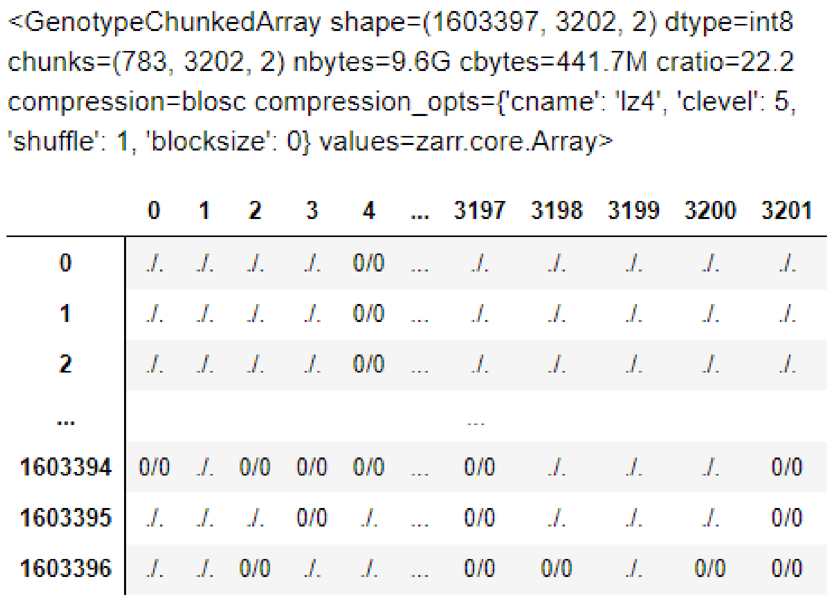
1.1.2 Key Data Fields

A brief description of what each field is and how it will be used by our software/calculations. All the following data fields were either imported from external data or precalculated, then stored in folders in a compressed, chunked Zarr format.

1.1.2.1 GT_BI

This includes the genotypes taken from the 1000 Genomes 30x on GRCh38 Genotypes VCF files. When processing the initial VCF file into Zarr, the Genotypes data is stored in the GT folder. When the main variants data is filtered to remove multiallelic sites and indels the indices of the positions removed were stored as a mask and this was applied to similarly reduce the genotypes data. A guide through this process is included in the Github Repository Notebooks Folder – Zarr Data Storage - Biallelic and Non-Indel.

Genotypes are stored in a GenotypeChunkedArray, a format included within the scikit allel package designed specifically to store chunked, compressed Genotype arrays to minimize storage and RAM requirements when performing queries on the data. An image of the array used can be seen in Figure 3 – The original array has been split into chunks containing 783 rows of data each, reducing the storage requirements from 9.6Gb as an uncompressed NumPy array to be 442Mb as a compressed Zarr folder named 'GT_BI'.



```
<GenotypeChunkedArray shape=(1603397, 3202, 2) dtype=int8
chunks=(783, 3202, 2) nbytes=9.6G cbytes=441.7M cratio=22.2
compression=blosc compression_opts={'cname': 'lz4', 'clevel': 5,
'shuffle': 1, 'blocksize': 0} values=zarr.core.Array>
```

	0	1	2	3	4	...	3197	3198	3199	3200	3201
0	./.	./.	./.	./.	0/0	...	./.	./.	./.	./.	./.
1	./.	./.	./.	./.	0/0	...	./.	./.	./.	./.	./.
2	./.	./.	./.	./.	0/0	...	./.	./.	./.	./.	./.
...							...				
1603394	0/0	./.	0/0	0/0	0/0	...	0/0	./.	./.	./.	0/0
1603395	./.	./.	./.	0/0	./.	...	0/0	./.	./.	./.	0/0
1603396	./.	./.	0/0	./.	./.	...	0/0	0/0	./.	0/0	0/0

Figure 3: Genotypes data stored in GenotypeChunkedArray format.

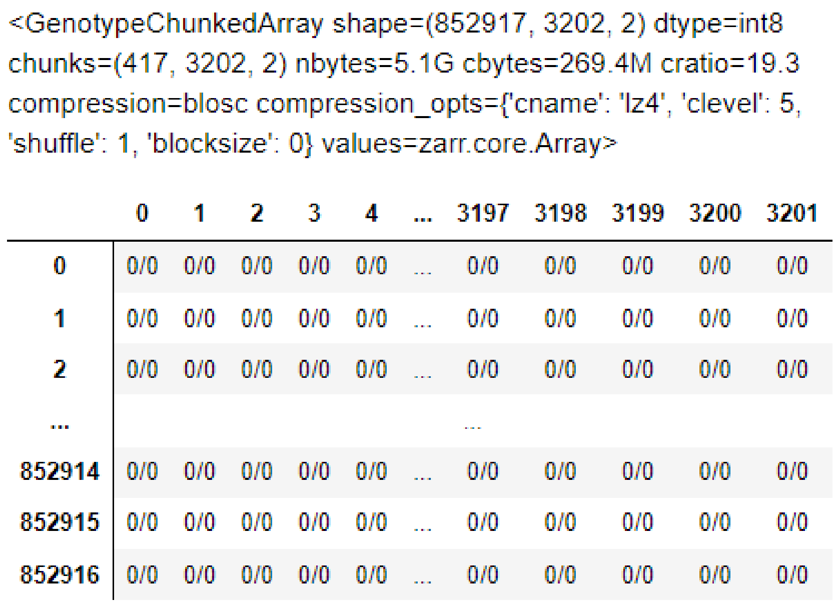
1.1.2.2 PH_GT_BI

This folder includes the phased genotypes taken from the 1000 Genomes 30x on GRCh38 Phased Genotypes VCF files. These can be accessed through the following link:

[1000 Genomes Phased Genotypes VCF]

The 1000 Genomes project included phased genotyping in the 30x dataset, meaning that alleles were identified on the maternal and paternal chromosomes separately. This information is crucial for calculating Haplotype Diversity (see 1.5.8 stats_funcs.py). The phased genotypes were converted into a GenotypeChunkedArray similarly to the genotypes data (see Figure 4), thus reducing the data storage required for this data from 5.1Gb to 269Mb.

