Documentation

Introduction

This web application prototype is designed to retrieve information on Single Nucleotide Polymorphisms (SNPs) seen in Type 1 Diabetes Mellitus 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. Functional information and Gene Ontology information are obtained through Ensembl's VEP tool which is all 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. If the user inputs a region containing multiple SNPs, it will return a Manhattan plot of p-values, which is interactive and can be saved as a png file.

Prerequisites

Python 3.10.10	
Flask	2.2.3
Flask-WTF	1.1.1
bokeh	2.4.3
Jinja2	3.1.2
pandas	1.5.3
matplotlib	3.7.0
numpy	1.24.2

Structure

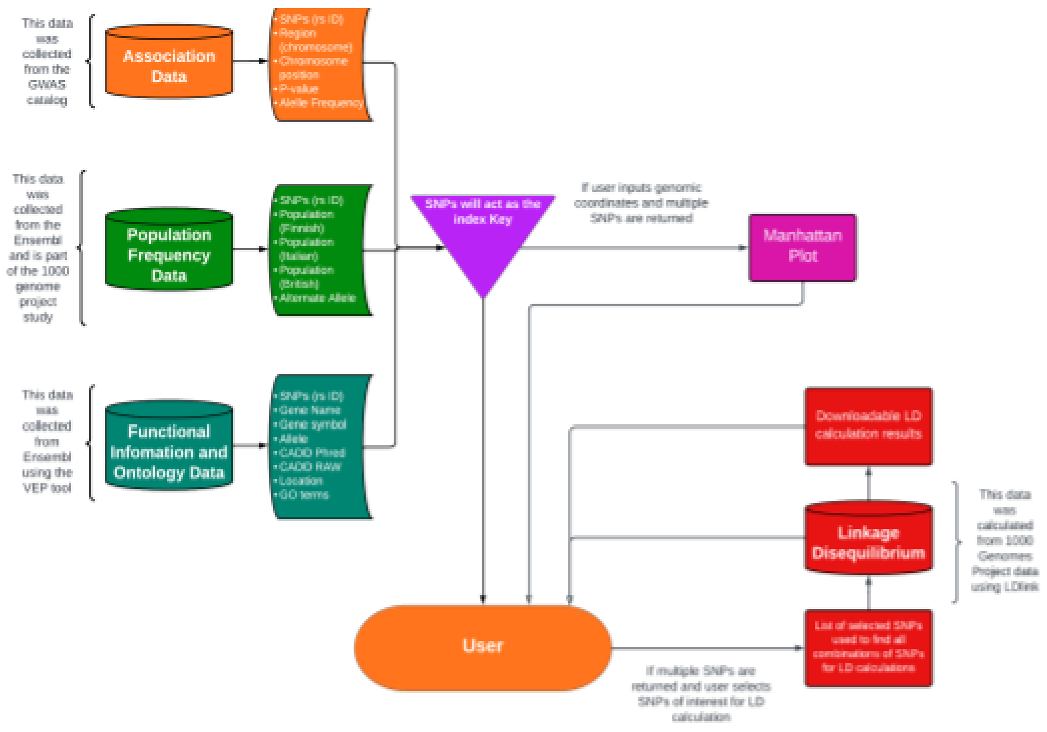

This data was collected from the GWAS catalog
Association Data
• SNPs (rs ID)
• Region (chromosome)
• Chromosome position
• P-value
• Allele Frequency
This data was collected from the Ensembl and is part of the 1000 genome project study
Population Frequency Data
• SNPs (rs ID)
• Population (Finnish)
• Population (Italian)
• Population (British)
• Alternate Allele
This data was collected from Ensembl using the VEP tool
Functional Infomation and Ontology Data
• SNPs (rs ID)
• Gene Name
• Gene symbol
• Allele
• CADD Phred
• CADD RAW
• Location
• GO terms
SNPs will act as the index Key
If user inputs genomic coordinates and multiple SNPs are returned
Manhattan Plot
Downloadable LD calculation results
Linkage Disequilibrium
This data was calculated from 1000 Genomes Project data using LDlink
List of selected SNPs used to find all combinations of SNPs for LD calculations
User
If multiple SNPs are returned and user selects SNPs of interest for LD calculation

Fig. 1: Shows the software schematic structure of what databases were used, which sections were generated using each database, and the plots generated in consequence.

GWAS

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

```
fileIn = getPath('gwas_catalog_v1.0-associations_e108_r2023-01-14.tsv') #
https://www.ebi.ac.uk/gwas/docs/file-downloads
fileOut = getPath('gwas_trimmed.tsv')

data = pd.read_csv(fileIn, sep='\t', low_memory=False)     # Reads gwas tsv
data=removeSpecial(data)    # removes special characters in column names
```

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.

```
data=data.query("disease_trait=='Type 1 diabetes' or
study.str.contains('type 1 diabetes')")
data = data.loc[data.snps.str.contains(r'rs[0-9]+')]        # get only
snps with rsids
#data = data.loc[data['CHR_ID']=='6']                        # Select
only rows for chromosome 6
```

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.

```
data = data[["snps","region","chr_pos","chr_id","p_value","mapped_gene"]]
```

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".

```
data=removeDupeSNP(data)    # Remove duplicates (leaving the entry with
largest p value)
newCol=[removeDupeGeneMap(r["mapped_gene"]) for i, r in data.iterrows()]
data["mapped_gene"]=newCol
```

Duplicate mapped_gene information is then removed.

```
data.rename(columns = {'snps':'rsid'}, inplace = True)

data.to_csv(fileOut, sep='\t', index=False)
```

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 (Junnila, 2015). The Toscani Italian population was chosen due to evidence of high rates of T1D in Northern Italy due to genetic risk (Carla et al. 2009). The British (in England and Wales) population was chosen due to increasing variance of T1D incidence in these regions (Crow, 1991).

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
```

Variant frequency data for chosen populations were retrieved and used to create a TSV file containing the alternate allele and its frequency for all three populations for each SNP in the database. Due to some SNPs in the GWAS dataset not having an rsID, allele frequency data was excluded for SNPs at positions chr2:136066296, chr6:27349237, chr6:28594470, chr6:32978829 and chr19:42174477.

Functional information and Gene Ontology

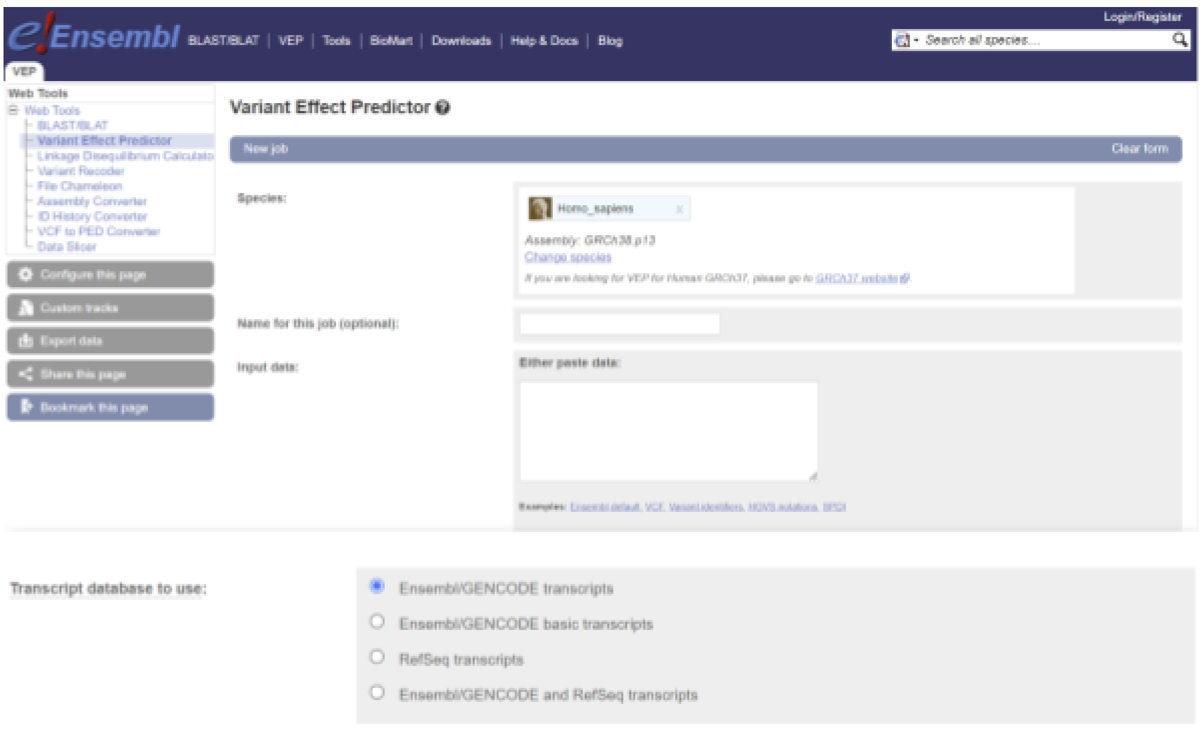
There are several different measures of functional impact such as CADD (Combined Annotation Dependent Depletion). CADD is a tool used for predicting the potential harm caused by genetic variants known as its deleteriousness. A CADD score indicates the likelihood of a variant being deleterious.(CADD - Combined Annotation Dependent Depletion, 2021).

One advantage of CADD, over other measures of functional impact such as SIFT or PolyPhen, is that it 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.(Rentzsch, P., et al 2021)

Gene Ontology (GO) is a standardised system that provides a framework for describing genes, gene products in terms of their biological processes, molecular functions, and cellular components. It is an essential tool in functional genomics research, where it is used to annotate gene function and analyse functional relationships between genes by mapping SNPs to genes, and their associated GO terms to infer the potential functional consequences of the SNP (The Gene Ontology Consortium, 2021 and Ashburner et al., 2000)

Collecting data

The Ensembl's Variant Effect Predictor (VEP) web tool was used to gather the Functional and Ontology data by submitting a job with the rsIDs from the T1D_GWAS_add.tsv file separated by commas.



Variant Effect Predictor

Species: Homo_sapiens

Assembly: GRCh38.p13

If you are looking for VEP for Human GRCh37, please go to GRCh37 website

Name for this job (optional):

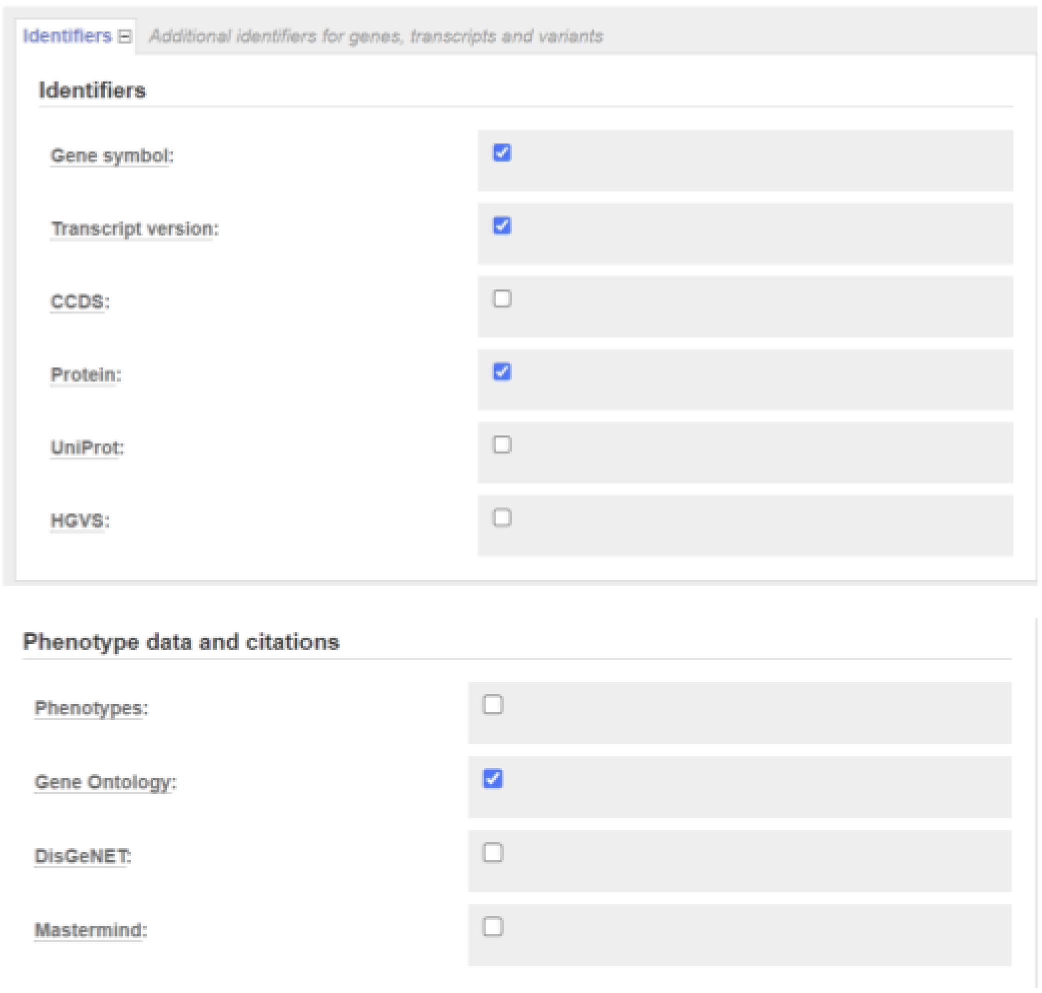
Input data: Either paste data:

Transcript database to use:

- [✓ Ensembl/GENCODE transcripts]
- Ensembl/GENCODE basic transcripts
- RefSeq transcripts
- Ensembl/GENCODE and RefSeq transcripts

Fig. 2: This is the Variant Effect Predictor (VEP) job submission page

From the Additional identifiers tab the following options were selected: Gene Symbol, Protein, and Gene Ontology. This allowed for us to associate the ontology terms with Genes and Protein names.



Identifiers

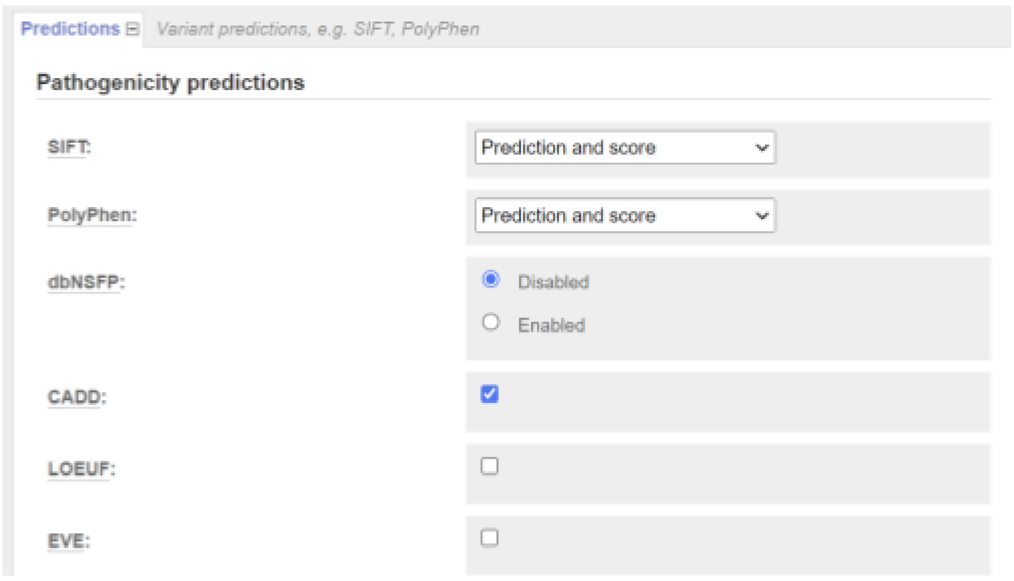
Gene symbol:	✓
Transcript version:	✓
CCDS:	☐
Protein:	✓
UniProt:	☐
HGVS:	☐

Phenotype data and citations

Phenotypes:	☐
Gene Ontology:	✓
DisGeNET:	☐
Mastermind:	☐

Fig. 3: Shows the identifiers and phenotype data tabs from the job submission page on VEP

CADD was selected in the Prediction column to add a column containing the Raw CADD score and the CADD Phred score which then both are associated with the rsID and the Genes and Protein names.


Predictions ⊟ *Variant predictions, e.g. SIFT, PolyPhen*

Pathogenicity predictions

SIFT:	Prediction and score
PolyPhen:	Prediction and score
dbNSFP:	[✓ Disabled] Enabled
CADD:	✓
LOEUF:	☐
EVE:	☐

Fig. 4: This is the predictions tab of the job submission page

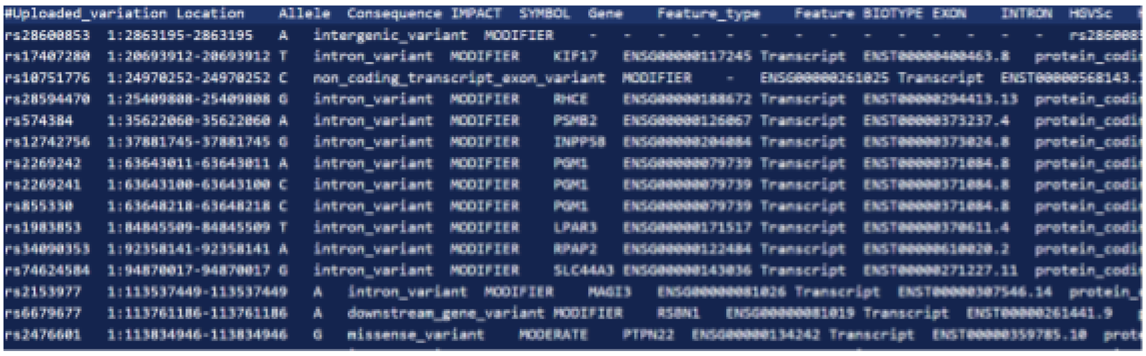

```
#Uploaded_variation Location     Allele  Consequence IMPACT  SYMBOL  Gene    Feature_type    Feature BIOTYPE EXON    INTRON  HGVSc
rs28600853  1:2863195-2863195   A   intergenic_variant  MODIFIER    -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   rs286008
rs17407280  1:20693912-20693912 T   intron_variant  MODIFIER    KIF17   ENSG00000117245 Transcript  ENST00000400463.8   protein_codi
rs10751776  1:24970252-24970252 C   non_coding_transcript_exon_variant  MODIFIER    -   ENSG00000261025 Transcript  ENST00000568143.
rs28594470  1:25409808-25409808 G   intron_variant  MODIFIER    RHCE    ENSG00000188672 Transcript  ENST00000294413.13  protein_codi
rs574384    1:35622060-35622060 A   intron_variant  MODIFIER    PSMB2   ENSG00000126067 Transcript  ENST00000373237.4   protein_codi
rs12742756  1:37881745-37881745 G   intron_variant  MODIFIER    INPP5B  ENSG00000204084 Transcript  ENST00000373024.8   protein_codi
rs2269242   1:63643011-63643011 A   intron_variant  MODIFIER    PGM1    ENSG00000079739 Transcript  ENST00000371084.8   protein_codi
rs2269241   1:63643100-63643100 C   intron_variant  MODIFIER    PGM1    ENSG00000079739 Transcript  ENST00000371084.8   protein_codi
rs855330    1:63648218-63648218 C   intron_variant  MODIFIER    PGM1    ENSG00000079739 Transcript  ENST00000371084.8   protein_codi
rs1983853   1:84845509-84845509 T   intron_variant  MODIFIER    LPAR3   ENSG00000171517 Transcript  ENST00000370611.4   protein_codi
rs34090353  1:92358141-92358141 A   intron_variant  MODIFIER    RPAP2   ENSG00000122484 Transcript  ENST00000610020.2   protein_codi
rs74624584  1:94870017-94870017 G   intron_variant  MODIFIER    SLC44A3 ENSG00000143036 Transcript  ENST00000271227.11  protein_codi
rs2153977   1:113537449-113537449   A   intron_variant  MODIFIER    MAGI3   ENSG00000081026 Transcript  ENST00000307546.14  protein_
rs6679677   1:113761186-113761186   A   downstream_gene_variant MODIFIER    RSBN1   ENSG00000081019 Transcript  ENST00000261441.9
rs2476601   1:113834946-113834946   G   missense_variant    MODERATE    PTPN22  ENSG00000134242 Transcript  ENST00000359785.10  prot
```

Fig. 5: The TSV file containing all the CADD scores and GO Terms

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

Trimming data

After running the job, we get an output text file with several columns however we are only interested in the following:
Uploaded_Variation: the reference SNP identifier for the variant.
Allele: the alternative allele observed at the variant site.
location: the location of the variant within the affected gene
Gene: the Ensembl gene ID of the affected gene.
Symbol: the gene symbol or name.
CADD_PHRED: Phred-scaled CADD score (Combined Annotation-Dependent Depletion), which predicts the deleteriousness of variants.
CADD_RAW: the raw CADD score, which is a measure of the deleteriousness of variants.
GO Terms: Gene Ontology (GO) terms associated with the affected gene.

The Functional information was extracted using the following script utilising pandas dataframe function to select the required columns and then converted back into a new csv file using pandas **to_csv()** function.

```
import pandas as pd
Association_table_filename = 'Functional_and_Ontology_data.tsv'
df = pd.read_csv(Association_table_filename, sep='\t')
columns_to_keep = ['Uploaded_variation','Allele','CADD_PHRED',
'CADD_RAW', 'PolyPhen', 'SIFT']
df = df[columns_to_keep]
print(df)

df.to_csv("Func_trimmed.csv", index=False)
```

The Gene ontology data was filtered in a similar way using the following script.

```
import pandas as pd
Association_table_filename = 'Functional_and_Ontology_data.tsv'
df = pd.read_csv(Association_table_filename, sep='\t')
columns_to_keep = ['Uploaded_variation','Location','SYMBOL', 'Gene', 'GO']
df = df[columns_to_keep]
print(df)

df.to_csv("GO_trimmed.csv", index=False)
```

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.

Linkage Disequilibrium

Linkage disequilibrium (LD) is the degree of non-random association of alleles at different loci, within a population. LD is typically measured by two metrics: D’ and r^2 (Slatkin, M. et al., 2008).

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

$$D' = \frac{D}{D_{max}} \quad where \quad D_{max} = \begin{cases} max\{-p_A p_B, -(1-p_A)(1-p_B\} & when\ D < 0 \\ min\{p_A(1-p_B), (1-p_A)p_B\} & when\ D > 0 \end{cases}$$

r^2 is the correlation coefficient between two loci:

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

Both D’ and r^2 metrics take a non-zero value up to 1. A value of 1 indicates that alleles of a pair of SNPs are completely non-randomly associated whereas a value close to 0 indicates a near-independent association between alleles (Machiela, M.J. et al., 2015).

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 (Italy) and British populations are loaded in with pandas as separate dataframes. Each data frame has their index set to the first column which contains SNP rsIDs.

```
import pandas as pd
# Finland (FIN)
LD_D_FIN = pd.read_table('FIN_D.txt')
LD_D_FIN = LD_D_FIN.set_index('RS_number')
LD_r2_FIN = pd.read_table('FIN_r2.txt')
LD_r2_FIN = LD_r2_FIN.set_index('RS_number')
# Italy - Tuscany (TSI)
LD_D_TSI = pd.read_table('TSI_D.txt')
LD_D_TSI = LD_D_TSI.set_index('RS_number')
LD_r2_TSI = pd.read_table('TSI_r2.txt')
LD_r2_TSI = LD_r2_TSI.set_index('RS_number')
```

```
# British (GBR)
LD_D_GBR = pd.read_table('GBR_D.txt')
LD_D_GBR = LD_D_GBR.set_index('RS_number')
LD_r2_GBR = pd.read_table('GBR_r2.txt')
LD_r2_GBR = LD_r2_GBR.set_index('RS_number')
```

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.

```
# Take list of SNPs and creates a pair of lists containing the 1st and 2nd SNP of each
combination
def SNP_pair_lists(SNP_list):
    SNP_combinations = list(itertools.combinations(SNP_list,2))
    SNP_1_list = []
    SNP_2_list = []
    for SNP_pair in SNP_combinations:
        SNP_1_list.append(SNP_pair[0])
        SNP_2_list.append(SNP_pair[1])
    return SNP_1_list, SNP_2_list

SNP_1_list, SNP_2_list = SNP_pair_lists(SNP_list)
```

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 data frame 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.

```
# Create Empty dataframe
LD_dataset = pd.DataFrame(columns=['SNP_1', 'SNP_2', 'FIN_D\'', 'FIN_r2', 'TSI_D\'',
'TSI_r2', 'GBR_D\'', 'GBR_r2'])
# Indexes the respective LD calculation for each pair and adds it to the data
for SNP_1,SNP_2 in zip(SNP_1_list,SNP_2_list):
    # Finland
    FIN_D = LD_D_FIN[SNP_1].loc[SNP_2]
    FIN_r2 = LD_r2_FIN[SNP_1].loc[SNP_2]
    # Italy - Tuscany
    TSI_D = LD_D_TSI[SNP_1].loc[SNP_2]
    TSI_r2 = LD_r2_TSI[SNP_1].loc[SNP_2]
    # British
    GBR_D = LD_D_GBR[SNP_1].loc[SNP_2]
    GBR_r2 = LD_r2_GBR[SNP_1].loc[SNP_2]
    # Create row of data and combine with LD dataset dataframe
    row_list = [SNP_1, SNP_2, FIN_D, FIN_r2, TSI_D, TSI_r2, GBR_D, GBR_r2]
    row = pd.DataFrame(row_list).T

    row.columns = LD_dataset.columns
    LD_dataset = pd.concat([LD_dataset, row])
# Write out LD dataset as a TSV
LD_dataset.to_csv('LD_T1DM_Chr6.tsv', sep="\t", index=False)
```

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.

```python
def remove_invalid_SNPs(SNP_list, LD_dataset_file = "data/TSVs/LD_T1DM_Chr6.tsv"):
# remove SNPs returned from query which have no LD values in LD dataset
    # Load LD dataset as pandas dataframe
    LD_df = pd.read_table(LD_dataset_file)
    # checks for SNPs in subset which are not in LD dataset
    invalid_list = []
    for SNP in SNP_list:
        if SNP not in LD_df['SNP_1'].tolist(): # check if SNP is in LD dataset
            invalid_list.append(SNP) # add to list of invalid SNPs
    print(invalid_list)
    # remove invalid SNPs from SNP list passed to LD plot
    for SNP in invalid_list:
        SNP_list.remove(SNP)
    return SNP_list

SNP_list = remove_invalid_SNPs(SNP_list)
```

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

```python
# Load LD dataset and create empty dataframe for filtered results
LD_df = pd.read_table(LD_dataset_file)
LD_results_df = pd.DataFrame(columns=['SNP_1', 'SNP_2', 'FIN_D\'', 'FIN_r2', 'TSI_D\'',
'TSI_r2', 'GBR_D\'', 'GBR_r2'])
# create a pair of lists containing the 1st and 2nd SNP of each combination
SNP_1_list, SNP_2_list = SNP_pair_lists(SNP_list)
# Loop indexing LD dataset using each pair of SNPs
for SNP_1,SNP_2 in zip(SNP_1_list,SNP_2_list):
    LD_row = LD_df.loc[((LD_df['SNP_1'] == SNP_1) & (LD_df['SNP_2'] == SNP_2) |
                        (LD_df['SNP_1'] == SNP_2) & (LD_df['SNP_2'] == SNP_1))]
    LD_results_df = pd.concat([LD_results_df, LD_row])
```

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.

```
# load LD dataset
LD_df = pd.read_table('LD_T1DM_Chr6.tsv')
# checks for SNPs in subset which are not in LD dataset
SNP_list = remove_invalid_SNPs(SNP_list)
# create a pair of lists containing the 1st and 2nd SNP of each combination
SNP_1_list, SNP_2_list = SNP_pair_lists(SNP_list)
```

An empty dataframe is created to be filled with LD values used to create a matrix of values used for the LD plot. The pair of SNP lists are 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.

```
LD_matrix_df = pd.DataFrame(columns=[SNP_list]) # One column per SNP in list (Since a
list object is passed, could just pass the SNP list
for SNP_1 in SNP_list:
    # Create empty list
    LD_value_list = []
    # Sub-loop - Loops to create list of datapoints
    for SNP_2 in SNP_list:
        if SNP_1 == SNP_2:
            SNP_Datapoint = 1
            LD_value_list.append(SNP_Datapoint)
        else:
            #try:
            # Search for specific row containing value
            LD_row = LD_df.loc[((LD_df['SNP_1'] == SNP_1) & (LD_df['SNP_2'] == SNP_2) |
                                (LD_df['SNP_1'] == SNP_2) & (LD_df['SNP_2'] == SNP_1))]
            # Extract value and add to list
            SNP_Datapoint = LD_row['GBR_r2'].tolist()[0] # currently using Finnish data
            LD_value_list.append(SNP_Datapoint)
            #except:
                #invalid_list.append((SNP_main,SNP_second))
    # Convert into dataframe row and transpose
    row = pd.DataFrame(LD_value_list).T
    row.columns = LD_matrix_df.columns
    LD_matrix_df = pd.concat([LD_matrix_df, row])
```

The LD matrix data frame is passed to the LD_plot function. This function was based on code from the ld_plot package publicly available via PyPI. The number of matrix rows 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. A colour bar is added to the figure based on the colour map for the heatmap

```python
def LD_plot(LD, labels, title):

    n = LD.shape[0]

    figure = plt.figure()

    # mask triangle matrix
    mask = np.tri(n, k=0)
    LD_masked = np.ma.array(LD, mask=mask)

    # create rotation/scaling matrix
    t = np.array([[1, 0.5], [-1, 0.5]])
    # create coordinate matrix and transform it
    coordinate_matrix = np.dot(np.array([(i[1], i[0])
                                         for i in itertools.product(range(n, -1, -1),
range(0, n + 1, 1))]), t)
    # plot
    ax = figure.add_subplot(1, 1, 1)
    ax.spines['bottom'].set_position('center')
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.spines['left'].set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.tick_params(axis='x', which='both', top=False)
    plt.pcolor(coordinate_matrix[:, 1].reshape(n + 1, n + 1),
                   coordinate_matrix[:, 0].reshape(n + 1, n + 1), np.flipud(LD_masked),
edgecolors = "white", linewidth = 1.5, cmap = 'OrRd')
    plt.xticks(ticks=np.arange(len(labels)) + 0.5, labels=labels, rotation='vertical',
fontsize=8)
    plt.colorbar()

