Additional File 1: GaiaAssociation source code. This python notebook file (.ipynb) includes the functional source code required to run GaiaAssociation.

```
##import neccesary libraries
import pandas as pd
import pyranges as pr
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
import operator
import scipy
from scipy.cluster.hierarchy import dendrogram, leaves_list
import scipy.spatial.distance as ssd
import os, glob
import sys
import seaborn as sns
import statistics
import math
import argparse
from scipy.optimize import root
from scipy.stats import norm
import warnings


####

#### Gaia Association:

#### Compare chromatin accessibility data (e.g. ATAC-seq, DNase-seq) against loci sets (e.g. de-novo mutations, SNPs, rare variants) to detect cell-specific enrichment of loci.

### Named for James Lovelock's Gaia Hypothesis

####


##Primary function for associating ATAC and GWAS
def gaiaAssociation(atacLocation, gwasLocation, chromosomeSize, outputLocation, uniqueCount=0, lociCutoff=0, lociSelection = 0, subsettingRegion=0, windowSize = 100000):

    ## ensure all the given locations and files are accesible and real
    if not os.path.exists(atacLocation):
        sys.exit("Cell region folder cannot be found")
    if not os.path.exists(gwasLocation):
        sys.exit("Loci folder cannot be found")
    if not os.path.exists(chromosomeSize):
        sys.exit("Chrom size file cannot be found")
```

```
43      if not os.path.exists(outputLocation):
44          print("Output folder cannot be found, attempting to build it:")
45          os.mkdir(outputLocation)
46          if not os.path.exists(outputLocation):
47              sys.exit("Output folder cannot be built")
48          else:
49              print("Output folder succesfully built:")
50
51      ## pyranges is a slightly outdated method and receives warnings about
… not properly identifying true or false values in certain instances
52      warnings.filterwarnings('ignore', module='pyranges')
53
54      ##Bring in our atac files for each cell type
55      dataFrameList = []
56      cellNames = []
57      for filename in glob.glob(atacLocation + "/" + '*.txt'):
58
59          loopFrame = pd.read_csv(filename, sep="\t")
60
61          ##this checks if there was no header, if the first column name is
… chr1, it shifts columns down into a row
62          if "chr1" in loopFrame.columns[0] or "Chr1" in
… loopFrame.columns[0]:
63              columnFrame = loopFrame.columns.to_frame().T
64              loopFrame = pd.concat([columnFrame,loopFrame],
… ignore_index=True)
65
66          ##Gives first three columns the names 'Chromosome', 'Start',
… "End"
67          loopFrame = loopFrame.rename({loopFrame.columns[0] :
… 'Chromosome', loopFrame.columns[1] : 'Start', loopFrame.columns[2] :
… "End"}, axis=1)
68
69          ##ensure start and stop are numeric values
70          loopFrame["Start"] = pd.to_numeric(loopFrame["Start"])
71          loopFrame["End"] = pd.to_numeric(loopFrame["End"])
72
73          ##ensure the chromosome id doesnt have white space and is the
… string datatype
74          loopFrame['Chromosome'] = loopFrame['Chromosome'].str.strip()
75          loopFrame['Chromosome'] = loopFrame['Chromosome'].astype(str)
76
77          ##add this dataframe to our list of atac-sets
78          dataFrameList.append(loopFrame)
79
80          ##Save cell type names by formatting the file names
```

```
81          s = filename[:-4]
82          cellNames.append(s[s.rindex('/')+1:])
83
84      if(len(dataFrameList)==0):
85          sys.exit("No Cell region files found")
86
87      ##do some light formating to ensure all atac-seq have the same
… labeling
88      for i in range(len(dataFrameList)):
89          if "seqnames" in dataFrameList[i].columns:
90              dataFrameList[i] =
… dataFrameList[i].rename(columns={"seqnames": "chr"})
91          if "chr" in dataFrameList[i].columns:
92              dataFrameList[i] = dataFrameList[i].rename(columns={"chr":
… "Chromosome"})
93          if "start" in dataFrameList[i].columns:
94              dataFrameList[i] = dataFrameList[i].rename(columns={"start":
… "Start"})
95          if "end" in dataFrameList[i].columns:
96              dataFrameList[i] = dataFrameList[i].rename(columns={"end":
… "End"})
97
98          ##ensure there is a size column and a Size column, this
… redundancy exists because of different package dependencies request for
… formatting
99          dataFrameList[i]["Size"] = dataFrameList[i]["End"] -
… dataFrameList[i]["Start"]
100
101     ##create an empty matrix to store our associations in
102     matrix1 = np.zeros((len(dataFrameList),len(dataFrameList)))
103
104     print("Formatting Cell Regions: ")
105
106     ##create list to store our pyRanges objects in, merge them to ensure
… regions arent double represented
107     prRanges = []
108     for i in range(len(dataFrameList)):
109
110         ##this particular bit of circular coding creates a pyrange,
… merges itself to remove redundancy, turns it back into a dataframe,
… creates a size column, and then turns it back into a pyrange. This is
… because pyrange doesnt have a function to add a size column and merging
… removes the size column, this tedium is their fault, but it isn't super
… time intensive
111         tempRange = pr.PyRanges(dataFrameList[i])
112         loopRange = tempRange.merge(strand=False)
```

```
113         df = pd.DataFrame(list(zip(loopRange.Chromosome, loopRange.Start,
