Project Documentation

Team Duck Duck Go

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 the p-values.

Prerequisites:

Functions used throughout:

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

```
def hello():
    print("Hello world!")
```

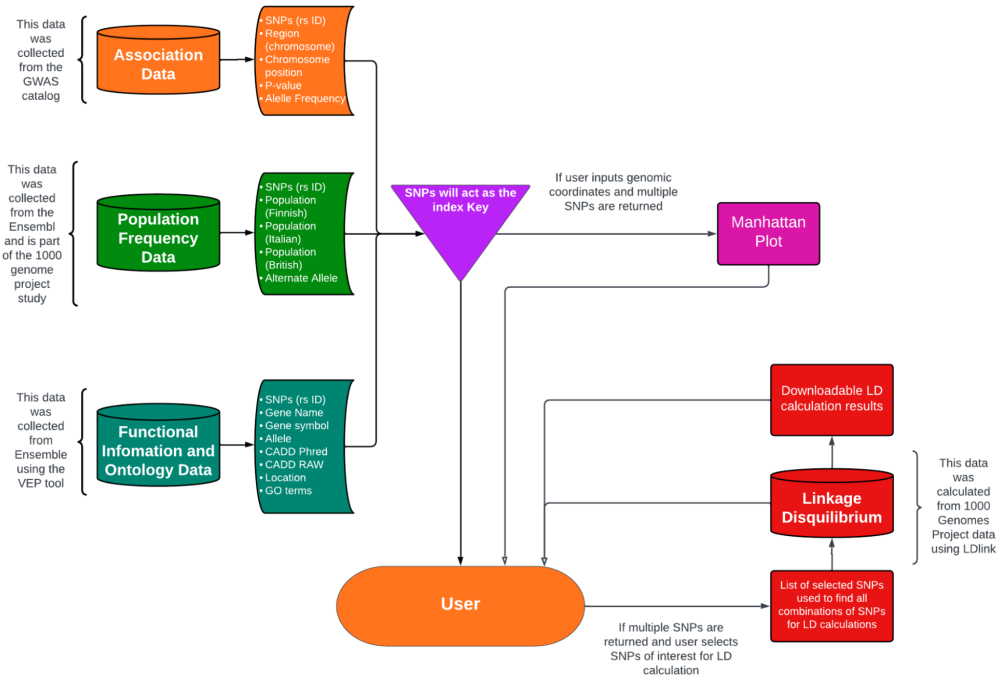
```
  naughtyList=[]                                # List of lists of (index, p-val) we
want to drop
for i in dupesDict:
    snp = dupesDict[i]                          # Get list of (index, pVal)
sortByP=sorted(snp,key=lambda x: 0-x[1])    # Sort by p-value
sortByP=sortByP[1:]                           # 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


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

```
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
```

Structure:



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)

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

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:

rsid: the reference SNP identifier for the variant.

Allele: the alternative allele observed at the variant site.

impact: the impact of the variant on the affected gene

consequence: the consequence of the variant on the affected gene

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.

Feature_Type: the type of genomic feature the variant is located in

Feature: the Ensembl ID of the specific feature the variant is located in

Exon: the exon number(s) affected by the variant.

Intron: the intron number(s) affected by the variant.

HGVSc: the HGVS nomenclature for the variant at the cDNA .

HGVSp: the HGVS nomenclature for the variant at the protein level.

cDNA_position: the position of the variant within the cDNA sequence of the affected gene.

CDS_position: the position of the variant within the coding sequence of the affected gene.

Protein_position: the position of the variant within the protein sequence of the affected gene.

Amino_acids: the amino acid change resulting from the variant.

Codons: the DNA codon change resulting from the variant.

Existing_variation: additional identifiers for the variant in other databases.

Distance: the distance to the nearest feature in the same or opposite strand.

Strand: the genomic strand the variant is located on.

FLAGS: additional information about the variant .

SYMBOL_SOURCE: the source of the gene symbol or name.

HGNC_ID: the HGNC gene ID of the affected gene.

MANE_SELECT: indication of whether this transcript is the MANE (Matched Annotation from NCBI and EMBL-EBI) Select transcript.

MANE_PLUS_CLINICAL: indication of whether this transcript is the MANE Select Plus Clinical (MPC) transcript.

TSL: transcript support level (a measure of transcript annotation confidence).

APPRIS: annotation of principal isoforms for each gene.

ENSP: the Ensembl protein ID of the affected protein.

SIFT: prediction of the effect of the variant on protein function.

PolyPhen: prediction of the effect of the variant on protein function.

CLIN_SIG: clinical significance of the variant.

SOMATIC: indication of whether the variant is somatic or germline.

PHENO: phenotype association of the variant.

PUBMED: PubMed ID of publications reporting functional evidence of the variant.

MOTIF_NAME: name of the DNA motif affected by the variant.

MOTIF_POS: position of the variant within the affected DNA motif.

HIGH_INF_POS: indication of whether the variant falls in a highly conserved position within the DNA motif.

MOTIF_SCORE_CHANGE: the effect of the variant on the score of the affected DNA motif.

TRANSCRIPTION_FACTORS: transcription factors that bind the affected DNA motif.

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 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 r2.

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 r2 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 r2 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 r2 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.

```
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 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.

```
# 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.

```
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 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.

```
# 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)
```

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.

```
# 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'])
# 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 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.

```
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 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.

```
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")
```

Manhattan Plot:

Flask:

Navigation:

Citation: