Project Documentation

Team Duck Duck Go

Contents

Introduction

This web application prototype is designed to retrieve information on Single Nucleotide Polymorphisms (SNPs) seen in Type 1 Diabetes patients identified by Genome wide association studies (GWAS). The database will use information from the GWAS catalogue, along with population data from Ensembl and the 1000 Genomes Project and functional information and Gene Ontology information obtained through Ensembl's VEP tool which is all is retrievable through a user friendly interface through the input of an rsID, Chromosome position or a Gene name. The site also allows the user to calculate Linkage Disequilibrium (LD) of SNPs selected for each population producing a text file containing the LD values and plot these values as a LD heatmap. The user is also able to enter multiple SNPs and return a Manhattan plot of p-values.

Prerequisites:

Package	Version
Python	3.10.10
Flask	2.2.3
Flask-WTF	1.1.1
bokeh	3.0.3
Jinja2	3.1.2
pandas	1.5.3
matplotlib	3.7.0
numpy	1.24.2

Functions used throughout:

A number of functions were used throughout the code such as...

```
# some python code will go here...
```

Structure

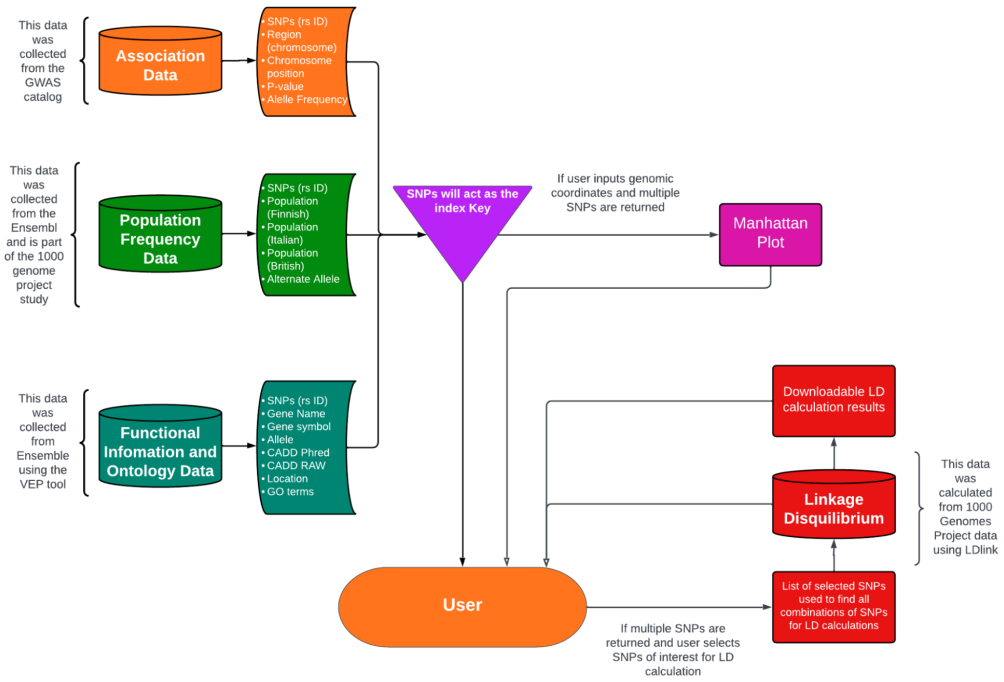

Figure 1: Structure of the data flow

GWAS

This information was downloaded from the GWAS catalogue where a TSV file was downloaded and then trimmed:

```
# some python code will go here...
```

This code uses pandas to open the TSV file, creates a dataframe called data, and removes any special characters from the column names, as SQL does not interact with special characters very well.

```
# some python code will go here...
```

The data frame is then filtered further so that it only has data that references T1D in the "disease_trait" column so that only T1D data remains in the dataframe. Next all SNPs that don't have rsIDs are removed, as some cells had incompatible data in this column.

```
# some python code will go here...
```

The dataframe is then trimmed again so that it contains only the columns of interest "snps", "region", "chr_pos", "chr_id", "p_value", "mapped_gene".

```
# some python code will go here...
```

Duplicate mapped_gene information is then removed.

```
# some python code will go here...
```

Next the column 'snps' was renamed to 'rsid' and made the change directly to the dataframe by setting inplace=True.