…  loopRange.End)),columns =['Chromosome', 'Start', 'End'])
114         df["Size"] = df["End"] - df["Start"]
115         testRange = pr.PyRanges(df)
116         prRanges.append(testRange)
117 
118     if uniqueCount != 0:
119 
120         ## Create unique sets of ATAC based on a cutoff value of number
…  of overlaps
121         overlapCutoff = int(uniqueCount)
122 
123         ##create New locations to store unique regions
124         prRangesUnique = []
125         prFrames = []
126 
127         ##save the current dataframes to new location for altering (this
…  may be redundant based on python memory handling)
128         print("Finding Unique Cell Region Peaks Between Cell Types:")
129         for j in range(len(prRanges)):
130             prFrames.append(prRanges[j].df)
131 
132         ##Loop through each atac set and compare against all others,
…  remove locations with overlaps greater than overlap cutoff
133         for j in range(len(prFrames)):
134 
135             if( j != len(prFrames) - 1):
136                 print("Cell Type " + str(j+1), end = " - ")
137                 testFrame = pd.concat(prFrames[:j] + prFrames[j+1 :])
138             else:
139                 print("Cell Type " + str(j+1), end = "\n")
140                 testFrame = pd.concat(prFrames[:j])
141 
142 
143 
144             allButMain = pr.PyRanges(testFrame)
145             loopRange = prRanges[j].coverage(allButMain)
146             mainFrame = loopRange.df
147             mainFrame = mainFrame[mainFrame["NumberOverlaps"] <=
…  overlapCutoff]
148             mainFrame = mainFrame.drop(columns=['NumberOverlaps',
…  'FractionOverlaps'])
149             mainRange = pr.PyRanges(mainFrame)
150             prRangesUnique.append(mainRange)
151 
152 
```

```
153             if not os.path.exists(outputLocation + '/unique_regions'):
154                 os.mkdir(outputLocation + '/unique_regions')
155                 if not os.path.exists(outputLocation +
… '/unique_regions'):
156                     print("Output folder for new unique regions cannot be
… built")
157                     uniqueSave = False
158                 else:
159                     uniqueSave = True
160             else:
161                 uniqueSave = True
162 
163             if uniqueSave == True:
164                 mainFrame.to_csv(outputLocation + '/unique_regions' +
… '/unique_regions_' + cellNames[j][:10] + "_" + str(uniqueCount) +
… '.txt',sep='\t')
165 
166 
167         prRanges = prRangesUnique
168 
169 
170     ## If a merging region file is included then subset our ATAC sets
… using this region set
171     if subsettingRegion != 0:
172 
173         print("Subsetting Cell Region Peaks Based on given mask region
… file:")
174 
175         geneLocation = subsettingRegion
176         geneFrame = pd.read_csv(geneLocation, sep="\t")
177 
178         geneFrame = geneFrame.rename(columns={"Chr": "Chromosome",
… "start": "Start", "end" : "End"})
179 
180         geneRange = pr.PyRanges(geneFrame)
181 
182         if "seqnames" in geneFrame.columns:
183             geneFrame = geneFrame.rename(columns={"seqnames": "chr"})
184         if "chr" in geneFrame.columns:
185             geneFrame = geneFrame.rename(columns={"chr": "Chromosome"})
186         if "start" in geneFrame.columns:
187             geneFrame = geneFrame.rename(columns={"start": "Start"})
188         if "end" in geneFrame.columns:
189             geneFrame = geneFrame.rename(columns={"end": "End"})
190 
191         ## Only keep regions which intersect with this subsetting region
```

```
192         overlappedForms = []
193         overlappedRanges = []
194         for item in prRanges:
195             loopRange = geneRange.intersect(item)
196             loopRange = loopRange.merge(strand=False)
197             loopFrame = loopRange.df
198 
199             loopFrame["Size"] = loopFrame["End"] - loopFrame["Start"]
200 
201             loopRange = pr.PyRanges(loopFrame)
202 
203             overlappedForms.append(loopFrame)
204             overlappedRanges.append(loopRange)
205 
206         prRanges = overlappedRanges
207         dataFrameList = overlappedForms
208 
209 
210     print("Comparing cell regions between cell types: ")
211 
212     ##this function loops through each cell type and compares it to every
… cell type with a lower index number than itself. the comparison is a
… simple overlap calculation, it then adds these values to its respective
… matrix
213     for i in range(len(dataFrameList)):
214 
215         if( i != len(dataFrameList) - 1):
216             print("Cell Type " + str(i+1), end = " - ")
217         else:
218             print("Cell Type " + str(i+1), end = "\n")
219 
220         for j in range(len(dataFrameList)):
221 
222             if i > j:
223 
224                 loopRange = prRanges[i].intersect(prRanges[j])
225                 if loopRange.df.empty:
226                     matrix1[i,j] = 1
227                     matrix1[j,i] = 1
228                 else:
229                     overlap = sum(loopRange.End) - sum(loopRange.Start)
230                     sizeA = sum(prRanges[i].Size)
231                     sizeB = sum(prRanges[j].Size)
232 
233                     weightValue = 1- ((overlap)/ (sizeA) *
… (overlap)/(sizeB))
```

```
234
235                     matrix1[i,j] = weightValue
236
237                     matrix1[j,i] = weightValue
238