    # add title
    plt.title(f"{title}", loc = "center")

    return figure

LD_heatmap_plot = ld_plot(LD_matrix_df, SNP_list, "LD plot title")
```

The linkage disequilibrium plot is a heatmap visualising data from a matrix of LD values. Each cell of the heatmap represents a pairwise LD value of two SNPs with the cell colour corresponding with the LD value based on a colour map scale shown by the colour bar on the right of the plot. Below the heatmap plot is a list of labels.

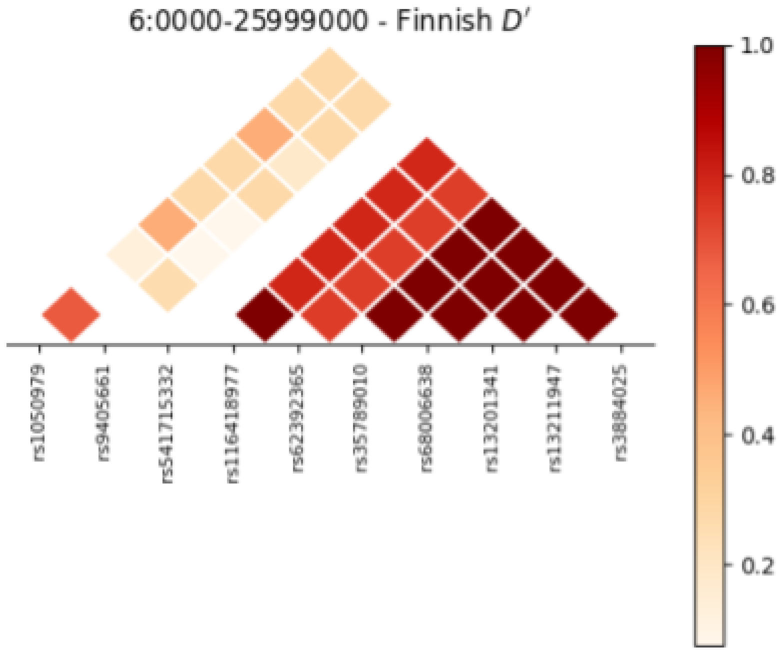


Fig 6. Linkage Disequilibrium heatmap plot of SNPs within chromosomal coordinates range chr6:0-25999000. Plot shows D' values calculated using allele frequency data from Finnish 1000 Genomes Project. Heatmap cell colour values are indicated by the colour bar right of heatmap. SNPs included in the plot are indicated by the x-axis of the plot.

Manhattan Plot

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.

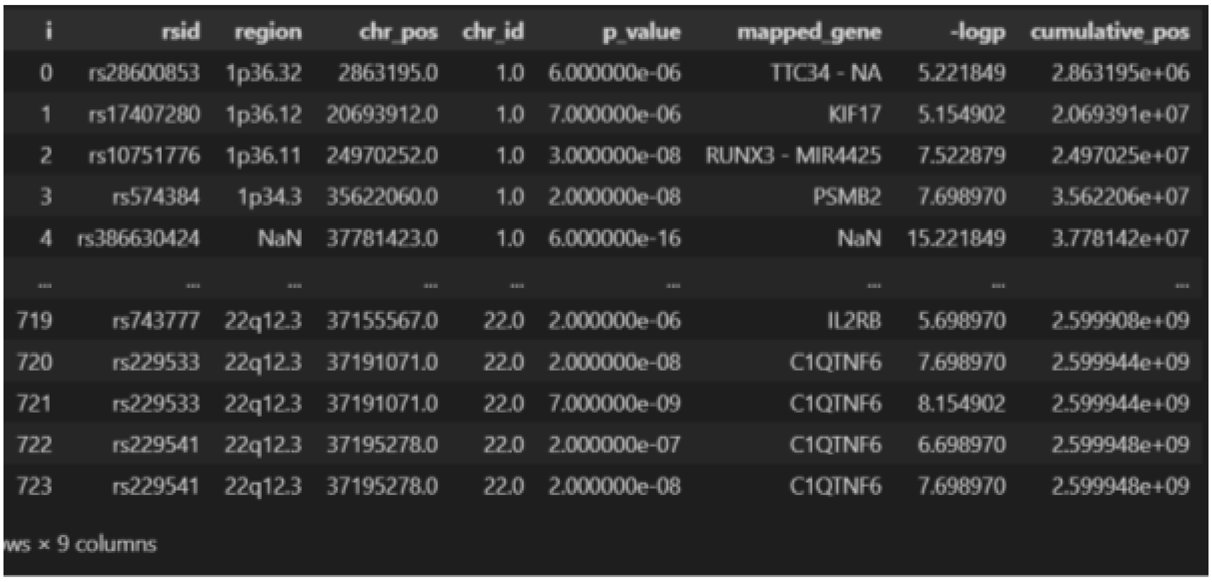
A TSV file containing all SNPs in type 1 diabetes was produced and edited using the pandas library in Python. The Manhattan plot needs the P-value of a SNP and the chromosomal position. Using pandas, the -logp and cumulative positions were generated.

```
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')
```

Above code shows pandas being used to create an additional column for -logp values and a column for cumulative positions to be added to GWAS_T1D.tsv file by using the p-value, 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" )
```
The Manhattan plots are generated using the Bokeh python library. Cumulative positions (x) are plotted against -logp (y). Each chromosome is separated by alternating grey and black sections.



i	rsid	region	chr_pos	chr_id	p_value	mapped_gene	-logp	cumulative_pos
0	rs28600853	1p36.32	2863195.0	1.0	6.000000e-06	TTC34 - NA	5.221849	2.863195e+06
1	rs17407280	1p36.12	20693912.0	1.0	7.000000e-06	KIF17	5.154902	2.069391e+07
2	rs10751776	1p36.11	24970252.0	1.0	3.000000e-08	RUNX3 - MIR4425	7.522879	2.497025e+07
3	rs574384	1p34.3	35622060.0	1.0	2.000000e-08	PSMB2	7.698970	3.562206e+07
4	rs386630424	NaN	37781423.0	1.0	6.000000e-16	NaN	15.221849	3.778142e+07
...	...	...	...	...	...	...	...	...
719	rs743777	22q12.3	37155567.0	22.0	2.000000e-06	IL2RB	5.698970	2.599908e+09
720	rs229533	22q12.3	37191071.0	22.0	2.000000e-08	C1QTNF6	7.698970	2.599944e+09
721	rs229533	22q12.3	37191071.0	22.0	7.000000e-09	C1QTNF6	8.154902	2.599944e+09
722	rs229541	22q12.3	37195278.0	22.0	2.000000e-07	C1QTNF6	6.698970	2.599948e+09
723	rs229541	22q12.3	37195278.0	22.0	2.000000e-08	C1QTNF6	7.698970	2.599948e+09

ws × 9 columns

Table 1: shows all the SNPs used from the database, with the -logp and cumulative positions. Total of 724 SNPs were used for the Manhattan plot.

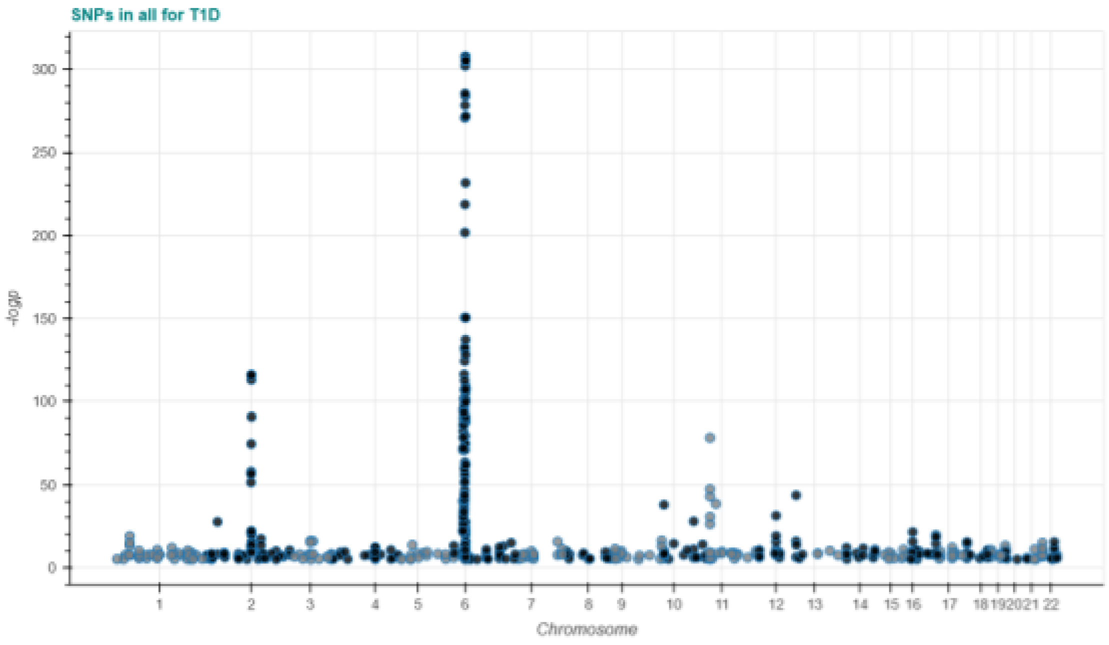


Fig. 7: Shows all GWAS SNPs found in Type 1 Diabetes shown in a Manhattan Plot. Users can view all SNPs for T1D, and plot the SNPs that they have searched for. This feature is available when multiple SNPs are retrieved for the region searched.

