How to run the code

1. Unzip and put all the files in one folder
2. Download Jupyter Notebook
3. Navigate to the folder with the files, open the command prompt on that folder, and type “jupyter notebook” in the command line to open the folder in Jupyter Notebook
4. Open the ipynb files in Jupyter Notebook and run the code

Implementation Code

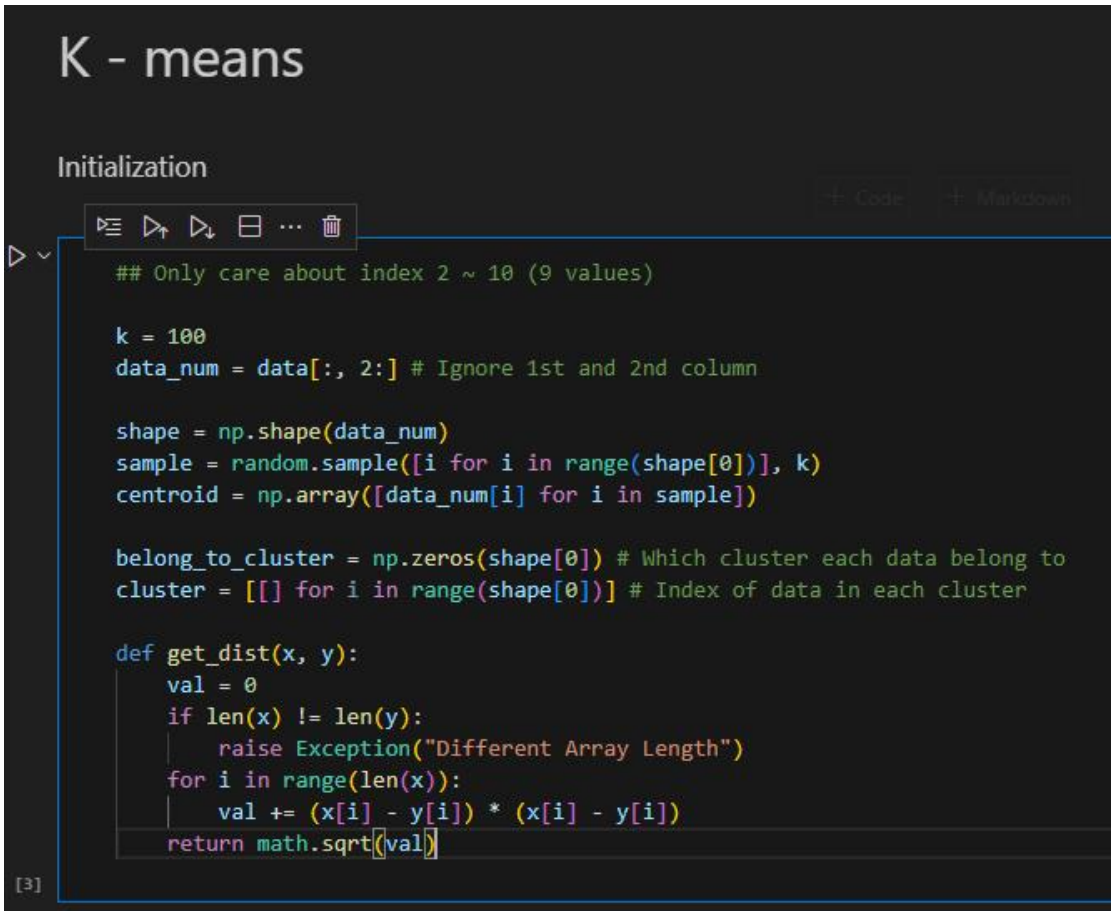
First, the k-means clustering is done on clustering.ipynb
Open the tsv files and get the data from it. We deal with missing values by deleting rows that have at least one missing values using dropna().

```
[2]
import numpy as np
import pandas as pd
import random
import math

#f = open("E-MTAB-8626_tpms.tsv", "r")
#print(f.read())

df = pd.read_csv('E-MTAB-8626_tpms.tsv', sep='\t', skiprows=4)
df = df.dropna() # Drop missing values
#print(df)
data = df.to_numpy()
print(data)
```

```
[['ETS1-1' 'ETS1-1' 0.3 ... 0.4 2.0 0.3]
 ['HRA1' 'HRA1' 2.0 ... 2.0 1.0 2.0]
 ['ICR1' 'ICR1' 5.0 ... 4.0 4.0 4.0]
 ...
 ['YPR201W' 'ARR3' 10.0 ... 7.0 9.0 9.0]
 ['YPR202W' 'YPR202W' 2.0 ... 2.0 2.0 2.0]
 ['YPR204W' 'YPR204W' 3.0 ... 3.0 2.0 2.0]]
```

The variable k indicate the number of clusters we want for the k-means. Since we only care about the numerical values for the k-means, we ignore the first two columns of our data.
First, we initialize the centroids by sampling k random points without replacement from the data, and have the points be the starting points for the centroids.