239     ## If the user only included one atac-seq we need to skip steps which
… would fail with only atac, this includes making a dendrogram using scipy
240     if len(dataFrameList) > 1:
241         ##Create the first figure, the dendrogram of ATAC-seq datasets
242         distArray = ssd.squareform(matrix1)
243         plt.figure(figsize=(10, 16))
244         Z = scipy.cluster.hierarchy.linkage(distArray, method='single',
… metric='euclidean')
245         dn = dendrogram(Z, labels=cellNames, orientation = "left")
246         plt.savefig(outputLocation + '/cell_region_dendrogram.pdf',
… bbox_inches = "tight")
247     else:
248         print("You have only included one cell region set, this will omit
… the cell region dendrogram")
249
250     print("Formatting loci:")
251
252     ##Bring in our Loci sets
253     gwasFrame = 0
254
255     ## columns we wish to save
256     final_columns = ["CHR_ID", "Start", "CHR_POS", "End", "Chromosome",
… "DISEASE/TRAIT"]
257     if lociSelection != 0:
258
259         if lociSelection[-3:] == "tsv":
260             lociSelectDF=pd.read_csv(lociSelection,sep='\t')
261         elif lociSelection[-3:] == "csv":
262             lociSelectDF=pd.read_csv(lociSelection)
263         tempColumnList = lociSelectDF.columns
264         final_columns.extend(tempColumnList)
265
266
267
268     ##add GWAS in tsv format
269     for filename in glob.glob(gwasLocation + "/" + '*.tsv'):
270         loopFrame = pd.read_csv(filename, sep="\t", low_memory=False)
271
272         ## give a backup id by file count, in case they dont have an
… identity column
273         s = filename[:-4]
```

```
274         loopFrame["fileId"] =  (s[s.rindex('/')+1:])
275 
276         loopFrame = loopFrame[(loopFrame[loopFrame.columns[0]].notna()) &
…  (loopFrame[loopFrame.columns[1]].notna())]
277 
278         ##These if statements are to account for various formatting
…  mistakes
279         if "DISEASE/TRAIT" not in loopFrame.columns:
280             loopFrame["DISEASE/TRAIT"] = loopFrame["fileId"]
281         if "Start" in loopFrame.columns:
282             loopFrame["CHR_POS"] = loopFrame["Start"].astype(int)
283         if "start" in loopFrame.columns:
284             loopFrame["CHR_POS"] = loopFrame["start"].astype(int)
285         if "Reference_name" in loopFrame.columns:
286             loopFrame["Chromosome"] = loopFrame["Reference_name"]
287             loopFrame["CHR_ID"] = loopFrame["Reference_name"]
288         if "reference_name" in loopFrame.columns:
289             loopFrame["Chromosome"] = loopFrame["reference_name"]
290             loopFrame["CHR_ID"] = loopFrame["reference_name"]
291         if "Chromosome" in loopFrame.columns:
292             loopFrame["CHR_ID"] = loopFrame["Chromosome"]
293         if "chr" in loopFrame.columns:
294             loopFrame["CHR_ID"] = loopFrame["chr"]
295 
296         loopFrame["Chromosome"] = loopFrame["CHR_ID"]
297 
298         ##if gwas is missing chr part of chromosome ids than add it
299         if (1 in loopFrame.Chromosome.unique()) or ('1' in
…  loopFrame.Chromosome.unique()):
300                     loopFrame["Chromosome"] = "chr" +
…  loopFrame["Chromosome"]
301 
302 
303         if not isinstance(gwasFrame, pd.DataFrame):
304 
…  loopFrame.drop(columns=loopFrame.columns.difference(final_columns),
…  inplace=True)
305             loopFrame = loopFrame[pd.to_numeric(loopFrame['CHR_POS'],
…  errors='coerce').notnull()]
306             loopFrame['CHR_ID'] = loopFrame['CHR_ID'].astype('category')
307             gwasFrame = loopFrame
308         else:
309 
…  loopFrame.drop(columns=loopFrame.columns.difference(final_columns),
…  inplace=True)
310             loopFrame = loopFrame[pd.to_numeric(loopFrame['CHR_POS'],
```

```
errors='coerce').notnull()]
            loopFrame['CHR_ID'] = loopFrame['CHR_ID'].astype('category')
            gwasFrame = pd.concat([gwasFrame, loopFrame],axis=0,
ignore_index=True)


    ## add GWAS in csv format
    for filename in glob.glob(gwasLocation + "/" + '*.csv'):

        loopFrame = pd.read_csv(filename, low_memory=False)

        ## give a backup id by file count, in case they dont have an
identity column
        s = filename[:-4]
        loopFrame["fileId"] =  (s[s.rindex('/')+1:])

        loopFrame = loopFrame[(loopFrame[loopFrame.columns[0]].notna()) &
(loopFrame[loopFrame.columns[1]].notna())]


        ##These if statements are to account for various formatting
mistakes
        if "DISEASE/TRAIT" not in loopFrame.columns:
            loopFrame["DISEASE/TRAIT"] = loopFrame["fileId"]
        if "Start" in loopFrame.columns:
            loopFrame["CHR_POS"] = loopFrame["Start"].astype(int)
        if "start" in loopFrame.columns:
            loopFrame["CHR_POS"] = loopFrame["start"].astype(int)
        if "Reference_name" in loopFrame.columns:
            loopFrame["Chromosome"] = loopFrame["Reference_name"]
            loopFrame["CHR_ID"] = loopFrame["Reference_name"]
        if "reference_name" in loopFrame.columns:
            loopFrame["Chromosome"] = loopFrame["reference_name"]
            loopFrame["CHR_ID"] = loopFrame["reference_name"]
        if "Chromosome" in loopFrame.columns:
            loopFrame["CHR_ID"] = loopFrame["Chromosome"]
        if "chr" in loopFrame.columns:
            loopFrame["CHR_ID"] = loopFrame["chr"]

        loopFrame["Chromosome"] = loopFrame["CHR_ID"]

        ##if gwas is missing chr part of chromosome ids than add it