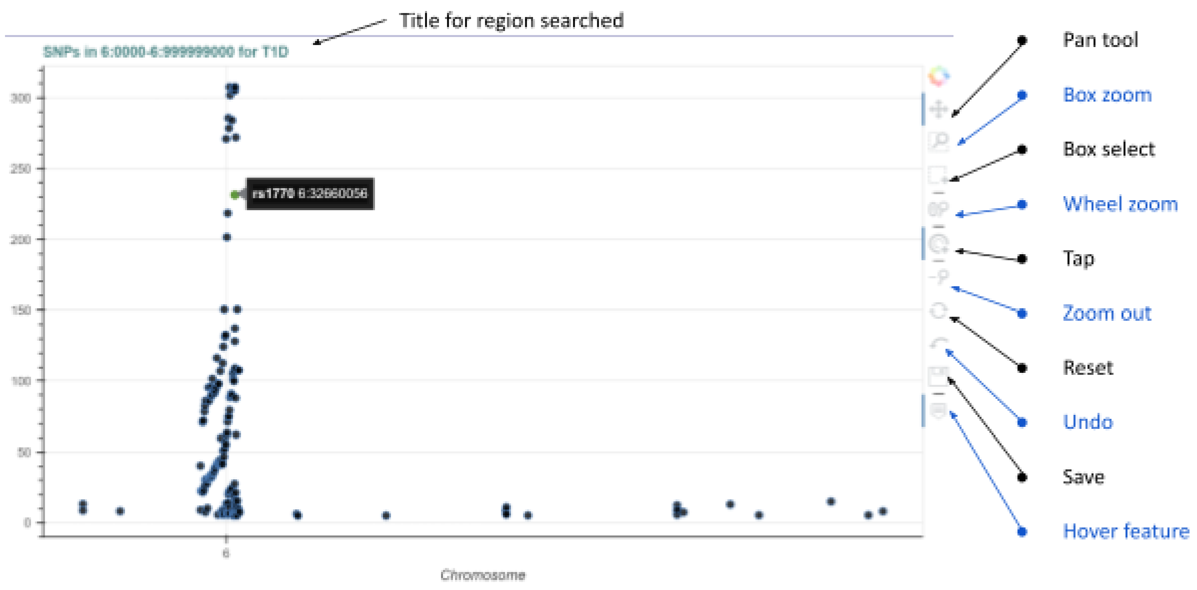


Fig. 8: Shows all GWAS SNPs for Type 1 Diabetes for only Chromosome 6.

Bokeh tools were used to add additional interactive functionality and make the graph user friendly. Circle plots were chosen for easy visibility and selecting of each SNP position. Tools and functionality added to the Manhattan plot include:

- Hover tool was added to allow the user to hover mouse over each plot, which turns it green, and shows the rs ID value next to the chromosomal position.
- Select tool allows users to select 1 or more plots by either clicking one plot, or holding shift and clicking plots for multiple plots, or
- Box select tool to select plots in an entire section, and turns the plots purple.
- Zoom tool allows the user to zoom into the graph and view plots that are close to each other or appear to overlap for more visual clarity.
- Reset and undo buttons allows the user to either go back to the original plot, or undo the previous action they had committed.
- Save tool saves the graph as a png file with the title included. The plots were made slightly transparent to make overlapping SNPs more visible.

Functions used throughout

A number of functions were used throughout the code such as **removeDupeSNP():** (seen below) this function removes duplicate SNPs from a pandas dataframe leaving only the SNP with the greatest P-value. This was done to provide a less cluttered output and overall more aesthetically pleasing table that is more "human friendly", with the exception of the Manhattan plot which uses the different P-values of the duplicate SNPs.
The code first identifies the duplicate rows, and creates a dictionary to store the index and p-value of each duplicate row.

```
def removeDupeSNP(dataframe):
# Removes duplicates from a pandas dataframe, leaving only greatest
```

```
p-value
   dataframe.reset_index(drop=True)
# Resets index back to 0
   dupeList = dataframe.duplicated(subset='snps',keep=False)
# Get list of duplicate values
   dupes=dataframe[dupeList]
# Select dataframe using above list
   dupesDict={}
   for index,row in dupes.iterrows():
# Iterate through df of duplicates, one row at a time
       rsVal=row["snps"]
# SNP name (rs value)
       snpTuple=(index,row["p_value"])
# Tuple containing index and p-val
       if rsVal in dupesDict:
# If it's seen the snp before,
           dictList=dupesDict[rsVal]
# go to the value for the snp,
           dictList.append(snpTuple)
# and add the index/ p-val tuple.
       else:
# If it hasn't seen the snp before,
           dupesDict.update({rsVal:[snpTuple]})
# create a listing for it.
   naughtyList=[]                                   # List of lists of
(index, p-val) we want to drop
```

The duplicate rows are then sorted by p-value, keeping only the row with the greatest p-value, and creating a list of the indices of the rows to be dropped.

```
   for i in dupesDict:
       snp = dupesDict[i]                           # Get list of (index,
pVal)
       sortByP=sorted(snp,key=lambda x: 0-x[1])     # Sort by p-value
       sortByP=sortByP[1:]                          # Select all but greatest
p value
       naughtyList.append(sortByP)


   dropList=[]                          # List of indices for rows we want to
drop
   for i in naughtyList:                # Enter first list
       for j in i:                      # Enter second list
           dropList.append(j[0])        # Add the index from each tuple
```

Finally, the function returns the input DataFrame with the identified duplicate rows removed.

```
    return(dataframe.drop(dropList))    # Return dataframe without duplicate
values
```

Another function made and used throughout was **removeDupeGeneMap()**, which removes duplicate gene symbols, or if no symbols are found it outputs “data unavailable” instead.

```
def removeDupeGeneMap(GeneMap):
    try:
        GeneMap=GeneMap.split(', ')
        uniques=""
        for item in GeneMap:
            if item not in uniques: # If the item hasn't been seen before,
                uniques+=(item)     # add it to the list.
                uniques+=(", ")     # Also add ' ,'
        return (uniques[:-2])       # Remove last ' ,'
    except:
        return ("Data unavailable")     # Return this if geneMap is empty
```

User Feedback

A short study was conducted with 10 molecular, and microbiologists, aged 25-60 years old. They were given the opportunity to use the software to search through our database. Each participant was given tasks to complete, and then were asked to give feedback based on their experience using the site. Each individual was either asked to search a SNP, genomic region, or gene of interest, and then had to retrieve GO data, Linkage disequilibrium plots, or a Manhattan plot.

Things that were improved based on 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 genomic data we now have access to through GWAS.

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 *
DB=DBpath()

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

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

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

```
import pandas as pd
import os
import sqlite3

_unav = "Data unavailable"
_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 = tsv_path
    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'} # Set 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 ext 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 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:

```
    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

            g_rsid,g_region,g_chr_pos,g_chr_id,g_p_value,g_mapped_gene=ret

gwasRet=(g_rsid,g_region,f'{g_chr_id}:{g_chr_pos}',g_p_value,removeDupeGeneMap
(g_mapped_gene))
            gwasRet=(_unav if i == None else i for i in gwasRet) # Turn empty
values into 'data unavailable'
            innerDict.update({"gwas":gwasRet})
            ### 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()))  # snp_list is keys
```

Themes

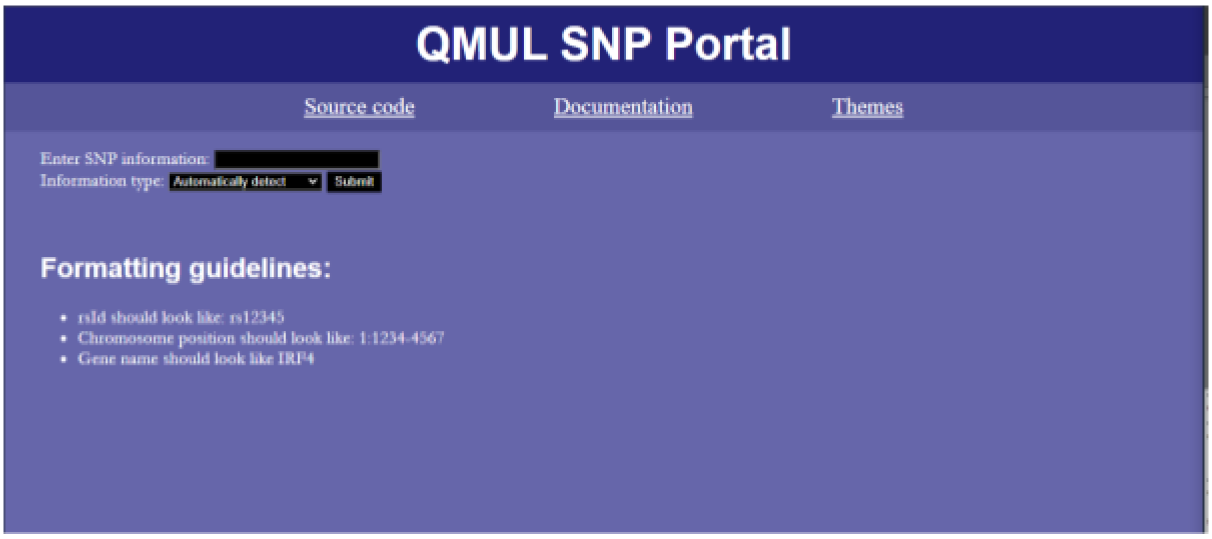
In response to user feedback, a page was added to allow for the selection of themes. These are provided for both aesthetic and accessibility reasons, and as such includes colours found by Rello and Bigham (2017) to help with readability, as well as custom themes based on each of the team members' individual aesthetic preferences. This functionality was provided through a **setTheme()** javascript function that, when called, sets the CSS variables to custom values, then stores those values in session storage. Said storage is then accessed through **getTheme()** on page load.