The variable belong_to_cluster indicates which cluster each data point belongs to, while the variable cluster indicates the index of data points that is in the particular cluster.
The function get_dist() returns the distance between two points.



The for loop here assign each point to the cluster which centroid has the smallest distance to the point.

```
for i in range(shape[0]):
    min_cluster = 0
    min_dist = get_dist(data_num[i], centroid[0])
    for j in range(1, k):
        cur_dist = get_dist(data_num[i], centroid[j])
        if cur_dist < min_dist:
            min_cluster = j
            min_dist = cur_dist
    if belong_to_cluster[i] != int(min_cluster):
        did_cluster_update = True
        up += 1
    belong_to_cluster[i] = int(min_cluster)
    cluster[min_cluster].append(i)
```

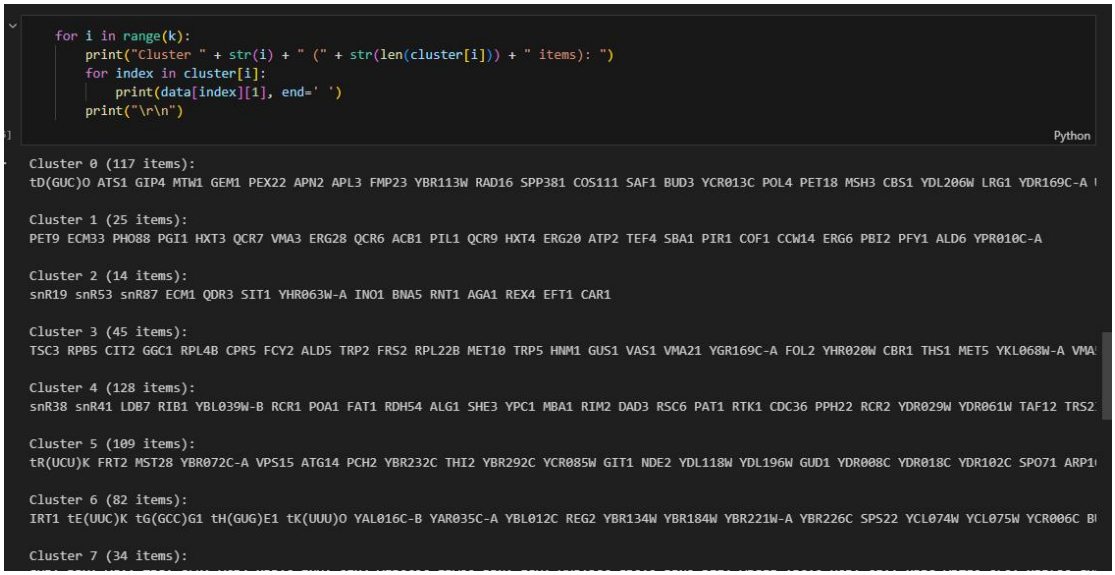
After updating the cluster each points belong to, we assign a new location for the centroid in each cluster to the average value in each axis of all points in the cluster that the centroid represents.

```
## Update centroid
for i in range(k):
    if len(cluster[i]) == 0:
        continue
    new_cent = np.zeros(9)
    for index in cluster[i]:
        new_cent = new_cent + data_num[index]
    new_cent = new_cent / len(cluster[i])
    centroid[i] = new_cent
```

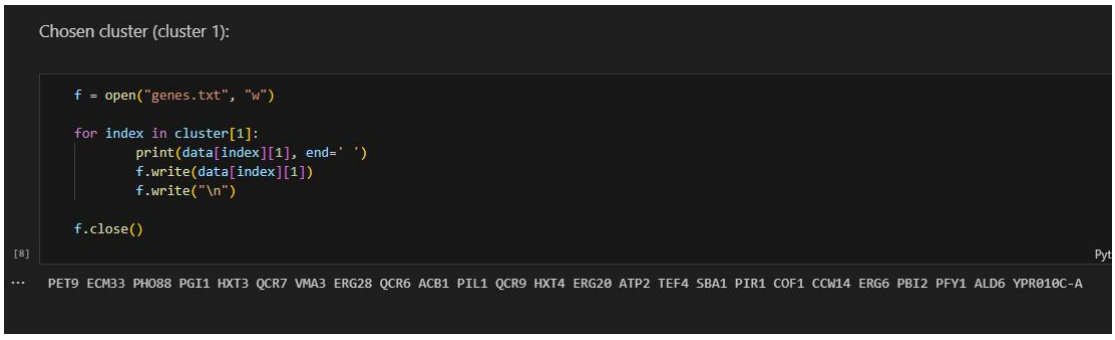
The process of alternating between updating the cluster each points belong to and updating the centroids' location keeps going on until each point doesn't get assigned into a new cluster anymore or the number of iterations exceed 750.

```
if did_cluster_update == False or it >= 750:
    break
```

Then we obtain the name of the genes in each cluster.