To ensure the corresponding values were omitted from the phased genotypes data when removing indels and multi-allelic sites from the main data, the 'PH_POS' folder (1.1.2.9) was used. This allowed us to match up the phased genotypes according to genomic position, thereby allowing us to remove only those which did not meet our criteria. Please see Zarr Data Storage - Biallelic and Non-Indel, for a walkthrough of this process.



```
<GenotypeChunkedArray shape=(852917, 3202, 2) dtype=int8
chunks=(417, 3202, 2) nbytes=5.1G cbytes=269.4M cratio=19.3
compression=blosc compression_opts={'cname': 'lz4', 'clevel': 5,
'shuffle': 1, 'blocksize': 0} values=zarr.core.Array>
```

	0	1	2	3	4	...	3197	3198	3199	3200	3201
0	0/0	0/0	0/0	0/0	0/0	...	0/0	0/0	0/0	0/0	0/0
1	0/0	0/0	0/0	0/0	0/0	...	0/0	0/0	0/0	0/0	0/0
2	0/0	0/0	0/0	0/0	0/0	...	0/0	0/0	0/0	0/0	0/0
...						...					
852914	0/0	0/0	0/0	0/0	0/0	...	0/0	0/0	0/0	0/0	0/0
852915	0/0	0/0	0/0	0/0	0/0	...	0/0	0/0	0/0	0/0	0/0
852916	0/0	0/0	0/0	0/0	0/0	...	0/0	0/0	0/0	0/0	0/0

Figure 4: Phased genotypes data stored in Zarr chunked array format.

1.1.2.3 AA

Ancestral allele sourced from ensembl v105, derived using a 9 primate EPO pipeline. Aligned by genomic positions to the SNV positions of chromosome 22.

1.1.2.4 AF

Allele frequencies by superpopulation; is a field in the INFO column of the vcf. Allele frequency represents the prevalence of an alternate allele within a given population. Derived allele frequency (DAF) and genotype frequency (GF) was calculated using AF values.

1.1.2.5 ALT

Alternate allele for the given SNP, responsible for identification of the alternate allele in DAF calculation (1.1.2.6).

1.1.2.6 DAF

A derived allele refers to a mutated allele, that is a variant that has arisen since the last common ancestor. i.e. an allele that is not identical to a given ancestral allele at a specific genomic position. It follows that Derived Allele Frequency (DAF) is given as the proportion of alleles within a given population that are not the ancestral allele. The ancestral sequence for chromosome 22 was found as a fasta file available from the ftp downloads section of ensembl v105. Given the sum of allele frequencies is 1 and only bi-allelic sites are considered, derived allele frequency was calculated using 1-AF_pop logic dependent on a match between the ancestral allele, the reference allele or the alternate allele or neither for each SNP.

The complete code for DAF calculations is available in the Derived Allele Frequency Calculations notebook on the GitHub repository.

1.1.2.7 GENE1 & GENE 2

The GENE columns help to identify the given name of the gene coding region where a particular SNP resides, if it does happen to be in a protein coding region. SNPs were matched by genomic position given it falls between transcription start and end positions NCBI RefSeq genes.

Within overlapping genomic regions where there are more than one gene, the second gene is represented by GENE2. The major gene which occupies the forward gene's genomic region and all other genes with no overlap are in GENE1.

In the initial stages of development, SNPs were simply matched to gene loci by iterating through a sorted list of transcription start and end points linked to gene names. Approximately one-quarter of all human protein-coding genes are known to overlap. Of these, 71.9% were paired or double overlapping genes, 20.5% triple overlapping and <8% with quadruple or more overlapping (Chen, Pan and Lin, 2019). Overlapping protein-coding genes mapping to the same gene loci are intuitively expected to show co-expression patterns. The code was amended to take overlapping genes into account. This resulted in an increase of ~250 genes detected resulting from successful mapping of small ORFs when compared to the flawed gene-matching logic. This revision enables identification of 2 genes at a given location, a feature of key biological importance.

The data processing workflow is on the Aligning Genes to position with overlap notebook on GitHub.

1.1.2.8 GF

Genotype Frequency indicates the frequency of a particular genotype: homozygous reference, homozygous alternate and heterozygous (ref|alt). Allele counts of each heterozygous and homozygous alt provided by the vcf formed the basis of allele frequency calculations. These allele counts helped to compute genotype frequencies according to Hardy-Weinberg equilibrium. All calculations are in the Genotype Frequency Calcuations notebook.

1.1.2.9 PH_POS

Phased position signals the position of phased genotypes.

1.1.2.10 POS

Genomic position of given SNP. Used as an identifier of SNPs as position is unique in the given dataset.

1.1.2.11 REF

Reference allele for the given SNP, responsible for identification of the alternate allele in DAF calculation.