Variant frequency data by population

SNP variant frequency data used in the database was obtained from the 1000 Genomes Project via Ensembl. Variant frequencies were obtained for the Finnish, Toscani Italian and British 1000 Genomes Project populations for each T1DM SNP in the EBI GWAS dataset. Finnish data was chosen as the greatest global incidence of T1D occurs in Finland due to Colloidal amorphous silica (ASi) present in the Finnish environment. The Toscani Italian population was chosen due to evidence of high rates of T1D in Northern Italy due to genetic risk. The British (in England and Wales) population was chosen due to increasing variance of T1D incidence in these regions.

Ensembl REST returns a list of dictionaries containing data for each study population. This function was used to request allele frequency data for 1000 Genomes Populations by SNP using a list of SNPs extracted from the GWAS dataset.

```
def variant_frequency_API(rsID):
    import requests, sys

    server = "https://rest.ensembl.org"
    ext = f"/variation/human/{rsID}?pops=1"
    r = requests.get(server+ext, headers={ "Content-Type" : "application/json"})

    if not r.ok:
      r.raise_for_status()
      sys.exit()

    decoded = r.json()
    return decoded
```

Functional information and Gene Ontology

We have used Ensembl's Variant Effect Predictor web tool to gather the Functional and Ontology data by submitting a job with the rsIDs as input from the above GWAS TSV file. After running the job, we get an output of text file with columns like:

The VEP file provides detailed information about the functional and ontological consequences of genetic variants, including their impact on genes, proteins, and pathways.

We have converted the text file to a tsv file and trimmed down the file to include only the rsID, Alleles, CADD_PHRED and CADD_RAW scores columns for the functional data and the rsID, location, gene, symbol, GO terms columns for the Ontology data. We have further used these TSV files in our database.

CADD (Combined Annotation Dependent Depletion) is a tool used for predicting the potential harm caused by genetic variants. The tool generates a score that indicates the likelihood of a variant being deleterious.

One advantage of CADD over SIFT and PolyPhen is that CADD integrates a larger and more diverse set of functional annotations. It also considers the effects of variants on non-coding regions of the genome, which can be important for understanding the functional consequences of variants that are not in protein-coding regions.

Linkage Disequilibrium

Linkage disequilibrium (LD) is the degree of non-random association of the allele of one SNP with the allele of another SNP within a population. LD is typically measured by two metrics: D' and r^2.

D' is the normalised values of D, the coefficient of linkage disequilibrium, where A and B are alleles of two SNPs in different loci:

$$D' = P_A - P_A P_B$$

r^2 is the correlation coefficient between two loci:

$$r^2 = \frac{D^2}{p_A(1-p_A)p_B(1-p_B)}$$

Collecting data

Linkage disequilibrium data was obtained from LDlink using the LDmatrix tool. D' and r^2 values for SNPs were calculated using 1000 Genomes Project data for all three populations. LD data was obtained by inputting a list of SNPs from the same chromosome and selecting the population which would be used for allele frequency data for LD calculations. LDmatrix would produce two text files containing a matrix of results for D' and r^2 values calculated between all SNPs pair combinations in the input list. This was performed separately for each population. Some SNPs did not have any LD data due to a lack of allele frequency data for those SNPs in the 1000 Genomes Project.

LD datasets containing D' and r^2 values for Finnish, Toscani and British populations are loaded in with pandas as separate dataframes. Each dataframe has their index set to the first column which contains SNP rsIDs.

```
# some python code will go here...
```

This function uses the itertools combination function to create a list of tuples containing all unique pairs of SNPs possible from a list of SNPs. The list is then separated into two lists containing the first and second element of each tuple.

```
# some python code will go here...
```

An empty dataframe is created to be filled with rows containing data from all six dataframes. This loop uses the two lists created from the SNP list to index each dataframe and extract the respective LD value. These are used to create a list which is converted into a single row pandas dataframe which is added to the empty dataframe using pandas concat until data for all relevant pairwise LD calculations have been added. The completed dataframe is then outputted as a TSV file.

```
# some python code will go here...
```

Outputting LD results

When a user searches by gene name or chromosomal coordinates, if multiple SNPs are returned, a list of SNPs is used to filter the LD dataset for all rows with entries for all pairwise LD calculations of SNPs in the list and output a results dataframe.