        if (1 in loopFrame.Chromosome.unique()) or ('1' in
loopFrame.Chromosome.unique()):
                    loopFrame["Chromosome"] = "chr" +
loopFrame["Chromosome"]
```

```
350
351         if not isinstance(gwasFrame, pd.DataFrame):
352
…   loopFrame.drop(columns=loopFrame.columns.difference(final_columns),
…   inplace=True)
353             loopFrame = loopFrame[pd.to_numeric(loopFrame['CHR_POS'],
…   errors='coerce').notnull()]
354             loopFrame['CHR_ID'] = loopFrame['CHR_ID'].astype('category')
355             gwasFrame = loopFrame
356         else:
357
…   loopFrame.drop(columns=loopFrame.columns.difference(final_columns),
…   inplace=True)
358             loopFrame = loopFrame[pd.to_numeric(loopFrame['CHR_POS'],
…   errors='coerce').notnull()]
359             loopFrame['CHR_ID'] = loopFrame['CHR_ID'].astype('category')
360             gwasFrame = pd.concat([gwasFrame, loopFrame],axis=0,
…   ignore_index=True)
361
362     if not isinstance(gwasFrame, pd.DataFrame):
363         sys.exit("No Loci files found")
364
365     ##format our Loci frame by getting rid of NAN values and then getting
…   rid of studies with non-numerical chromosome positions, if you want to
…   fix these chromosome positions manually, you can
366     print("Removing Empty Loci Values:")
367     gwasNoNAN = gwasFrame[(gwasFrame["CHR_POS"].notna()) &
…   (gwasFrame["DISEASE/TRAIT"].notna())]
368
369     gwasNoNAN = gwasNoNAN.reset_index(drop=True)
370
371     ## Location to save all non-numeric indexes
372     nonNumericIndexes = []
373
374     ## if chromosome position isn't an integer than check for non-int
…   rows to remove or try to force into int
375     if gwasNoNAN["CHR_POS"].dtype != np.int64:
376         print("Non integer chromosome positions detected, these will be
…   removed")
377         for index, row in gwasNoNAN.iterrows():
378             if isinstance(gwasNoNAN.at[index, "CHR_POS"], float):
379                 gwasNoNAN.at[index, "CHR_POS"] = int(gwasNoNAN.at[index,
…   "CHR_POS"])
380             elif isinstance(gwasNoNAN.at[index, "CHR_POS"], str):
381                 if gwasNoNAN.at[index, "CHR_POS"].isdigit():
382                     gwasNoNAN.at[index, "CHR_POS"] =
```

```
382… int(gwasNoNAN.at[index, "CHR_POS"])
383                 else:
384                     nonNumericIndexes.append(index)
385             elif not np.issubdtype(gwasNoNAN.at[index, "CHR_POS"], int):
386                 nonNumericIndexes.append(index)
387
388     ## Drop indexes of values that couldnt be forced into integer
389     gwasFormatted = gwasNoNAN.drop(index=nonNumericIndexes)
390
391     ##Create columns for turning into pyranges
392     gwasFormatted["Start"] = gwasFormatted["CHR_POS"]
393     if "End" not in gwasFrame.columns:
394         gwasFormatted["End"] = gwasFormatted["CHR_POS"] + 1
395     if "Chromosome" not in gwasFrame.columns:
396         temp = '\t'.join(gwasFrame.CHR_ID.unique())
397         if "chr" not in temp:
398             gwasFormatted["Chromosome"] = "chr" +
… gwasFormatted["CHR_ID"].astype(str)
399         else:
400             gwasFormatted["Chromosome"] =
… gwasFormatted["CHR_ID"].astype(str)
401
402     ##if the length is zero give it a 1 length
403     gwasFormatted.loc[gwasFormatted.Start == gwasFormatted.End, "End"] =
… gwasFormatted.loc[gwasFormatted.Start == gwasFormatted.End, "End"] + 1
404
405     ## Subset by loci selection if given
406     if lociSelection != 0:
407         gwasNames = []
408
409         eachLociFrame = []
410         columnList = lociSelectDF.columns
411         for index, row in lociSelectDF.iterrows():
412             temp = lociSelectDF.iloc[[index]]
413             dtName = str(list(temp[columnList[0]])[0])
414             lociLoopFrame = gwasFormatted[gwasFormatted[columnList[0]] ==
… list(temp[columnList[0]])[0]]
415             for item in columnList:
416                 lociLoopFrame = lociLoopFrame[lociLoopFrame[item] ==
… list(temp[item])[0]]
417                 dtName = dtName + str(list(temp[item])[0])
418             eachLociFrame.append(lociLoopFrame)
419             gwasNames.append(dtName)
420             if lociLoopFrame.shape[0] == 0:
421                 print(dtName)
422                 sys.exit("One of the user given loci groups does not
```

```
exist")



    ### rename the Disease/Trait column to more easily callable DT column
    gwasFormatted = gwasFormatted.rename(columns={"DISEASE/TRAIT": "DT"})

    ## subset loci based on cutoff value
    if lociCutoff != 0:

        print("Subsetting loci based on user-defined count:")
        vc = gwasFormatted.DT.value_counts()
        highCount = list(vc[vc > int(lociCutoff)].index)
        gwasFormatted =
gwasFormatted[(gwasFormatted["DT"].isin(highCount))]

    ##save pyranges for each disease/trait
    print("Turning GWAS into Pyranges Objects:")
    gwasPyranges = []
    gwasNonZeroPyranges = []
    indels = False

    if lociSelection == 0:
        gwasNames = list(set(gwasFormatted["DT"]))
        for item in set(gwasFormatted["DT"]):

            ##subset for each loci type
            loopFrame = gwasFormatted[(gwasFormatted["DT"] == item)]

            ##remove duplicate loci
            loopFrame = loopFrame.drop_duplicates(subset=['CHR_ID',
'Start', 'End'])

            ##Subset into appropriate columns and add to our list of
final loci ranges
            tempRange = pr.PyRanges(loopFrame)