1.1.2.12 RS_VAL

Refers to the dbSNP v153 RS numbers, matched to SNPs by genomic position.

1.2 dbs

A database created using SQLAlchemy, which is used to retain user information entered on the register.html page (1.4.5) and structured according to the user schema model defined in the models.py file (1.5.6).

1.3 Static

The static folder contains external stylesheets, image files and the CSS code that's specific to this development project.

1.3.1 external

This folder includes the externally written styling for specific icons – including logo's, image boxes and icons. The website has been constructed using the latest version of bootstrap (v5.1.3) - a front-end open-source toolkit.

1.3.2 img

A static folder holding the images displayed on the webpage, including the main background image, as well as the Queen Mary University of London logos.

1.3.3 style.css

A local style.css file has been written to augment the style that has been imported via the external style sheets (1.3.1). This file controls how elements like the input text boxes or the submit buttons look and interact, as well as influencing the indentation on lists and displaying the various images (1.3.2).

1.4 Templates

The templates folder contains the html files for each page on the website.

1.4.1 about.html

The about.html file helps to generate the about page which is a general overview of the purpose of the application and what it does. This page also includes a link to an informational video about the 1000 Genomes project.

1.4.2 base.html

The base.html file defines global variables for all the other html pages. This includes the css files, the navigation bar and message handling

1.4.3 contact.html

The contact.html file outlines the structure of the contact page which includes contact information and a form where enquiries can be sent by the user. An interactive google maps iframe targeting to QMUL is included for visual aid. The contact form is not yet functional but could easily be altered to work if this were rolled out as a full web server.

1.4.4 login.html

The login.html is responsible for the login page where login details are input into a form which passes these variables to auth.py for verification.

1.4.5 register.html

The register.html file is responsible for displaying the registration form where users can register for an account.

1.4.6 userguide.html

This file contains the HTML code on the *user guide*. A user guide was chosen to be added as a page on the website because it allows the users to understand the summary statistics that are being used, and how to interpret the results shown from a biological aspect. Furthermore, it gives the user a brief description of the dataset used, and how to input the queries and the output format of the queries.

The *user guide* page was set up as a responsive fixed width container, so it is easily readable for the user. Five headings were set up as rows, the headings are: Data Source, Input Format, Summary Statistics, Interpretation of Summary Statistic Results, and Output Format

The Data Source section contains links to where the data was gathered, in order for the patient to gain a better understanding of them. The hyperlinks take the users to the 30x dataset, genotypes dataset, and the phased genotype dataset.

The Input Format section presents the different queries the users can input separated into three different parts. The general query explaining the user can input a gene name, rsID, and the start and stop position of the SNP. The population query explaining the 5 different populations the user can select (AFR, AMR, SAS, EAS, EUR). The summary statistics query which explains and displays the summary statistics the users can select.

The Summary Statistics and Interpretation of Summary Statistic Results sections in the user guide, gives a brief explanation to the users of what the summary statistics used are and how to interpret the results that show for each statistic from a biological aspect.

The Output Format section shows the users how the results will be displayed, it mentions how there will be 5 different types of tables, for the summary statistics and explains the columns they contain. It also mentions to the user how for three of the summary statistics they are presented as a sliding window.

1.4.7 server.html

The server.html generates the input page of the 1kGP SNP browser where all input details from rs number to populations or statistics of choice are entered. Acts as a user facing interface for users to enter details of choice.

The page also includes logic that processes files that are communicated from the web_server.py file (1.5.9). If a user is logged in the page converts these files into unique links which the user can use to view prior query results.

1.5 Python Files

1.5.1 __init__.py

This file contains the python code for initializing the app server that the website runs on. It is called by main.py (1.5.9). A SQLAlchemy database is created if one does not already exist and registered to the dbs folder (1.2).

The function “create_app” initializes the Flask web application and assigns a secret key as well as an upload path to direct user-generated files. The user database initializes within this flask app structure and the html pages included within the templates folder (1.4) are routed and registered using the *Blueprint* function of Flask.

The *flask_login* function *LoginManager* is used to associate the app with the authentication functions encoded within auth.py (1.5.2). This enables the web-app to always track whether the user is currently logged in or not.

Lastly, the Dash app is instantiated over the flask app. Initially, a blank dashboard is constructed using the *init_dashboard* function imported from dashboard.py (1.5.4). This decision was made to avoid the dashboard requiring data on initial start-up of the app, it therefore also allows the website to launch faster.

1.5.2 auth.py

The routing logic for users registering, logging in and out is contained within this file. These login processes are handled by the Flask *LoginManager* that was instantiated on initialization of the app (1.5.1). The routes are handled so that the logout option is only enabled when the user is currently logged in.

When a new user is added to the database their password is encryted using the ‘sha256’ password hashing method provided by the werkzeug module. This provides an extra layer of protection to user data that is stored in the SQLAlchemy database.

1.5.3 basic_pgs.py

Contains basic routing logic for the home, about, contact and user guide pages. These pages only require that the routing redirects the user and renders the relevant html page and there is no ‘POST’ option included as there is no user interactivity coded into these pages.

1.5.4 dashboard.py

This file contains the functions used to process the data stored by the user’s query and display it within the Dash app framework.

The *get_data* function takes in the unique id of the user (the ‘uid’ parameter) and loads the data that has been created when the query was generated within the sever.html page (1.4.8) and handled by the web_server.py file (1.5.9).