Before filtering, the list is checked for any SNPs which are not in the LD dataset due to lack of LD data and any offending SNPs are removed from the list.

```
# some python code will go here...
```

The SNP list is then used to create two lists containing the first and second element of each tuple using the SNP_pair_lists() function defined earlier.

```
# some python code will go here...
```

The LD dataset containing all available data for pairwise LD calculations is loaded in with pandas and an empty dataframe is created for the filtered data. The pair of SNP lists are then used to index the LD dataset dataframe for all rows with pairs of SNPs relevant to the user's search query which are added to the LD results dataframe using pandas concat.

```
# some python code will go here...
```

LD heatmap plots

When a user searches by gene name or chromosomal coordinates, if multiple SNPs are returned, a list of SNPs is also used to extract LD values for all relevant pairwise SNP calculations to create a dataframe used to create a heatmap plot of LD values.

The LD dataset is loaded in with pandas and SNPs not present in the dataset are removed from the list of SNPs passed from the user query. The SNP list is then used to create two lists containing the first and second element of each tuple using the `SNP_pair_lists()` function defined earlier.

```
# some python code will go here...
```

An empty dataframe is created to be filled with LD values used to create the LD plot. The pair of SNP lists are then used to index the LD dataset dataframe and extract the LD value for all possible pairwise LD calculations from the SNP list. A row of LD values is created for each SNP where each column corresponds with the pairwise LD calculation with one of the SNPs from the list. Each row is added to the empty dataframe using pandas concat.

```
# some python code will go here...
```

The LD matrix dataframe is passed to the ld_plot function. The number of rows (n) is used to create a mask which will hide half of the heatmap to create a triangular plot. A coordinate matrix is also created to rotate the heatmap plot. The SNP list is used to create the axis labels located at the bottom of the plot. The function's title parameter passes a string which is used to determine the plot title.

```
# some python code will go here...
```

Manhattan Plot

A Manhattan plot is a type of scatter graph which displays P values of the entire Genome-wide association study (GWAS) on genomic scale. The P-values are shown in genomic order by chromosomal position on the x-axis. The y-axis shows the $-log10$ value of the P-value for each polymorphism.

A TSV file containing all SNPs in type 1 diabetes was produced. The Manhattan plot needs the P-value of a SNP and the chromosomal position. Using pandas, the $-logp$ and cumulative positions were generated.

```
import pandas as pd
import numpy as np

df = pd.read_csv('./TSVs/GWAS_T1D.tsv', sep='\t') # Make the table readable using
↪  pandas

# -log the P-values and column to the table
df['-logp']= - np.log(df['p_value'])

# Put variants in order by max position in chromosome
running_pos = 0 # moves all integers down so first pos is 0

cumulative_pos = [] # create list of new series for position in whole genome

for chrom, group_df in df.groupby('chr_id'): # Group the region in each chromosome
↪  together
    cumulative_pos.append(group_df['chr_pos'] + running_pos)
    running_pos+= group_df['chr_pos'].max() #tells us the last position in each
    ↪  chromosome

# Position of variant relative to whole chromosome, add column to the table
df['cumulative_pos'] = pd.concat(cumulative_pos)
df.to_csv('./TSVs/GWAS_T1D_add.tsv', sep ='\t')
```