            loopRange = tempRange
            df = pd.DataFrame(list(zip(loopRange.Chromosome,
loopRange.Start, loopRange.End)),columns =['Chromosome', 'Start', 'End'])
            df["Size"] = df["End"] - df["Start"]
            dfSingle = df[df["Size"] <= 1]
            testRange = pr.PyRanges(dfSingle)
            gwasPyranges.append(testRange)

            ##If there are indels we need to create range including all
these larger than 1 bp loci, otherwise mark that it is blank
```

```
463             if df.Size.max() > 1:
464                 print("Indels Detected in " + item + ", Statistical
… Method For Loci >1 bp Will Be Applied")
465                 indels = True
466                 df = df[df["Size"] > 1]
467                 indelLoopRange = pr.PyRanges(df)
468                 gwasNonZeroPyranges.append(indelLoopRange)
469             else:
470                 gwasNonZeroPyranges.append([])
471 
472     else:
473         for countIndel, item in enumerate(eachLociFrame):
474 
475             ##remove duplicate loci
476             loopFrame = item.drop_duplicates(subset=['CHR_ID', 'Start',
… 'End'])
477 
478             ##Subset into appropriate columns and add to our list of
… final loci ranges
479             tempRange = pr.PyRanges(loopFrame)
480             loopRange = tempRange
481             df = pd.DataFrame(list(zip(loopRange.Chromosome,
… loopRange.Start, loopRange.End)),columns =['Chromosome', 'Start', 'End'])
482             df["Size"] = df["End"] - df["Start"]
483             dfSingle = df[df["Size"] <= 1]
484             testRange = pr.PyRanges(dfSingle)
485             gwasPyranges.append(testRange)
486 
487             ##If there are indels we need to create range including all
… these larger than 1 bp loci, otherwise mark that it is blank
488             if df.Size.max() > 1:
489                 print("Indels Detected in " + gwasNames[countIndel] +
… "Statistical Method For Loci >1 BP Will Be Applied")
490                 indels = True
491                 df = df[df["Size"] > 1]
492                 indelLoopRange = pr.PyRanges(df)
493                 gwasNonZeroPyranges.append(indelLoopRange)
494             else:
495                 gwasNonZeroPyranges.append([])
496 
497     ##Create a reordered list of our ATAC-sets based on how they showed
… up in the dendrogram. This step will be skipped if there is only one
… ATAC-seq set
498     reorderPyranges = []
499     reorderDataframeList = []
500     reorderCellNames = []
```

```
501     if len(dataFrameList) > 1:
502         ## using the leaves from our scipy Z set, resort your pyranges
… and dataframes to be in the correct order
503         for item in reversed(leaves_list(Z)):
504             reorderPyranges.append(prRanges[item])
505             reorderDataframeList.append(dataFrameList[item])
506             reorderCellNames.append(cellNames[item])
507     else:
508         reorderPyranges = prRanges
509         reorderDataframeList = dataFrameList
510 
511     ## create matrices to store p-values and overlap information
512     heatmapMatrix = np.zeros((len(reorderPyranges),len(gwasPyranges)))
513     overlapMatrix = np.zeros((len(reorderPyranges),len(gwasPyranges)))
514 
515     ## create the chromosome ratios for every atac-seq set
516     chromSize = pd.read_csv(chromosomeSize, thousands=',')
517     chromSize = chromSize.sort_values(by=['Chromosome'])
518 
519 
520     ##create weight matrix
521 
522     ##Based on window sizes divide our chromosomes into equal sized
… chuncks
523     windowFrames = []
524     for index, row in chromSize.iterrows():
525 
526         ## get the size of this particular chromsome
527         chrSize = int(chromSize.at[index, "size in bp"])
528 
529         ## returned the rounded up version of the divided window size
… (3.01 becomes 4)
530         n = math.ceil(chrSize/windowSize)
531 
532         windowSizesLoop = []
533         zp = n - (chrSize % n)
534         pp = chrSize//n
535         for i in range(n):
536             if(i>= zp):
537                 windowSizesLoop.append(pp + 1)
538             else:
539                 windowSizesLoop.append(pp)
540 
541         chrWindowDF = pd.DataFrame(windowSizesLoop, columns =['Size'])
542         chrWindowDF["End"] = chrWindowDF["Size"].cumsum()
543         chrWindowDF["Start"] = chrWindowDF["End"] - chrWindowDF["Size"] +
```

```
543… 1
544          chrWindowDF.at[0,"Start"] = chrWindowDF.at[0,"Start"] -1
545          chrWindowDF["Chromosome"] = chromSize.at[index, "Chromosome"]
546
547          windowFrames.append(chrWindowDF)
548
549      allWindowsDF = pd.concat(windowFrames)
550
551      ## find overlaps and probability for each ATAC + GWAS
552      listOverlaps = []
553      overlapAllColumns = []
554
555      windowsRange = pr.PyRanges(allWindowsDF)
556      windowStarts = list(allWindowsDF.Start)
557      windowEnds = list(allWindowsDF.End)
558      windowChrs = list(allWindowsDF.Chromosome)
559      dictWindows = {"Start":windowStarts, "End":windowEnds,
… "Size":list(allWindowsDF.Size), "Chromosome":windowChrs}
560
561      print("Calculating Overlaps: ")
562      if indels == True:
563          print("Note: Indels signifigantly effect runtime")
564      for count, item in enumerate(reorderPyranges):
565
566          if( count != len(reorderPyranges) - 1):
567              print(reorderCellNames[count], end = " - ")
568          else:
569              print(reorderCellNames[count], end = "\n")
570
571          tempRange = windowsRange.coverage(item)
572          dictWindows[reorderCellNames[count]] =
… list(tempRange.FractionOverlaps)
573
574          ##if there are indels then save the number of atac peaks per each
… window, this will be need for calculations at the end
575          if indels == True:
576              dictWindows[reorderCellNames[count] + "PeakCount"] =
… list(tempRange.NumberOverlaps)
577
578          ##find the number of overlaps
579          for count2, item2 in enumerate(gwasPyranges):
580              tempRange2 = windowsRange.coverage(item2)
581              dictWindows[reorderCellNames[count] + gwasNames[count2]] =
… list(tempRange2.NumberOverlaps)
582
583              ## If Indels exist we need to do some more expansive
```

```
583… calculations
584             if indels == True:
585
586                 ## if indels exist in this particular gwas set continue
…  on to indel caluclations
587                 if gwasNonZeroPyranges[count2]:
588
589                     ##Grab all indels that overlap with the current ATAC
…  set
590                     indelRangeTemp =
…  gwasNonZeroPyranges[count2].coverage(item)
591                     indelOverlap = indelRangeTemp.df
592
593                     ## keep only those that overlap
594                     indelOverlap =
…  indelOverlap[indelOverlap["NumberOverlaps"] > 0]
595
596                     ##now create a list of every bp where an indel exists
…  at
597                     indelOverlap = indelOverlap.reset_index(drop=True)
598                     indelOverlap['range']=indelOverlap.apply(lambda x :
…  list(range(x['Start'],x['End'])),1)
599                     indelChroDict = {}
600                     for item in list(set(indelOverlap.Chromosome)):
601                         indelChroLoop =
…  indelOverlap[indelOverlap["Chromosome"] == item]
602                         indelLoopLocations = [item for sublist in
…  indelChroLoop["range"] for item in sublist]
603                         indelChroDict[item] = indelLoopLocations
604
605                     indelExtraOverlap = []
606
607                     ##Get the number of these overlaps for each window so
…  we can add them to the count
608                     indelDoubleOverlap = indelOverlap.drop(['range',
…  'NumberOverlaps'], axis=1)
609                     indelDoubleOverlap = pr.PyRanges(indelDoubleOverlap)
610                     doubleOverlapCount =
…  windowsRange.coverage(indelDoubleOverlap)
611                     doubleOverlapList =
…  list(doubleOverlapCount.NumberOverlaps)
612
613                     ## for each window space, find how much length of
…  indel exists in the window
614                     for windowCount, windowItem in
…  enumerate(windowStarts):
```

```
615
616                         if windowChrs[windowCount] in indelChroDict:
617                             indelWindowSize = [num for num in
…  indelChroDict[windowChrs[windowCount]] if num <= windowEnds[windowCount]
…  and num >= windowStarts[windowCount]]
618                         else:
619                             indelWindowSize = []
620
621                         ## if none exist, add zero to list of indel
…  overlap, if it does add the total number of bp that do
622                         if len(indelWindowSize) == 0:
623                             indelExtraOverlap.append(0)
624                         else:
625                             indelSize = len(indelWindowSize)
626                             indelExtraOverlap.append(indelSize)
627
628                     ##grab the number of peaks per window
629                     listCount = dictWindows[reorderCellNames[count] +
…  "PeakCount"]
630
631                     ##multiply the extra length we need to add to every
…  atac peak within every window
632                     listSize = [a*b for a,b in
…  zip(listCount,indelExtraOverlap)]
633
634                     ## take this extra size to calculate an added
…  probability
635                     listProb = [a/b for a,b in zip(listSize,
…  dictWindows["Size"])]
636
637                     ##Add this probability to the base probability for a
…  gwas specific probability
638                     dictWindows[reorderCellNames[count]+
…  gwasNames[count2] + "Prob"] = [a+b for a,b in
…  zip(dictWindows[reorderCellNames[count]], listProb)]
639
640                     ##Add number of indel overlaps
641                     dictWindows[reorderCellNames[count] +
…  gwasNames[count2] + "IndelOverlap"] =  doubleOverlapList
642
643                     ## Get the total number of indels in region to add to
…  count
644                     totalIndelRange =
…  windowsRange.coverage(gwasNonZeroPyranges[count2])
645                     totalIndelList = list(totalIndelRange.NumberOverlaps)
646
```

```
                    ## add the count of indels to the original gwas count
                    dictWindows[reorderCellNames[count] +
gwasNames[count2]] = [a+b for a,b in
zip(dictWindows[reorderCellNames[count] + gwasNames[count2]],
totalIndelList)]


    ## check to see if you can make a subfolder
    if not os.path.exists(outputLocation+ '/overlaps'):
        os.mkdir(outputLocation + '/overlaps')
        if not os.path.exists(outputLocation + '/overlaps'):
            print("Output folder cannot be modified, so overlaps will not
be saved. Try running again with higher permissions.")