The web_server.py file redirects the user to a specific url based on the users submitted query id and where or not they are logged in, this is explained in more detail in section 1.5.9. This information is passed to the *uid_from_url* function within dashboard.py which then extracts which folder the data has been stored in. This information is then used to retrieve the information for the plots and tables.

The *init_dashboard* function is called within the __init__.py file (1.5.1) and initializes a dash app by passing in the current Flask server. The function assigns a prefix called ‘/results/’ to each redirect and relates the layout of the app to the URL that is being passed using the code below:

```
dash_app.layout = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])
```

This ensures that on initial startup the dash app remains blank, and the dashboard is only generated once data has been passed to it via the redirect to a query-specific URL.

Dash Bootstrap Components are used by the *display_data* function to orientate the tables and charts into separate “cards” that determine the layout on the page. The main table is arranged so that basic information such as gene position, reference and alternate allele are displayed alongside either allele frequency, genotype frequency or derived allele frequency data. This decision was made to improve the usability of the platform, given the extensive amount of information that needed to be displayed. Export buttons allow each data table to be extracted separately.

The *init_callbacks* function contains the dash app callbacks that determine how the dash app changes when provided with data. Here the information is rendered, and the plots and charts are constructed. The charts are displayed depending on whether the user has made selections for window, step size and at least one population. Each plot is styled similarly using the *plotly_fmt* function, which specifies parameters, such as gridlines or font size, which are included within the Plotly Express package that is being used to construct the charts.

Table style is also controlled by the *create_small_table* (for the summary statistics and Fst tables) and the *create_data_table* (for the main SNP data) functions. The *table_type* function ensures that the column data is assigned the right datatype so that filtering will work correctly.

1.5.5 data_proc.py

Herein lies the main function for processing the user’s query and outputting the data which is needed to construct the dashboard in dashboard.py (1.5.4). During the loading and filtering of the data the storage registry is set to disk rather than RAM using the code detailed in Data Pipeline (1.1.1).

Static paths are directed to the Zarr data, the population data, and the gene aliases dictionary. The *filter_data* function then takes in the information included within the user query that is passed to it by the *web_server.py* file (1.5.9) - including chromosome, start and end positions, gene name and rs values (if selected), as well as the chosen summary statistics and populations.

Relevant fields to load are determined according to these input parameters and the SNP data is then loaded, including the genotypes and phased genotypes. The SNP positions in both the main data and the phased data are loaded separately and are used to track data slicing throughout.

Filtering of this data then occurs based on whether the passed parameter strings are empty (I.e., “ ”) or not. If they are not empty, then the data is sliced according to boolean masks created by the *create_index_mask* function imported from stats_funcs.py (1.5.8). For example, if a gene name is specified, the function returns of boolean mask of all SNPs – being true for only the positions of the gene in the chromosome. The data is then reduced according to this mask.

Throughout filtering the data, the scikit allel package is used to ensure the arrays remain chunked and compressed, this is done using the *subset_G_array* function also imported from stats_funcs.py (1.5.8). Whilst these arrays can be treated like NumPy arrays, doing so will cause them to be loaded into RAM each time filtering occurs. Keeping them compressed throughout drastically reduces memory requirements and speed during processing.

After the data has been filtered according to the user’s specifications a check is performed to test whether there are any SNPs within this remaining region. If not, a string message is assigned to the “stats_df” variable and this is returned to the web_server.py file (1.5.9) to be displayed to the user as "Query returned no matching SNPs.".

If there are matching SNPs then the *PopulationFiltering* function from stats.py (1.5.8) is called to return Pandas dataframes for the summary stats and Fst calculations for that region, as well as the allele counts and the genomic positions of any segregating variants between the populations selected. This information is then passed back to the web_server.py file to be stored and later used to display the dashboard.

1.5.6 models.py

The *flask_login* module requires that a user is given the following properties – has an *is_authenticated*, an *is_active*, an *is_ananymous* and a *get_id* method. In this file the UserMixin class is used to assign these properties to a “User” class that is generated and stored in the SQLAlchemy database whenever a new user is registered through register.html (1.4.5).

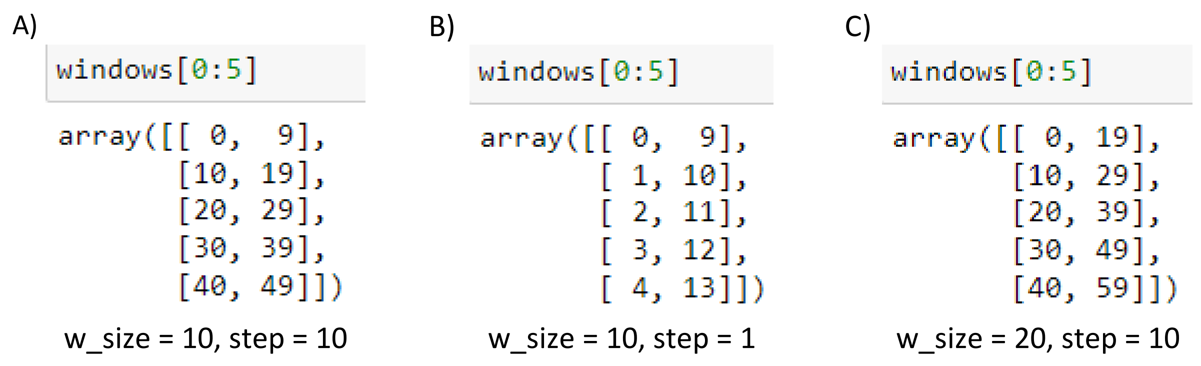
1.5.7 plot_funcs.py

This file contains the python code for all the summary statistic functions that are plotted as sliding windows (Tajima's D, Nucleotide diversity, Watterson Theta, Fst). For the functions to work NumPy, pandas, and scikit allel were imported.