The original file 'GWAS_T1D.tsv' is used to create an additional column for $-logp$ values and creating a column for cumulative positions by using the chromosome and chromosomal position values. The cumulative position is needed to show the position of the SNP in the entire genome, rather than its position in the chromosome.

```
# Separate by chromosome ID, and colour them
index_cmap = linear_cmap('chr_id', palette = ['grey','black']*11,low=1,high=22)

## Format figure
p = figure(frame_width=800,                # graph size
           plot_height=500,          # graph size
           title=f"SNPs in {SNP_req} for T1D",# Title added in html
           toolbar_location="right",
           tools="""pan,hover,xwheel_zoom,zoom_out,box_zoom,
           reset,box_select,tap,undo,save""",# Tool features added to make graph
            ↪  interactive
           tooltips="""
           <div class="manPlot-tooltip">
               <span class=tooltip-rsid>@rsid </span>
               <span class=tooltip-chrPos>@chr_id:@chr_pos</span>
           </div>
           """# Shows when mouse is hovered over plot
           )

# Add circles to the figure to represent the SNPs in the GWAS data
p.circle(x='cumulative_pos', y='-logp',# x and y-axis
        source=df,
        fill_alpha=0.8,# Transparency of plot
        fill_color=index_cmap,# Colour of plot
        size=7,# Size of plot
        selection_color="rebeccapurple", # Colour of plot when selected
        hover_color="green"
        )
```

Bokeh tools was used to make the graph user friendly. Circle plots were chosen for easy visibility and selecting of each SNP position. Hover feature was added to allow the user to hover mouse over each plot, which turns it green, and shows the rsID value next to the chromosomal position. The select feature allows users to select 1 or more plots by either clicking one plot, or holding shift and clicking plots for multiple plots, or using the box select tool to select plots in an entire section, and turns the plots purple. The zoom features allow the user to zoom into the graph and view plots that are close to each other or seem to be overlapping for more visual clarity. The reset and undo buttons allows the user to either go back to the original plot, or undo the previous action they had committed. The save tool will save the graph as

a png file with the title included. The plots were made slightly transparent to make overlapping SNPs more visible.

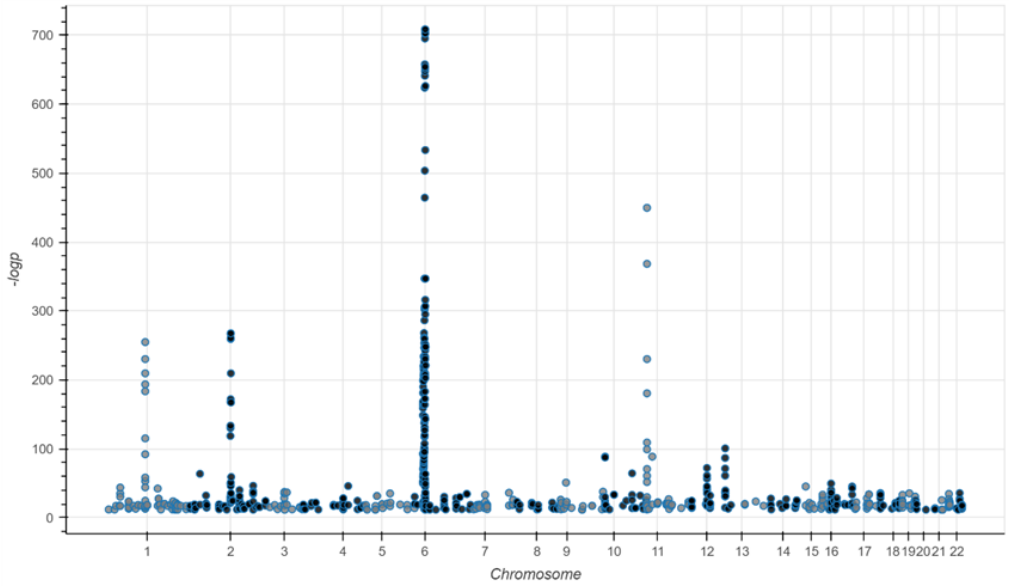

Figure 2: All GWAS SNPs found in Type 1 Diabetes shown in a Manhattan Plot. Users can view all SNPs for T1D, and plot the SNPs that they have searched for. This feature is available when multiple SNPs are retrieved for the region searched.