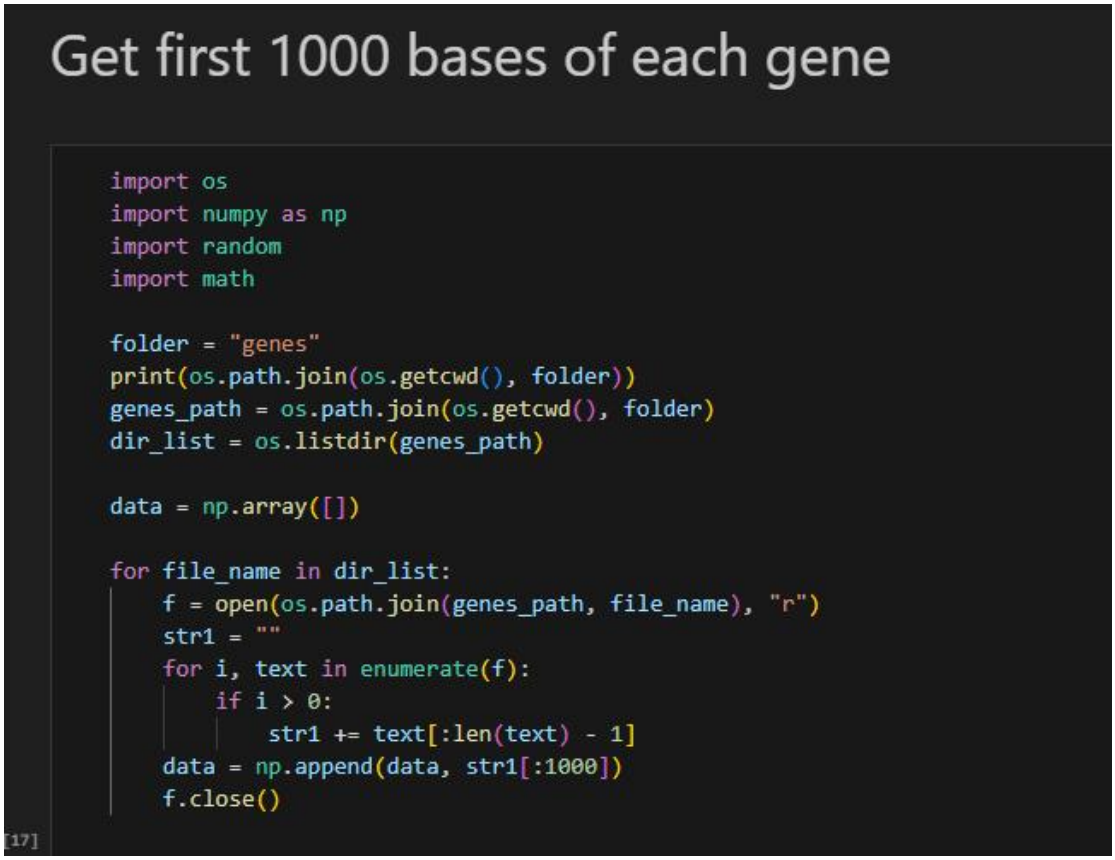


For this experiment I choose cluster 1 to analyze, and so I export the names of the genes in that cluster to the file “genes.txt”.



For the second part of the project, I did the motif finding using Gibbs Sampling on motif_finding.ipynb

First, the code opens every single file in the genes folder then extract the first kilobase of each sequence and store it in the variable “data”.



w here indicates the length of the motif we want to find and can be changed to any positive integer. num_genes indicate the number of sequences that we have (here we have 25). PWM indicates the Position Weight Matrix.

First we calculate the background probability of each base which is obtained from the total number of appearance of that particular base divided by the total number of bases across all sequences.