The "*get_windows*" function is used to set the window size for the sliding window plots of the summary statistics. It takes as parameters the segregated positions, the window size and step. The function returns the start and end positions of the segregated variants selected by storing them as variables. The range between the two positions is stored in a NumPy array. This function then calculates the window size depending on what the user specifies by using the following code which utilizes the scikit learn *moving_statistic* function:

```
allel.moving_statistic(pos, statistic=lambda v:[v[0],v[-1]], size=w_size, step=step)
```

An example of the impact of the window size (w_size) and step parameters can be found in Figure 5.



A)

```
windows[0:5]

array([[ 0,  9],
       [10, 19],
       [20, 29],
       [30, 39],
       [40, 49]])
```

w_size = 10, step = 10

B)

```
windows[0:5]

array([[ 0,  9],
       [ 1, 10],
       [ 2, 11],
       [ 3, 12],
       [ 4, 13]])
```

w_size = 10, step = 1

C)

```
windows[0:5]

array([[ 0, 19],
       [10, 29],
       [20, 39],
       [30, 49],
       [40, 59]])
```

w_size = 20, step = 10

Figure 5: *An illustration of the impact that the parameters w_size and step have on the windows that the summary statistics in the charts are calculated across.*

The "*remove_nans*" function is used so all the nan values are removed if there is the absence of SNPs in a given window. It takes in the parameters of x and y, where x accounts for the SNP positions, and y accounts for the summary statistics values. Within the function the values returned from the summary statistics (y) are put into a mask.

The "*plot_data_seqdiv*" function is used to calculate the values for nucleotide diversity and plot it as a sliding window. Takes as parameters the segregated positions, the segregated allele count for one population or two populations that the user has specified, the window size, and the step. The "*get_windows"* functions is stored in a variable as windows, which is then taken as a NumPy array and the blocks in the centre from the windows act as the x axis. To obtain the nucleotide diversity values for the y axis for the sliding window, the following code was used which utilizes the scikit learn *windowed_diversity* function:

```
allele.windowed_diversity(seg_pos, ac-seg[pop1][:], windows=windows)[0]
```

Both x and the y axis are reduced to where only windows and values where SNPs are present by using the function “*remove_nans*”. Both the x and y values are then stored as a dictionary. When the populations are selected by the user the function will return the sliding windows of the nucleotide diversity as a data frame. If two populations are selected it will return the sliding window with two separate lines form the y axis each representing the nucleotide diversity values for each population, if only one population is selected then it will just show one line representing the nucleotide diversity values for the selected nucleotide diversity.

The “*plot_data_watt_theta*” function is used to calculate a further measure of genetic diversity by using the Watterson’s theta estimator method and plotting the data in sliding windows. The “*plot_data_tajimas*” function is used to calculate the values for Tajima's D which is a statistical test for neutrality, and plots the values as a sliding window.

Both functions follow the same process as the function “*plot_data_seqdiv*” shown above, but utilizes either the *windowed_watterson_theta* or *windowed_tajima_d* functions from scikit allel to calculate the y axis values.

The “plot_data_fst” function is used to calculate the Fst values for comparing two populations that have been selected by the user. It takes as parameters the segregated positions, segregated allele count, first population, second population, window size, and step. It functions similarly to the other functions in this notebook but requires two populations rather than being able to function with just one. It utilizes the *windowed_hudson_fst* function from scikit allel to calculate the y axis values. Hudson’s estimator for Fst was used as it is insensitive to the proportion of sample size and does not regularly disregard the Fst, due to this Bhatia et al., 2013 suggested the use of Hudson’s estimator to calculate the Fst values for pairs of population, (Bhatia et al., 2013).

1.5.8 stats_funcs.py

Stats_funcs contains all the sub-functions that generate the data tables presented in the dashboard. First, two dictionaries are defined which allows the dataframes created to use the full names for the statistics and populations (I.e., ‘AFR’ becomes ‘African’ and ‘hap_div’ becomes ‘Haplotype Diversity’).

The *subset_G_array* function uses the scikit allel *chunked.core.subset* and *chunked.core.compress* functions to ensure that the arrays stay compressed throughout data filtering – this is integral to the *filter_data* function within data_proc.py (1.5.5).

Create_index_mask creates a boolean mask beween a start and stop index in the data provided. This mask is later used to subset the data by the True values in the mask.

The *“haplotype_diversity”* function estimates the haplotype diversity statistic. This function takes in a phased genotype chunked array as its parameter and returns haplotype diversity as a floating-point value. The variable *"genotype_array"* will hold a numpy array and is utilized to compute a "*haplotype_array*".

When a *"haplotype_array"* is instantiated, it contains haplotype data as a two-dimensional numpy array of integers. The first dimension aligns to the variants genotyped *(n_variants)*, while the second dimension is equivalent to the number of haplotypes *(n_haplotypes)* . Finally, "*allel.haplotypediversity*" calculates haplotype diversity from the instantiated "*haplotype_array"* (scikit-allel - Explore and analyse genetic variation — scikit-allel 1.3.3 documentation, 2022).