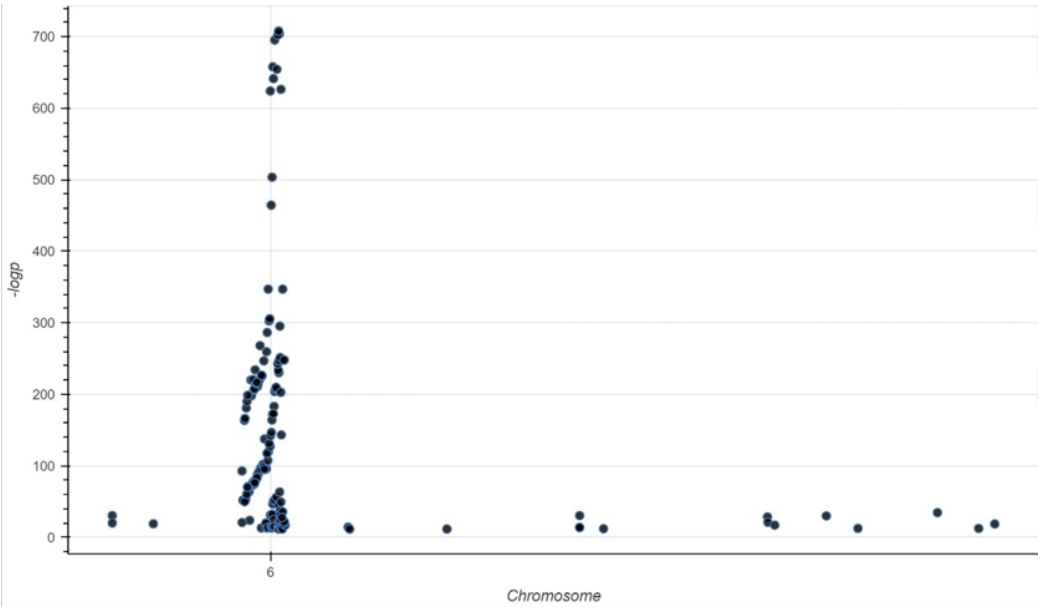

Figure 3: All GWAS SNPs for Type 1 Diabetes, Chromosome 6 only

Website

Flask

The main page was created using the following function. By adding the `@app.route()` decorator, the flask module can run the function whenever the root page is accessed. The function creates an instance of class `QueryForm`, which it uses to receive user input. The input is then formatted to better be understood by future functions, before the parsed request and the request type are passed to the next page as the subpage and URL parameters, respectively.

```
@app.route('/', methods=['GET','POST'])
def index():
        form = QueryForm() # Create form to pass to template
        SNP_req = None
        req_type = None
        if form.validate_on_submit():
                # Get request from text form
                SNP_req = form.SNP_req.data
                # Remove whitespace from request
                SNP_req = SNP_req.replace(' ', '')
                # Get request type from dropdown menu
                req_type = form.req_type.data
                return redirect(url_for('SNP', SNP_req =
                ↪  SNP_req,req_type=req_type))
        return render_template('index_page.html', form=form, SNP_req=SNP_req,
        ↪  req_type=req_type,debug=debug)
```

The following function first receives the `SNP_req` by reading the subpage name, and the `req_type` by making a `HTTP GET` request to retrieve the relevant argument. If the request type is set to automatically detect, the request is tested against a series of regex searches to determine the intended request type.