```
// Get the root element
var r = document.querySelector(':root');
function setTheme(textColour="white", contrast_bg="black",
            mild_bg= "#66a", med_bg= "#559", strong_bg= "#227") {

    // Set value of colour variables
    r.style.setProperty('--textColour', textColour);
    r.style.setProperty('--contrast_bg', contrast_bg);
    r.style.setProperty('--mild_bg', mild_bg);
    r.style.setProperty('--med_bg', med_bg);
    r.style.setProperty('--strong_bg', strong_bg);

    // Saving selection to session storage
    sessionStorage.setItem("textColour", textColour);
    sessionStorage.setItem("contrast_bg", contrast_bg);
    sessionStorage.setItem("mild_bg", mild_bg);
    sessionStorage.setItem("med_bg", med_bg);
    sessionStorage.setItem("strong_bg", strong_bg);
}

function getTheme(){

    // retrieve colours from session storage
    let textColour = sessionStorage.getItem("textColour");
    let contrast_bg = sessionStorage.getItem("contrast_bg");
    let mild_bg = sessionStorage.getItem("mild_bg");
    let med_bg = sessionStorage.getItem("med_bg");
    let strong_bg = sessionStorage.getItem("strong_bg");
    // set theme
    setTheme(textColour, contrast_bg, mild_bg, med_bg, strong_bg)
}

// if there's anything in session storage, retrieve the theme
if (sessionStorage.getItem("textColour")){
    getTheme
}