The SummaryStats function calculates nucleotide diversity, Watterson's Theta, Tajima's D and Haplotype Diversity for all SNPs within the filtered query range, according to whether the user has specified those stats to be returned. The scikit allel functions *sequence_diversity, watterson_theta, tajima_d* and the *haplotype_diversity* function above perform these tasks when given the positions of segregating variants and population allele counts.

Create_data_table converts the statistics generated by SummaryStats into a Pandas dataframe, it fills any NaN values with '*' and rounds numeric values to 4 decimal places.

The *PopulationFiltering* function prepares the variants, genotypes and phased genotypes data for input to the functions listed above. It subsets the genotypes table according to the populations selected by the user and uses the *count_alleles_subpops* function from scikit allel to produce an AlleleCountsChunkedTable containing the allele counts for every population.

A boolean mask is then applied over the allele counts table to determine whether the SNPs are segregating among the populations selected. If there are no segregating SNPs then this is returned as an error message to the user. If there are segregating SNPs then SummaryStats is used to calculate the two genetic diversity metrics, Tajima's D and haplotype diversity.

Lastly, the combinations function of the itertools module is used to obtain all combinations of the different populations. These combinations are then fed through a for loop to calculate the Fst value for every pair of populations. The statistics and Fst dataframes, as well as the allele counts and SNP positions of the segregating variants are then returned to the data_proc.py function (1.5.5).

1.5.9 web_server.py

The web_sever.py file contains the routing logic for the server.html page (1.4.8). It uses the *current_user* function from the flask_login module to first check whether the user is logged in or not. It then determines which save location to save any query data generated in. If the user is not logged in, any data will be saved in the 'guest' folder inside the 'uploads' folder. If the user is logged in, their login ID will be returned by the *current_user* function according to when they were registered within the SQLAlchemy database (e.g., the first user registered will have a login ID of '1').

This information is then used to locate any saved data from prior queries, which is passed to server.html to display as query hyperlinks if the user is logged in.

The "if request.method == 'POST'' code runs only when a user has submitted information to the '/server' route - I.e., has submitted a query. Information from the forms is then stored as separate variables and various error handling is used to ensure the user queries match the expected search criteria (I.e., Chromosome stop position is after the start position).

If the user is not logged in, this 'POST' method triggers a deletion of the files being stored in the guest folder that have been created by any prior queries – this ensures that non-logged in users cannot permanently store data.

Two lists are then made to indicate which summary stats and populations the user has selected, and these are fed as parameters, along with the other query information, to the *filter_data* function in data_proc.py (1.5.5). The following information is returned from this function:

- stats_df – A Pandas dataframe containing the calculated summary stats
- fst_df - A Pandas dataframe containing the Fst values for the population combinations
- ac_seg – An AlleleCountsChunkedTable constructed using scikit allel, containing the allele counts for each population at each SNP position.
- seg_pos – The SNP positions of all segregating variants in the populations selected.
- variants – A VariantChunkedTable containing information about allele frequencies, genotype frequencies and derived allele frequencies for each SNP in the query range.

These are then saved to the appropriate folder, given the user's login status, to be retrieved by the dashboard.py functions (1.5.4) to be displayed. Lastly, a specific route, directing the dash app to the location of this data is generated and then flask function *redirect* is used to direct the user to that URL.

1.5.10 main.py

The main file that is called to run the web application. It imports the *create_app* function from __init__.py (1.5.1) and uses this to launch the server. The `threaded` parameter makes sure that multiple requests are handled by separate threads. `host` and `port` parameters define the routing for requests, this is key for handling requests from the docker container after hosting on Cloud Run.

1.6 Other files

1.6.1 ignore files

These include the .gitignore, .gcloudignore and .dockerignore files which are responsible for defining a list of files within the app directory that are ignored when pushing to GitHub, Google cloud and building docker images respectively.

The files that are ignored currently include the dbs folder (1.2) and the uploads folder (containing any stored user and guest data). The database folder is excluded so any senstive user information is not shared on Github and the uploads folder is ignored to prevent unecessary uploads of saved information.

1.6.2 requirements.txt

The minimum requirements for running this software are detailed within the requirements.txt file. This includes the modules required and the versions of those modules that have been used during this development project.

1.6.3 Dockerfile

The Dockerfile is a configuration script which defines parameters for building a docker container. The *`FROM`* variable defines the official Python docker image and specific platform used to build the image. *`ENV PYTHONBUFFERED True`* forwards the port assignment to the python application which is defined in the app.run parameters in the main.py file. The *`RUN`* blocks install dependencies and requirements and the *`CMD`* block executes Gunicorn which runs the webserver.

2 Limitations of current software and potential improvements

2.1 Limitations

2.1.1 Zarr Storage

Zarr data storage format is a relatively recent technology, enabled by modules such as Dask, Blosc and Scikit Allel. While it has proven to work effectively during this development project, and the storage demands of the data (~900Mb) and the memory demands of a search (~100-600Mb) are not extensive, there is no guarantee that this is the *best* solution for this particular task.

Given more time it would be appropriate to explore alternatives, such as using relational databases, or different compressor technologies, and benchmark their performance against one another.

Furthermore, as it is less well-known or widely used currently, using this sort of technology may discourage collaboration on the project or be a limiting factor to people understanding its function.