The `DBreq()` function is then called to retrieve the required information via SQL commands. This function will be elaborated upon in the following section.

```
@app.route('/SNP/<SNP_req>', methods=['GET','POST'])
def SNP(SNP_req):
        # Get type of information inputted
        req_type=request.args.get('req_type',default="empty_req_type")
        assert req_type != "empty_req_type", "request type is empty"

        if req_type == 'auto':
                if re.search(r'rs\d+',SNP_req):
                        req_type='rsid'
                elif re.search(r'\d:\d+', SNP_req):
                        req_type='coords'
                elif re.search(r'\w{1,10}', SNP_req):
                        req_type='geneName'
                else:
                        raise Exception("couldn't determine request type")
                        req_type= None
                if debug:
                        print("detected type:", req_type)
        if req_type!='geneName':
                SNP_req = SNP_req.lower() # Ensure snp name is in lowercase
                ↪  letters
        name=SNP_req
        reqRes,SNP_list=DBreq(SNP_req, req_type) # Make SQL request
        if reqRes: # If the response isn't None
                assert isinstance(reqRes, dict),"invalid db request return value"
                l=len(reqRes)
                if l==1:
                        name=str(list(reqRes.keys())[0])
                else:
                        name=f'{l} SNPS'
                return render_template('view.html', reqRes=reqRes, name=name,
                ↪  req_type=req_type, len=len(SNP_list),
                ↪  SNP_req=SNP_req,debug=debug,SNP_list=SNP_list)
        else:                                           # If SNP is not found:
                return render_template('not_found.html', name=name)
```

SQLite

To create the database, the `.to_SQL()` method from Pandas was used, which receives a dataframe of a TSV file, as well as an SQLite3 connection, and produces an SQLite compatible `.db` file

```
from db_scripts import *

gwas = getPath("T1D_GWAS_add_clean.tsv")
pop  = getPath("population_variation_noSpecial.tsv")
func = getPath('Func_trimmed.tsv')
ont  = getPath("GO_new.tsv")
DB=DBpath()

if os.path.exists(DB):       # If the file exists,
    os.remove(DB)            # delete it.

pdDB(gwas, "gwas",       {"rsid":"TEXT PRIMARY KEY"})
pdDB(pop,  "population", {"rsid":"TEXT REFERENCES gwas(rsid)"})
pdDB(func, "functional", {"rsid":"TEXT REFERENCES gwas(rsid)"})
pdDB(ont,  "ontology",   {"rsid":"INTEGER REFERENCES gwas(rsid)"})
```

The above code makes use of various functions from the `db_scripts.py` custom module:

```
import pandas as pd
import os
import sqlite3

_database = "snps.db"

def pdDB(tsv_path,table_name,dtype): # Adds tsv to SQL database
    conn = sqlite3.connect(DBpath()) # Opens (or creates) a db file
    cur = conn.cursor() # Sets cursor
    df = pd.read_csv(tsv_path, sep='\t')
    df.to_sql(name=table_name, con=conn, index=False, dtype=dtype)

def getPath(file,tsv=None): # Returns the absolute path to a file that is in same
↪  folder as script
    filenames=('tsv', 'csv', 'txt') # List of table storing files
    ext=file.split('.') # Splits by '.'
    ext=ext[-1].lower() # Uses this information to get extension of file
    path = os.path.dirname(os.path.abspath(__file__)) # Gets current path of file
    if (tsv==True) or (tsv==None and any([x == ext for x in filenames])): # If
    ↪  it's a table storing file:
            filepath = os.path.join(path,"TSVs",file) # Sets path relative to
            ↪  current file, inside 'TSVs' folder
    else:
            filepath = os.path.join(path,file) # Sets path relative to current
            ↪  file
    return filepath

def DBpath(): # Returns the absolute path to the database
    return(getPath(_database))
```

After connecting to the database file, rsIDs are extracted from the user's request, if necessary via cross-referencing with with GWAS dataset.

```
def DBreq(request, request_type, manPlot=False):

    ###### Setting up database connection ######
    filepath=DBpath()                    # Sets database path
    assert os.path.exists(filepath),"Database file not found"
    conn = sqlite3.connect(filepath)     # Opens db file
    cur = conn.cursor()                  # Sets cursor

    ###### Getting rsids from request ######
    if request_type=="rsid":
        request=request.split(',') # Split request by comma separator

    elif request_type=="coords": # co-ordinate (6:1234-5678) search
        coord1,coord2=request.split('-') # Split request by hyphen separator
        chr,coord1=coord1.split(':') # Get chromosome by splitting by colon
        req=(chr,coord1,coord2)
        res = cur.execute("SELECT rsid FROM gwas WHERE chr_id LIKE ? AND chr_pos
        ↪  BETWEEN ? AND ?",req)
        ret=res.fetchall()
        request=[i[0] for i in ret] # SQL request returns list of singleton
        ↪  tuples, this line converts them to flat list

    elif request_type=='geneName': # Gene symbol (eg IRF4) search
        req=(request,) # Request must be in a tuple
        res = cur.execute("SELECT rsid FROM gwas WHERE mapped_gene LIKE ?",req)
        ret=res.fetchall()
        request=[i[0] for i in ret] # SQL request returns list of singleton
        ↪  tuples, this line converts them to flat list

    else:
        raise Exception("Unsupported type "+str(request_type))
```