// otherwise set it to default
else{
    setTheme()
}
```

Navigation

Home

The home page of the site contains links to the software's source code, documentation and a web page which can be used to change the website's theme. **Figures 10** and **11** show the "Source code" and "Documentation" links which open up pages to the main Github repository and a pdf version of the documentation respectively.



QMUL SNP Portal

Source code Documentation Themes

Enter SNP information:

Information type: Automatically detect Submit

Formatting guidelines:

- rsId should look like: rs12345
- Chromosome position should look like: 1:1234-4567
- Gene name should look like IRF4

Fig. 9: This shows the main home page of the software

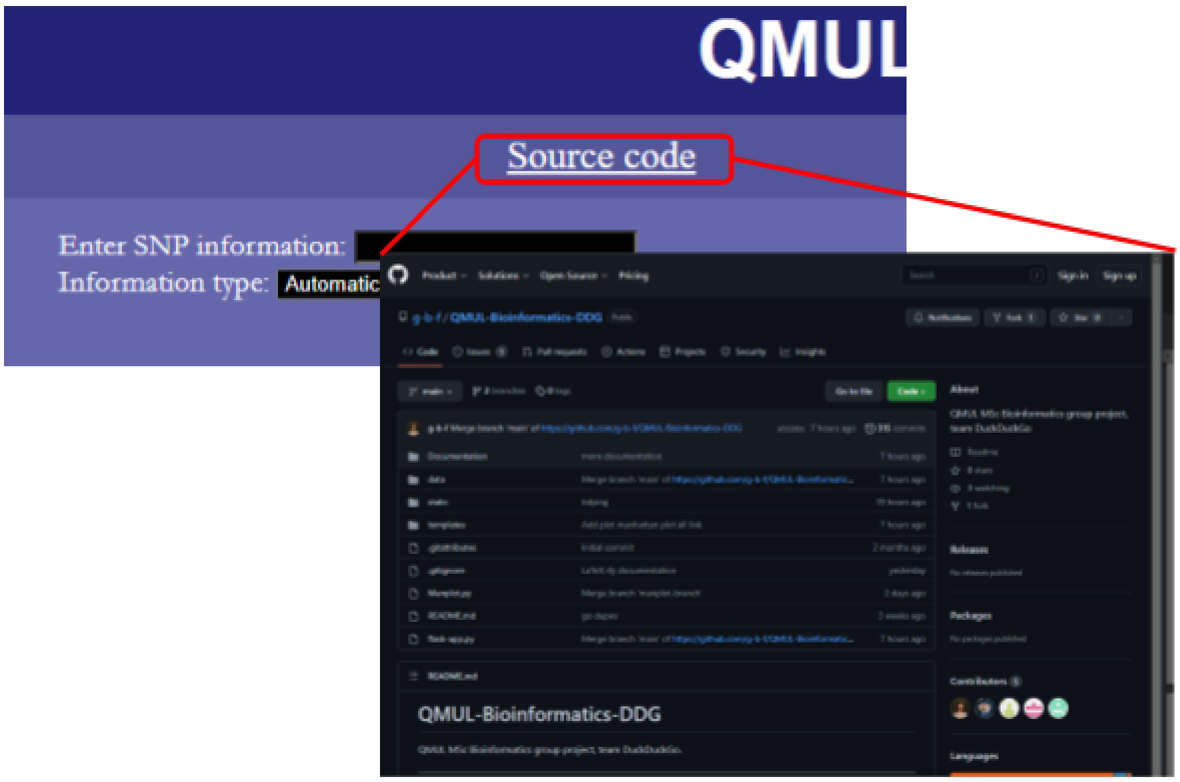


Fig. 10: Shows the 'Source Code' link, and the screenshot of the Github page it opens up

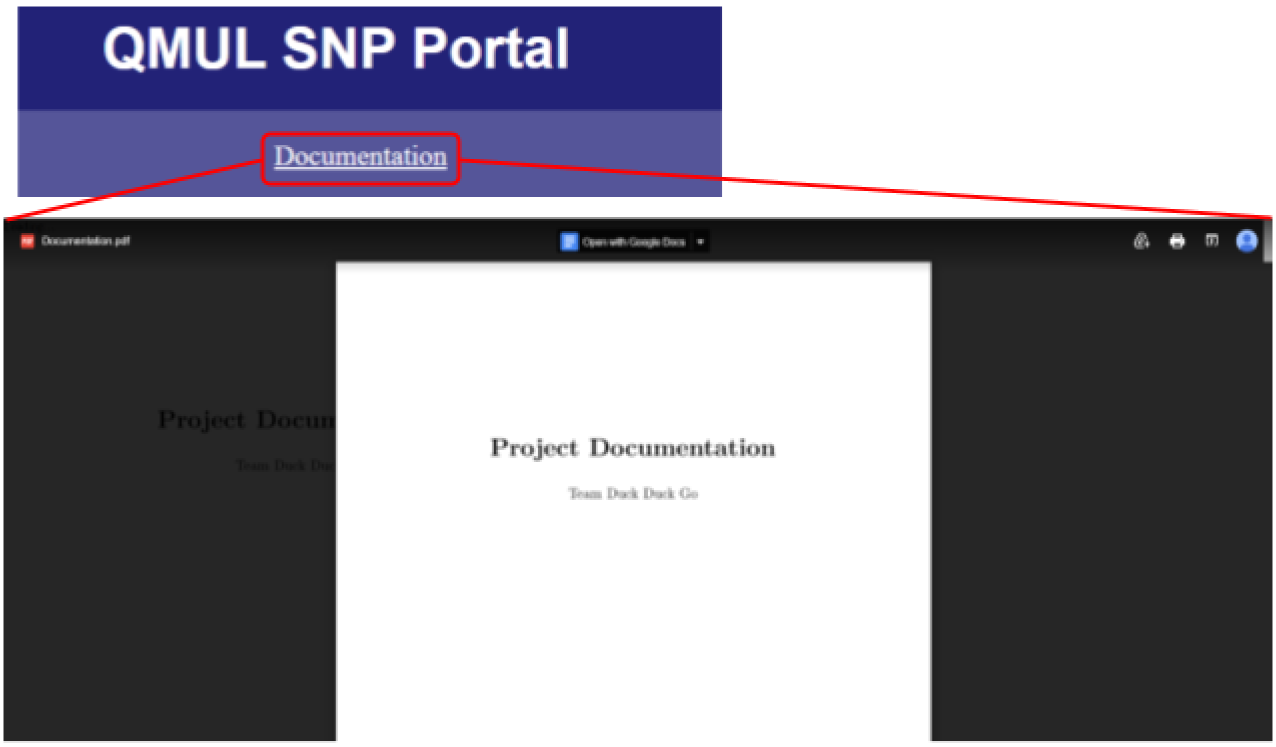


Fig. 11: Shows the 'Documentation' link, and the PDF image of the documentation that opens up

The themes link takes the user to the page seen below where they can select a theme based on the user's preference such as the light and dark filters. The page also contains three filters highlighted by the white box that are designed to help with the readability for users with dyslexia using single soft pastel coloured backgrounds with high contrasting fonts as recommended by the British Dyslexia Association (British Dyslexia Association, 2018).

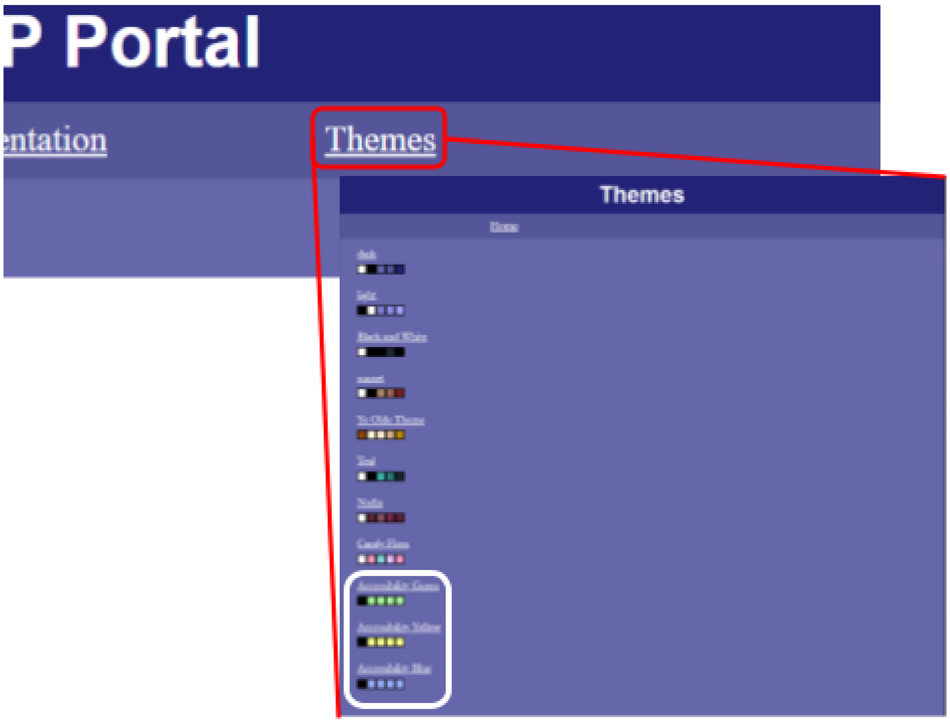
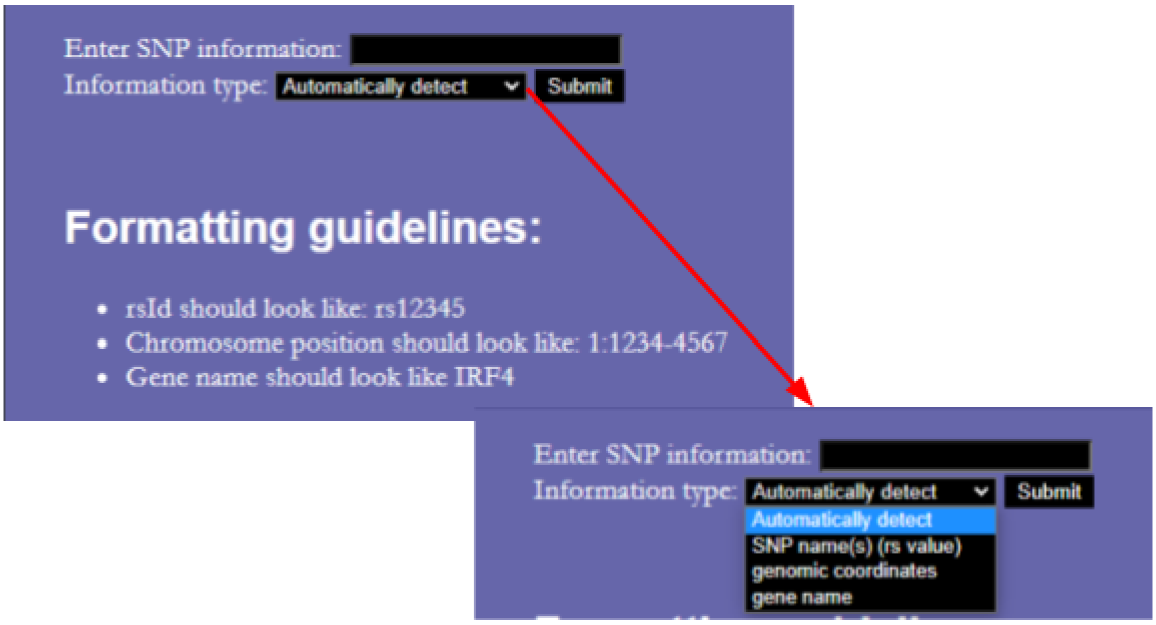


Fig. 12: This image shows all the different themes available, including themes specific for the visually impaired

The user can search for SNP information through a variety of criteria demonstrated in the formatting guidelines. By default the type of information is set to automatically detect the users input however this can be set to one specific type through using the dropdown.



Enter SNP information:
Information type: Automatically detect   Submit

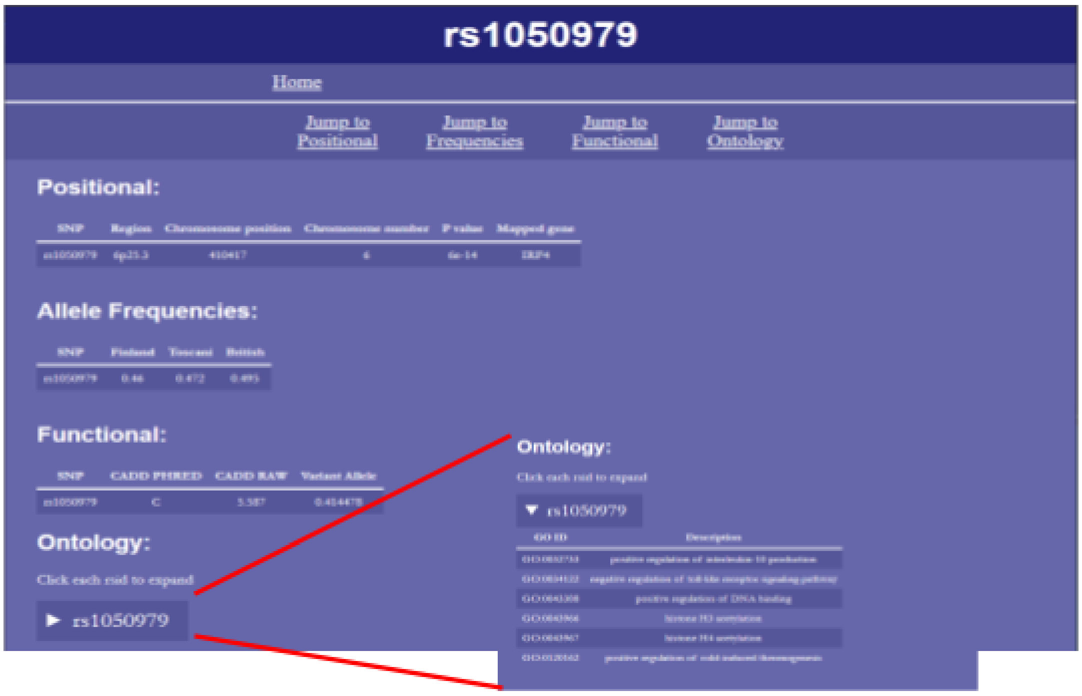
Formatting guidelines:

- rsId should look like: rs12345
- Chromosome position should look like: 1:1234-4567
- Gene name should look like IRF4

Fig. 13: Shows the drop down menu that users can use to select the type of data they want to search

Results

When single rsIDs are entered the following page is returned, the user can jump to each section using the four quick links making the page more efficient to navigate when there is only one section of interest.



rs1050979

Home

Jump to Positional | Jump to Frequencies | Jump to Functional | Jump to Ontology

Positional:

SNP	Region	Chromosome position	Chromosome number	P value	Mapped gene
rs1050979	6p25.3	410417	6	6e-14	IRF4

Allele Frequencies:

SNP	Finland	Toscani	British
rs1050979	0.46	0.472	0.495

Functional:

SNP	CADD PHRED	CADD RAW	Variant Allele
rs1050979	C	3.587	0.414478

Ontology:

Click each rsid to expand

▶ rs1050979

Ontology:

Click each rsid to expand

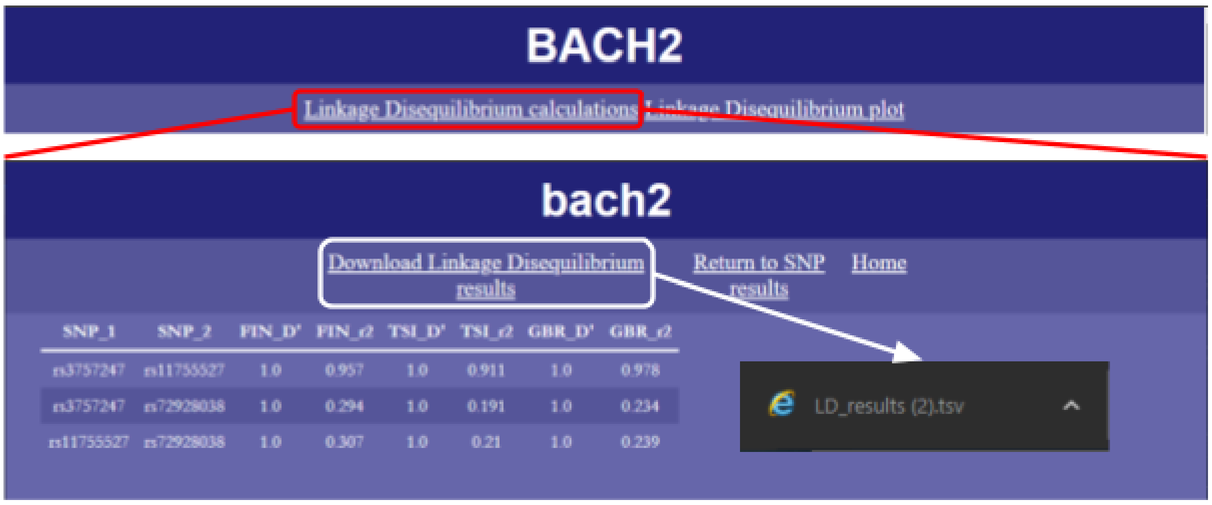
▼ rs1050979

GO ID	Description
GO:0032733	positive regulation of interleukin-10 production
GO:0034122	negative regulation of toll-like receptor signaling pathway
GO:0043388	positive regulation of DNA binding
GO:0043966	histone H3 acetylation
GO:0043967	histone H4 acetylation
GO:0120162	positive regulation of cold-induced thermogenesis

Fig. 14: Shows the quick links feature which allows users to jump from sections quickly rather than scroll through the page

LD

When a gene name or chromosomal coordinates is entered and multiple SNPs are returned, the option for the user to retrieve LD calculations is available if the user clicks the button highlighted by the red box called “linkage Disequilibrium calculations”. This will take them to a page containing a table showing the D’ and r2 values for all pairwise LD calculation for SNPs returned by the query for each of the populations if LD data is available. These results are downloadable as a tsv by the user by clicking the button highlighted by the White Box shown below.


BACH2
Linkage Disequilibrium calculations
Linkage Disequilibrium plot
bach2
Download Linkage Disequilibrium results
Return to SNP results
Home
LD_results (2).tsv

SNP_1	SNP_2	FIN_D'	FIN_r2	TSI_D'	TSI_r2	GBR_D'	GBR_r2
rs3757247	rs11755527	1.0	0.957	1.0	0.911	1.0	0.978
rs3757247	rs72928038	1.0	0.294	1.0	0.191	1.0	0.234
rs11755527	rs72928038	1.0	0.307	1.0	0.21	1.0	0.239

Fig. 15: This is the table of LD calculation results for the gene BACH2

The user can also return back to the results page to continue using the data from the previous query or return to the home page to do a new search. If the user returns to The SNP results they can also select the “Linkage Disequilibrium plot button” seen below. The user will be able to scroll through all of the LD plots for the different populations.

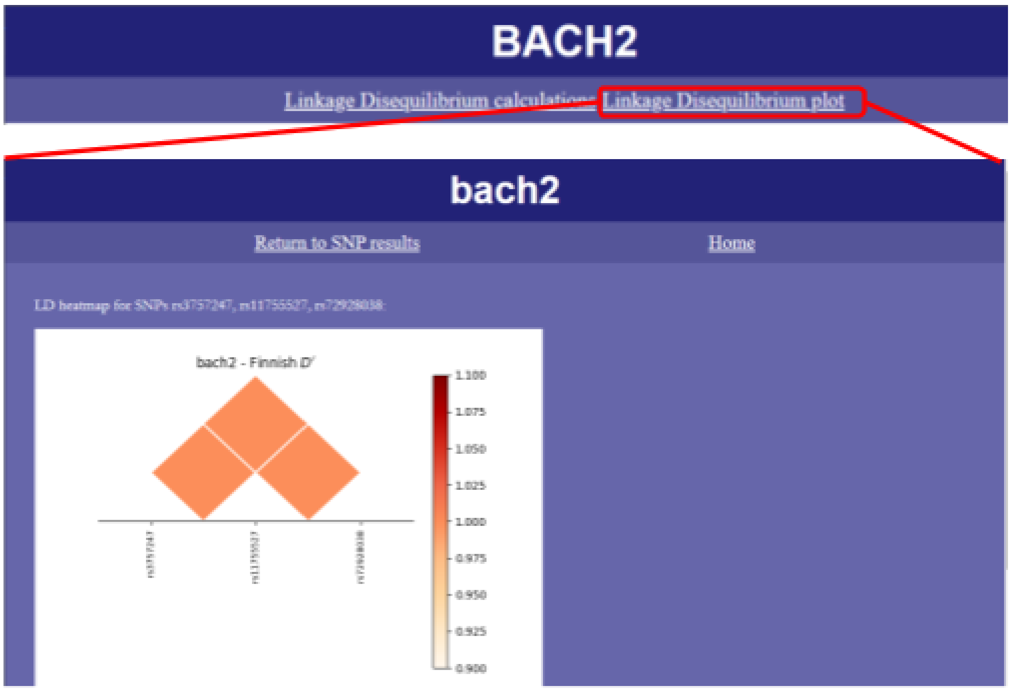


Fig. 16: This is the table of LD plot results for the gene BACH2

Manhattan plot

When the user puts an input in where several SNPs are present such as here where all of chromosome 6 has been entered, the Manhattan plot button highlighted in red will be present once clicked the user will be greeted with the interactive Manhattan plot. There are also buttons allowing the user to return to the homepage, or the results page as well as plot all the SNPs for T1D in GWAS.