2.1.2 Python Flask Web Deployment

While Flask offers an easy-to-approach web development framework, it has been chosen for this project based on the current coding capabilities of the developers rather than because it is necessarily the best tool for the job. In the current web-app there are a mix of languages being used as python files respond to requests that return information from either JavaScript, CSS or HTML files.

JavaScript is often the language of choice for web developers and remains the most popular coding language to use in these sorts of tasks, as it can solely handle both front and back-end processes. Given more time for development it may be more effective to design the whole application in JavaScript, this would also likely aid the compatibility of the application to run on different browsers or platforms (e.g., mobile).

2.1.3 Extensive Error Testing

Error testing was performed by iterating through lists of possible user queries - I.e., lists of possible gene names, aliases, rs values and various start and end positions - and recording whether an error was raised during the data processing stage. Any errors were then investigated as stand-alone queries before fixes were made to the code.

While this approach is robust, it is far from extensive, and to deploy a real server this testing would have to be extended across many more combinations of different inputs. Given the time constraints it was not possible during this project, but we have tested enough to have reasonable confidence in the output given this is a development server.

2.1.4 Derived Allele Frequency

By definition, a derived allele is a variant allele that has arisen since the last common ancestor i.e., a mutated allele. Derived allele frequency (DAF) is therefore represented as the proportion of alleles in a population at a specific genomic position which do not match the ancestral allele. Because the dataset retains bi-allelic SNVs only, DAF was calculated based on the assumption that the sum of reference and alternate allele frequencies is equal to 1. This assumption is false for multi-allelic SNVs with multiple alternate alleles. If multi-allelic sites were included, a more robust method of calculation would involve generating individual allele counts using the genotype table and then evaluating DAF.

2.2 Improvements

2.2.1 Additional Summary Statistics

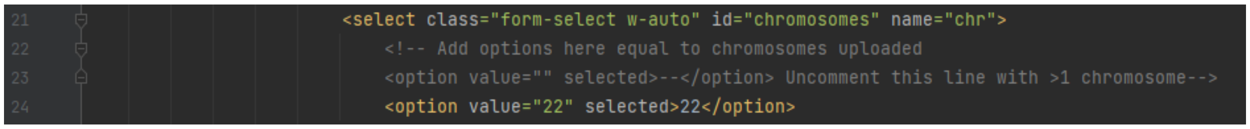
This research project had to be done in the space of 6 weeks, giving the team little time to add more diverse ways to calculate summary statistics. For example, to calculate the genetic diversity we chose to use nucleotide diversity and Watterson’s theta. With a bit of extra time, we could have added the function *allel.mean_pairwise_difference* from scikit learn, as it is another way to measure genetic diversity, since it calculates the mean number of pairwise differences between chromosomes within a population. To add more summary statistics results would allow the user to have better comparison between them.

Furthermore, some summary statistics (Tajima’s D, nucleotide diversity, Watterson’s theta, and Fst) are plotted as sliding windows. In order to give the user different ways of visualizing the summary statistics we could have utilized plots such as PCA or UMAP methods.

2.2.2 Additional Chromosomes

The addition of more chromosomes would be integral to the functionality of this as a real server. To aid a future developer in doing this we have written Jupyter Notebooks describing the key processes in downloading, storing, and accessing the data. A developer would simply need to follow these guides to add more data as separate Zarr folders for each chromosome.

The functions have been written with additional chromosomes in mind and comments have been left in the code where items would need to be altered, see Figure 6 for an extract of the server.html code.



```
<select class="form-select w-auto" id="chromosomes" name="chr">
    <!-- Add options here equal to chromosomes uploaded
    <option value="" selected>--</option> Uncomment this line with >1 chromosome-->
    <option value="22" selected>22</option>
```

Figure 6: Commented code to alter the server.html file to incorporate more chromosomes

2.2.3 Include Country-Level Populations

1kGP SNP currently includes populations on a continental level. Given more time, populations at a country level could be incorporated and the user could be given a choice when querying at which level they would like to inspect for differences.

The web app has been designed so that incorporating this functionality would not require wholesale code changes. The population data file currently includes detail at this level and so to alter the population filtering the following line of code in the *PopulationFiltering* function of stats_funcs.py (1.5.8) could be altered:

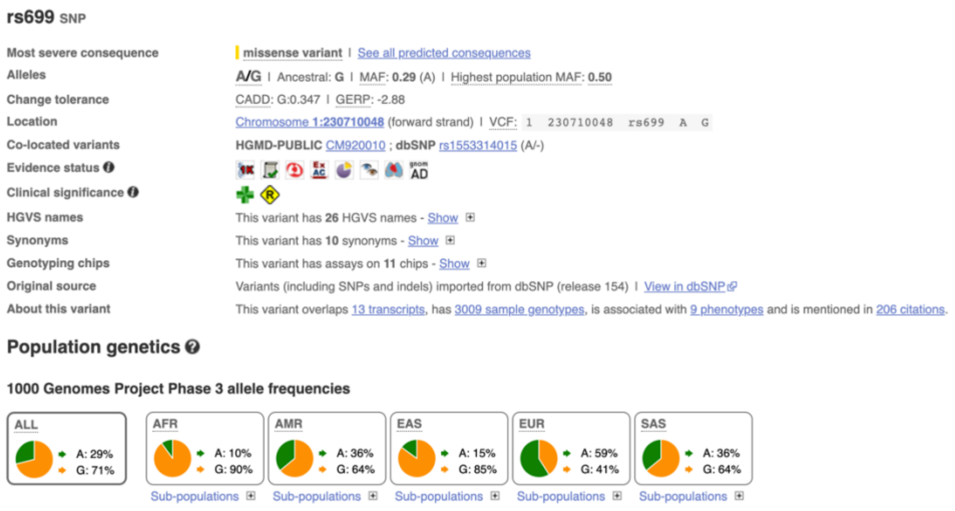
```
sample_selection = pop_data.super_pop.isin(pops).values
```

If ‘super_pop’ (bold in code above) is changed to ‘pop’ the function would then filter data on a country level instead. Flow-through changes would then have to be made in the dashboard to incorporate the correct space needed for the Fst table and the dropdowns for the charts. Despite this not being included in the current functionality, it is one we have considered and coded with in mind, so that this would not be difficult for a future developer to incorporate.