            overlapSave = False
        else:
            overlapSave = True
    else:
        overlapSave = True

    print("Calculating P-Values: ")
    for count, item in enumerate(reorderPyranges):

        if( count != len(reorderPyranges) - 1):
            print(reorderCellNames[count], end = " - ")
        else:
            print(reorderCellNames[count], end = "\n")

        for count2, item2 in enumerate(gwasPyranges):
            overlapCoverage = item.coverage(item2)
            if overlapSave:
                overlapCoverageFrame = overlapCoverage.df
                overlapCoverageFrame =
overlapCoverageFrame[overlapCoverageFrame["NumberOverlaps"] != 0]
                overlapCoverageFrame.to_csv(outputLocation + '/overlaps'
+ '/overlap_' + gwasNames[count2][:10] + '_' +
reorderCellNames[count][:10] + '.txt',sep='\t')
            overlapValue = sum(list(item.coverage(item2).NumberOverlaps))
            overlapMatrix[count,count2] = overlapValue
            if indels == True:
                overlapValue = overlapValue +
sum(dictWindows[reorderCellNames[count] + gwasNames[count2] +
"IndelOverlap"])


            ## if there are not indels, use the normal probabilities, if
there are indels, use the gwas specific window probabilites
```

```
683             if indels == False:
684                 out1, out2 = zip(*filter(all,
… zip(list(dictWindows[reorderCellNames[count] + gwasNames[count2]]),
… list(dictWindows[reorderCellNames[count]])))
685             elif gwasNonZeroPyranges[count2]:
686                 out1, out2 = zip(*filter(all,
… zip(list(dictWindows[reorderCellNames[count] + gwasNames[count2]]),
… list(dictWindows[reorderCellNames[count]+ gwasNames[count2] + "Prob"])))
687             else:
688                 out1, out2 = zip(*filter(all,
… zip(list(dictWindows[reorderCellNames[count] + gwasNames[count2]]),
… list(dictWindows[reorderCellNames[count]])))
689 
690             ## since psinib is a statistical method, sometimes root
… function finds results well outside the region near 0, this check
… accomodates this by turning all values over 1 into 1
691             valueSinib = psinib(overlapValue-1, out1, out2,
… lowerTail=False)
692             if valueSinib <= 1:
693                 heatmapMatrix[count,count2] = valueSinib
694             else:
695                 print("P-Value equal greater than one found. note: Sinib
… sometimes uses a statistical approximation of a p-value for non
… convergent cases, this can allow for impossible p-values. These  values
… are set to 1 in results.")
696                 heatmapMatrix[count,count2] = 1
697     try:
698         print("Lowest P-Value: " + str(heatmapMatrix.min()))
699     except ValueError:
700         pass
701     try:
702         if heatmapMatrix.max() > 1:
703             print("P-Value greater than one found in final matrix.")
704     except ValueError:
705         pass
706 
707 
708     newHeatMatrix = heatmapMatrix
709     ax = sns.heatmap(newHeatMatrix,
… cmap="YlGnBu",annot=True,xticklabels=gwasNames)
710     ax.xaxis.set_ticks_position('top')
711     ax.xaxis.set_label_position('top')
712     plt.xticks(rotation=90)
713     plt.savefig(outputLocation + '/pvalues_heatmap.pdf', bbox_inches =
… "tight")
714 
```

```
715     ## Create the final figure
716     figure, axis = plt.subplots(2, 2, figsize=(40, 30))
717     sns.heatmap(newHeatMatrix, cmap="YlGnBu", cbar=False, ax=axis[1,
…   1],xticklabels=gwasNames)
718     axis[1, 1].xaxis.set_ticks_position('top')
719     axis[1, 1].xaxis.set_label_position('top')
720     plt.xticks(rotation=90)
721     sns.heatmap(newHeatMatrix, cmap="YlGnBu", annot=True, ax=axis[0, 0])
722 
723     axis[0, 1].grid(False)
724     axis[0, 1].axis('off')
725     if len(dataFrameList) > 1:
726         dendrogram(Z,  labels=cellNames, orientation = "left", ax=axis[1,
…   0])
727     else:
728         axis[1, 0].text(0.1, 0.5, cellNames[0],
…   horizontalalignment='right', verticalalignment='center',
…   transform=axis[1, 0].transAxes)
729         axis[1, 0].grid(False)
730         axis[1, 0].axis('off')
731     axis[1, 1].margins(x=0)
732     plt.savefig(outputLocation + '/complete_figure.pdf', bbox_inches =
…   "tight")
733 
734     ##save the matrix of p-values in case they want to do something else
…   with the formatting
735     newHeatFrame = pd.DataFrame(newHeatMatrix)
736     newHeatFrame.columns = gwasNames
737     if len(dataFrameList) > 1:
738         newHeatFrame.index = reorderCellNames
739     else:
740         newHeatFrame.index = cellNames
741 
742     newHeatFrame.to_csv(outputLocation + '/pvalue_matrix.txt',sep='\t')
743 
744 
745     ##save overlap Counts
746     newOverlapMatrix = overlapMatrix
747 
748     ## save the matrix of overlaps count in case they want to do
…   something else with the formatting
749     newOverlapFrame = pd.DataFrame(newOverlapMatrix)
750     newOverlapFrame.columns = gwasNames
751     if len(dataFrameList) > 1:
752         newOverlapFrame.index = reorderCellNames
753     else:
```

```
        newOverlapFrame.index = cellNames

    ## save the matrix of overlap counts in case this information is
desired
    newOverlapFrame.to_csv(outputLocation +
'/overlap_count_matrix.txt',sep='\t')



## Sinib Package Functions
## translated from R to Python from: https://github.com/boxiangliu/sinib

def q(u,p):
    return([(i * math.exp(u)) / (1 - i + i * math.exp(u)) for i in p])

def K(u,n,p):
    return(sum([a*math.log(1-b+b*math.exp(u)) for a,b in zip(n,p)]))