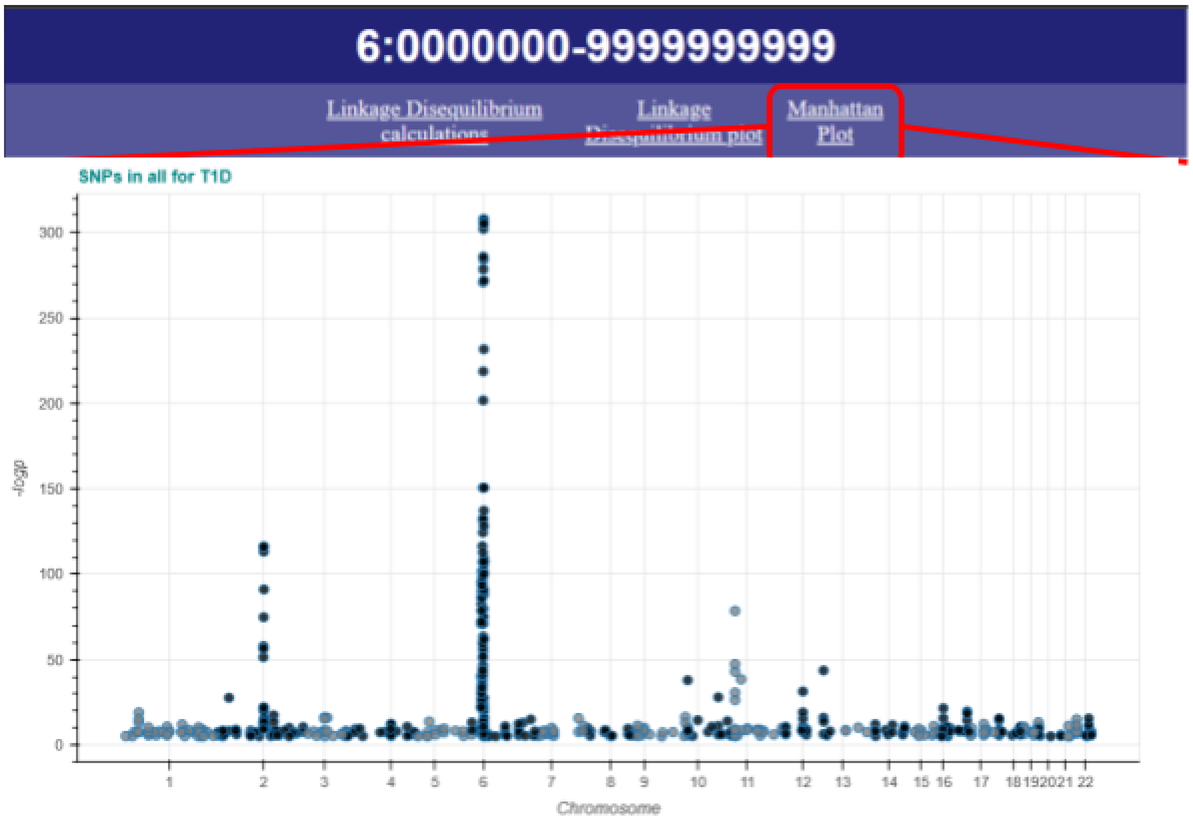


Fig. 17: Shows the Manhattan plot for all of the SNPs in chromosome 6

Limitations

- Linkage Disequilibrium heatmap plot scales poorly with queries that return a large number of SNPs due to having to filter the LD dataset, construct a matrix data frame and generate heatmap plots for six subsets of data, increasing time required to load the webpage. This is likely partially due to pandas dataframes being constructed row by row. Ideally a 2D series/List of lists should be used to create the data frame at once.
- Readability of SNPs labels in the plot is also greatly limited due to the small size of the plot. Further changes to plot parameters or using other packages for plotting LD data would need to be explored which can better accommodate LD plots for a larger number of SNPs.
- The Manhattan plot does not include all SNPs found within the database, this is due to insufficient information; there were missing p-values and chromosomal positions.

Future Developments

1. Although the database covers the entire genome, it still requires a few mapped genes for the SNPs, as well as the p-values. This can be done with more time and integrating data from other databases. This would also enable us to include additional SNPs in the Manhattan plot.
2. Currently the population data only looks at 3 populations; in future it should be possible to expand to all population data available from the 1000 Genomes Project for additional allele frequency.
3. Expand linkage disequilibrium data from Chr6 T1DM SNPs to T1DM SNPs genome wide.
4. Expand linkage disequilibrium data to additional 1000 Genomes Project populations. To reduce waiting times for loading the LD heatmap webpage, a way of enabling the user to specify which populations to create LD plots for would likely need to be implemented
5. Migrate Linkage Disequilibrium data to SQL database. This would allow each query to only process LD data for relevant SNPs, reducing time required to wrangle data to generate the LD results table and heatmap plots.
6. Add options to download Linkage Disequilibrium data as a matrix of either D’ or r^2 values for a population specified by the user. This would make subsequent analyses for the user easier as the user would not be required to manipulate the dataset into a matrix data structure.
7. Implement heatmap plots which have greater readability for queries returning a larger number of SNPs. Additionally, waiting times for generating plots could be reduced by also enabling the user to select which population and LD metric to create the plot with.

References

- Ashburner, M., Ball, C., Blake, J. *et al.* Gene Ontology: tool for the unification of biology. *Nat Genet* **25**, 25–29 (2000). https://doi.org/10.1038/75556
- British Dyslexia Association (2018) Dyslexia friendly style guide. Available at: https://www.bdadyslexia.org.uk/advice/employers/creating-a-dyslexia-friendly-workplace/dyslexia-friendly-style-guide (Accessed: March 1, 2023).
- CADD - Combined Annotation Dependent Depletion (2021). Available at: https://cadd.gs.washington.edu/info (Accessed: January 27, 2023).
- Carla Bizzarri, Patrizia Ippolita Patera, Claudia Arnaldi, Stefano Petrucci, Maria Luisa Manca Bitti, Raffaella Scrocca, Silvia Manfrini, Rosalba Portuesi, Raffaella Buzzetti, Marco Cappa, Paolo Pozzilli, the Immunotherapy Diabetes (IMDIAB) Group; Incidence of Type 1 Diabetes Has Doubled in Rome and the Lazio Region in the 0- to 14-Year Age-Group: A 6-Year Prospective Study (2004–2009). *Diabetes Care* 1 November 2010; 33 (11): e140. https://doi.org/10.2337/dc10-1168
- Crow YJ, Alberti KG, Parkin JM. Insulin dependent diabetes in childhood and material deprivation in northern England, 1977-86. BMJ. 1991 Jul 20;303(6795):158-60. doi: 10.1136/bmj.303.6795.158. PMID: 1878639; PMCID: PMC1670373.
- Junnila SK. Type 1 diabetes epidemic in Finland is triggered by zinc-containing amorphous silica nanoparticles. Med Hypotheses. 2015 Apr;84(4):336-40. doi: 10.1016/j.mehy.2015.01.021. Epub 2015 Jan 21. PMID: 25659493
- Rello, L. and Bigham, J. (2017) “Good background colors for readers,” Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility [Preprint]. Available at: https://doi.org/10.1145/3132525.3132546.
- Rentzsch, P., Schubach, M., Shendure, J., Kircher, M., 2021. CADD-Splice—improving genome-wide variant effect prediction using deep learning-derived splice scores. Genome Med. https://doi.org/10.1186/s13073-021-00835-9
- The Gene Ontology Consortium, The Gene Ontology resource: enriching a GOld mine, *Nucleic Acids Research*, Volume 49, Issue D1, 8 January 2021, Pages D325–D334, https://doi.org/10.1093/nar/gkaa1113
- Slatkin, M. (2008) “Linkage disequilibrium — understanding the evolutionary past and mapping the medical future,” Nature Reviews Genetics, 9(6), pp. 477–485. Available at: https://doi.org/10.1038/nrg2361.
- Machiela, M.J. and Chanock, S.J. (2015) “LDlink: A web-based application for exploring population-specific haplotype structure and linking correlated alleles of possible functional variants,” Bioinformatics, 31(21), pp. 3555–3557. Available at: https://doi.org/10.1093/bioinformatics/btv402.
- Rello, Luz and Jeffrey P. Bigham (2017). “Good Background Colors for Readers”. In: ASSETS ’17: The 19th International ACM SIGACCESS Conference on Computers and Accessibility. ACM. doi: 10.1145/3132525.3132546.