Allele frequencies are currently included within the 30x dataset by default for each continental population. If functionality were extended to include country-level populations then allele frequencies would have to be pre-computed separately for each country.

2.2.4 Addition to Visualizations Page

With extra time there could have been more additions to the visualizations page. To give a better visualization for the user of the values for each population of the summary statistics allele frequency, genotype frequency, and derived allele frequency, they could have also been shown in pie charts. Similarly, to ensembl as shown on figure 7.



rs699 SNP

Most severe consequence	missense variant I See all predicted consequences
Alleles	A/G I Ancestral: G I MAF: 0.29 (A) I Highest population MAF: 0.50
Change tolerance	CADD: G:0.347 I GERP: -2.88
Location	Chromosome 1:230710048 (forward strand) I VCF: `1 230710048 rs699 A G`
Co-located variants	HGMD-PUBLIC CM920010 ; dbSNP rs1553314015 (A/-)
Evidence status	
Clinical significance	
HGVS names	This variant has 26 HGVS names - Show
Synonyms	This variant has 10 synonyms - Show
Genotyping chips	This variant has assays on 11 chips - Show
Original source	Variants (including SNPs and indels) imported from dbSNP (release 154) I View in dbSNP
About this variant	This variant overlaps 13 transcripts, has 3009 sample genotypes, is associated with 9 phenotypes and is mentioned in 206 citations.

Population genetics

1000 Genomes Project Phase 3 allele frequencies

ALL	AFR	AMR	EAS	EUR	SAS
A: 29%	A: 10%	A: 36%	A: 15%	A: 59%	A: 36%
G: 71%	G: 90%	G: 64%	G: 85%	G: 41%	G: 64%
	Sub-populations	Sub-populations	Sub-populations	Sub-populations	Sub-populations

Figure 7: Pie charts of the population genetics 1000 Genome Projects Phase 3 allele frequencies for each population from rs699 in Ensembl.

2.2.5 Improved User-Interface and Experience

There are several main user-interface features that we would further develop if this were to be deployed. Firstly, in the case where a user searches for a single RS value, the summary statistics are often of little value or cannot be calculated (e.g., Tajima’s D). Currently they are still displayed despite this. Either the web-app would need to include information when this was the case to explain the reasons why certain statistics cannot be calculated, or the data should be omitted entirely.

Secondly, when the user selects a window then they can observe certain areas of interest by inspecting the charts. We would implement functionality that would let the user select a specific chromosomal area of interest in the graph which would then allow the extraction of the summary stats only for that region.

Currently all the statistics are rounded by default to 4 decimal places. It would be beneficial to either give the user control of this or to allow the user to download data that contains non-rounded figures. Also, the derived allele frequency columns are blank when there is no Ancestral Allele in our database (and so the DAF cannot be calculated). Ideally tooltips would be present to inform the user why data is not being displayed in cases like these.

Lastly, search queries are limited to 1Mb in size as it was felt that larger searches may be less insightful to the user and would place an unnecessary demand on any eventual server. Despite this, it may be worthwhile to record each time users try to enter queries longer than this length to ascertain the needs of the user more precisely. Then this limit could be adjusted over time accordingly if there was sufficient demand.

References

Byrska-Bishop, M. *et al.* (2021) 'High coverage whole genome sequencing of the expanded 1000 Genomes Project cohort including 602 trios'. bioRxiv, p. 2021.02.06.430068. doi:10.1101/2021.02.06.430068.

Scikit-allel.readthedocs.io. 2022. scikit-allel - Explore and analyse genetic variation — scikit-allel 1.3.3 documentation. [online] Available at: <https://scikit-allel.readthedocs.io/en/stable/index.html> [Accessed 27 February 2022].

Hodgkinson, A. and Eyre-Walker, A. (2010) 'Human Triallelic Sites: Evidence for a New Mutational Mechanism?', Genetics, 184(1), pp. 233–241. doi:10.1534/genetics.109.110510.

The Hardy-Weinberg Principle | Learn Science at Scitable 2010. Available at: https://www.nature.com/scitable/knowledge/library/the-hardy-weinberg-principle-13235724/ (Accessed: 27 February 2022).

Chen, C.-H., Pan, C.-Y. and Lin, W. (2019) 'Overlapping protein-coding genes in human genome and their coincidental expression in tissues', *Scientific Reports*, 9(1), p. 13377. doi:10.1038/s41598-019-49802-w.

Bhatia G., Patterson N., Sankararaman S., and Price A. (2013) 'Estimating and interpreting Fst: The impact of rare variants', Genome Research, 23(9), pp. 1514-1521. doi: 10.1101/gr.154831.113