def Kp(u,n,p):
    return(sum([a*b for a,b in zip(n,q(u,p))]))

def Kpp(u,n,p):
    q2=q(u,p)
    return(sum([a*b*(1-b) for a,b in zip(n,q2)]))

def Kppp(u,n,p):
    q2=q(u,p)
    return(sum([a*b*(1-b)*(1-2*b) for a,b in zip(n,q2)]))

def Kpppp(u,n,p):
    q2=q(u,p)
    return(sum([a*b*(1-b)*(1-6*b*(1-b)) for a,b in zip(n,q2)]))

def saddlepoint(u,n,p,s):
    return(Kp(u,n,p)-s)

def w(u,n,p):
    K2=K(u,n,p)
    Kp2=Kp(u,n,p)
    return(np.sign(u)*math.sqrt(2*u*Kp2-2*K2))

def u1(u,n,p):
    Kpp2=Kpp(u,n,p)
    return((1-math.exp(-u))*math.sqrt(Kpp2))

def p3(u,n,p,s,mu):
```

```
    w2=w(u,n,p)
    u12=u1(u,n,p)
    if ((u12==0.0) or (s==mu)):
        Kpp0=sum([a*b*(1-b) for a,b in zip(n,p)])
        Kppp0=sum([a*b*(1-b)*(1-2*b) for a,b in zip(n,p)])

return(1/2-(2*np.pi)**(-1/2)*((1/6)*Kppp0*Kpp0**(-3/2)-(1/2)*Kpp0**(-1/2)
))

    else:
        return(1-norm.cdf(w2)-norm.pdf(w2)*(1/w2-1/u12))


def u2(u,n,p):
    return(u*math.sqrt(Kpp(u,n,p)))

def k3(u,n,p):
    return(Kppp(u,n,p)*Kpp(u,n,p)**(-3/2))

def k4(u,n,p):
    return(Kpppp(u,n,p)*Kpp(u,n,p)**(-2))

def p4(u,n,p,s,mu):

    u2b=u2(u,n,p)
    k3b=k3(u,n,p)
    k4b=k4(u,n,p)
    wb=w(u,n,p)
    p3b=p3(u,n,p,s,mu)
    u1b=u1(u,n,p)
    if ((u1b==0) or (s==mu)):
        return(p3b)
    else:

return(p3b-norm.pdf(wb)*((1/u2b)*((1/8)*k4b-(5/24)*k3b**2)-1/(u2b**3)-k3b
/(2*u2b**2)+1/wb**3))

def calc_p4(s, n, p, mu):
    if (s == sum(n)):
        p4b=numpy.prod([a**b for a,b in zip(p,n)])
    else:
        test = root(lambda x: saddlepoint(x,n,p,s), 0)
        u_hat = test.x[0]
        if not test.success:
            n_trial=100000
            n_binom=len(p)
```

```
            mat = np.random.binomial(n, p, (n_trial, n_binom)).T

            S = np.sum(mat,axis=0)
            counterTemp = 0
            s = [s]*len(S)
            for l1,l2 in zip(S,s):
                if l1 >= l2:
                    counterTemp = counterTemp + 1
            p4_ = counterTemp/len(S)
        else:
            p4_=p4(u_hat,n,p,s,mu)

    return(p4_)

def psinib(q,size,prob,lowerTail=True,logP=False):

    if(not all(isinstance(x, int) for x in size)):
        sys.exit('Non-integer values were used for size values given to
psinib')
    if not all(isinstance(x, int) for x in size):
        n = [int(i) for i in size]
    else:
        n = [int(i) for i in size]

    p = [float(i) for i in prob]
    s = int(q)

    mu = sum([a*b for a,b in zip(n,p)])

    p4b = calc_p4(s+1,n,p, mu)

    if(lowerTail):
        if (logP):
            return(np.log(1-p4b))
        else:
            return(1-p4b)

    else:
        if(logP):
            return(np.log(p4b))
        else:
            return(p4b)

## Main Function which parses arguments and returns values

def main():
```

```
885
886     ## parse the given arguments to see if the neccesary libraries have
    … been given and
887     parser = argparse.ArgumentParser(prog ='gaia',
888                                      description ='Compare ATAC-seq data
    … to loci.')
889
890     parser.add_argument('-a', '--cellRegion', required = True,
891                         help='the folder location of the cell region bed
    … files stored in .txt format')
892     parser.add_argument('-g', '--loci', required = True,
893                         help='the folder location of the loci files
    … stored in .tsv format')
894     parser.add_argument('-c', '--chrom', required = True,
895                         help='the location of the chromsome size file
    … stored in a .csv format')
896     parser.add_argument('-o', '--output', required = True,
897                         help='the folder location you want the results to
    … be output into')
898     parser.add_argument('-u', '--peakUniqueness',  default=0, required =
    … False, type=int,
899                         help='a cutoff value for ATAC uniqueness (e.g. if
    … given 12, then any atac peak found in more than 12 atac sets will be
    … removed from all of them) - by default uniqueness is not considered')
900     parser.add_argument('-l', '--lociCutoff',default=0, required = False,
    … type=int,
901                         help='a loci cutoff value, will only consider
    … loci groups (phenotypes or cohorts) with more loci than this cutoff value
    … - by default this cutoff is 0')
902     parser.add_argument('-s', '--specificLoci',default=0, required =
    … False,
903                         help='a txt file with the specific loci phenotype
    … you would like to use. This can be very helpful if using a large loci set
    … with with many phenotypes, and you want to sort by more than just loci
    … count.' )
904     parser.add_argument('-m', '--maskRegion',default=0, required = False,
905                         help='a txt file containing a set of regions that
    … you want to subset each ATAC region by, for example a set of regions
    … around the TSSs of genes of interest. This will reduce the ATAC regions
    … to just those that overlap with this set of regions' )
906
907     parser.add_argument('-w', '--windowSize',  default=100000, required =
    … False, type=int,
908                         help='Size of the window you would like to create
    … around each loci to make the statistical comparison for enrichment')
909
```

```
910
911     args = parser.parse_args()
912
913     gaiaAssociation(args.cellRegion, args.loci, args.chrom, args.output,
  …  args.peakUniqueness, args.lociCutoff, args.specificLoci, args.maskRegion,
  …  args.windowSize)
914
```