The resultant list of SNPs is then iterated over to fetch each corresponding entry from the GWAS, Population, Functional, and Ontology tables. Each entry is then stored in a dictionary to be passed to the main flask code:

```
    ###### Getting dictionary of results ######
    returnDict={}
    for rsid in request:
        innerDict={}
        req=(rsid,) # Request must be in a tuple

        ### Getting gwas data ###
        if manPlot: # If it's a manhattan plot
            res = cur.execute("SELECT i,chr_pos,chr_id,cumulative_pos,logp FROM
            ↪  gwas WHERE rsid LIKE ?",req)
        else:
            res = cur.execute("SELECT
            ↪  rsid,region,chr_pos,chr_id,p_value,mapped_gene FROM gwas WHERE
            ↪  rsid LIKE ?",req)
        ret=res.fetchone()
        assert ret, "error fetching rsid for "+(rsid)

        if not manPlot: # Manhattan plot doesn't need any of the following
            innerDict.update({"gwas":ret})
            ### Getting population data ###
            res=cur.execute("SELECT * FROM population WHERE rsid LIKE ?", req)
            ret=res.fetchone()
            if not ret:
                ret=[_unav for i in range(3)]
            innerDict.update({"pop":list(ret)})
            innerDict['pop']=[round(i,3) for i in innerDict['pop'] if
            ↪  isinstance(i, float)] # remove allele strings, round to 3 dp

            ### Getting functional data ###
            res=cur.execute("SELECT * FROM functional WHERE rsid LIKE ?", req)
            ret=res.fetchone()
            if not ret:
                ret=(rsid,_unav,_unav,_unav)
            innerDict.update({"func":list(ret)})

            ### Getting ontology data ###
            res=cur.execute("SELECT go,term FROM ontology WHERE rsid LIKE ?", req)
            ret=res.fetchall()
            if not ret:
                ret=[(_unav, _unav)]
            innerDict.update({"ont":list(ret)})

        ### Adding results to inner dictionary ###
        if manPlot:
            returnDict.update({rsid:ret})
        else:
            returnDict.update({rsid:innerDict})

    return(returnDict,list(returnDict.keys()))
```

Navigation

Screenshot how site works how to navigate...

Themes

In response to user feedback, a page was added to allow for the selection of themes. These are provided for both aesthetic and accessibility reasons, and as such includes colours found by Rello and Bigham (2017) to help with readability, as well as custom themes created by the project team members, for fun and profit.

User Feedback

10 biologists, ages 25-60, were given the opportunity to use the software to search for either a SNP, region or gene. They were then asked to either retrieve GO data, Linkage disequilibrium plot, or the Manhattan plot.

Items that were improved after user feedback:

- Colours/Themes: users found it easier to read with a darker theme; background was made darker, themes added.
- Font size/style: users found the font easy to read and see, including users with visual impairments.
- Quick links: users did not like scrolling through to find the data they were looking for; adding the quick links makes it quicker to get to the terms.
- Return home button: this feature was added to all pages to allow users to return to the main page.
- Overall users found the genome browser very user friendly and easy to use and enjoyed all the interactive features. They said they liked how easy it is to search and retrieve information, and enlightened them on the vast amount of genome data we now have access to through GWAS.

Overall users found the genome browser very user friendly and easy to use and enjoyed all the interactive features. They said they liked how easy it is to search and retrieve information, and enlightened them on the vast amount of genome data we now have access to through GWAS.

References