Randomly initialize position of motif in each gene

```
w = 15 # Length of motif
num_genes = len(data)
PWM = np.zeros((w, 4)) # A = 0, C = 1, G = 2, T = 3 (motif length x 4)

bg_prob = np.zeros(4)
count_A = count_C = count_G = count_T = 0
for gene in data:
    for base in gene:
        if base == 'A':
            count_A += 1
        elif base == 'C':
            count_C += 1
        elif base == 'G':
            count_G += 1
        elif base == 'T':
            count_T += 1
bg_prob[0] = count_A / (num_genes * 1000)
bg_prob[1] = count_C / (num_genes * 1000)
bg_prob[2] = count_G / (num_genes * 1000)
bg_prob[3] = count_T / (num_genes * 1000)
```

I initialized a random position for the motif in each sequence. start_pos here indicates the starting index of the motif in each gene. cur_motif directly store the actual motif from the genes based on the start_pos indexes.

```
start_pos = [random.randint(0, 999 - w + 1) for i in range(num_genes)] # Starting position of the motif in each gene (num_genes)

cur_motif = np.array([]) # Current motif based on start_pos (num_genes x motif length)
for i in range(num_genes):
    cur_motif = np.append(cur_motif, data[i][start_pos[i]:start_pos[i] + w])
```

cur_gene indicates the current gene that we want to find the motif for. I begin from the gene with index 0, then 1, and so on, then go back to 0 again after the last gene.

First I construct the PWM by considering the current motifs in each gene sequence except the cur_gene. Each entry in the PWM indicates the probability that the letter in the current position is the current base.

It is obtained by (Number of appearances that the current base have + 1) / (Number of gene sequences being considered + 4). I add 1 and 4 to the numerator and denominator here respectively as pseudocount to avoid zero probability.

```
cur_gene = it % num_genes

# Construct PWM without current gene
for i in range(w):
    count_A = count_C = count_G = count_T = 0
    for j in range(num_genes):
        if j == cur_gene:
            continue
        if cur_motif[j][i] == 'A':
            count_A += 1
        elif cur_motif[j][i] == 'C':
            count_C += 1
        elif cur_motif[j][i] == 'G':
            count_G += 1
        elif cur_motif[j][i] == 'T':
            count_T += 1
    PWM[i][0] = (count_A + 1) / (num_genes - 1 + 4)
    PWM[i][1] = (count_C + 1) / (num_genes - 1 + 4)
    PWM[i][2] = (count_G + 1) / (num_genes - 1 + 4)
    PWM[i][3] = (count_T + 1) / (num_genes - 1 + 4)
```

Then I consider all possible motifs in the cur_gene, and get the position and motif with the best log score. The score is calculated using the formula below which is (probability current motif appears based on the PWM) / (probability current motif appears across all genomic sequence). To avoid excessive multiplications, we take the logarithm of that score.

$$S_x = \prod_{i=1}^{w} A_i = \prod_{i=1}^{w} \frac{Q_i}{P_i} = \prod_{i=1}^{w} \prod_{j=1}^{4} \left(\frac{q_{i,j}}{p_{i,j}}\right)^{c_{i,j}} \qquad F = \sum_{i=1}^{W} \sum_{j=1}^{4} c_{i,j} \log \frac{q_{i,j}}{p_j}$$

where S_x : score of motif x
W: width of motif
$c_{i,j}$: the count of nucleic base j in position i
$q_{i,j}$: the probability of nucleic base j in position i
$p_{i,j}$: the background probability of nucleic base j in position i
p_j : the background probability of nucleic base j, which is equal to $p_{i,j}$

After I find the motif with the best score for the particular sequence, I move on to the next one, and so on. The process of finding the best motif in each sequence continues on until the position of the motif doesn't change num_genes times, which means that the algorithm has reached convergence and I end the loop.

```
# Check score for each possible motif position, get max score
for i in range(1000 - w + 1):
    logscore = 0
    for j in range(w):
        if data[cur_gene][i + j] == 'A':
            logscore += math.log(PWM[j][0]/bg_prob[0])
        elif data[cur_gene][i + j] == 'C':
            logscore += math.log(PWM[j][1]/bg_prob[1])
        elif data[cur_gene][i + j] == 'G':
            logscore += math.log(PWM[j][2]/bg_prob[2])
        elif data[cur_gene][i + j] == 'T':
            logscore += math.log(PWM[j][3]/bg_prob[3])
    if logscore > max_score or i == 0:
        max_score = logscore
        best_pos = i


if start_pos[cur_gene] == best_pos: # If the motif position didnt change num_genes times in a row, end loop
    times_didnt_change += 1
else:
    times_didnt_change = 0

if times_didnt_change >= num_genes:
    break

# Update start position and motif
start_pos[cur_gene] = best_pos
cur_motif[cur_gene] = data[cur_gene][best_pos:best_pos + w]
```

To find the actual motif, we first need to calculate the PWM again, this time considering all the gene sequences we have. The motif is obtained by selecting the base with the highest probability in each position.

```
# Calculate PWM
for i in range(w):
    count_A = count_C = count_G = count_T = 0
    for j in range(num_genes):
        if cur_motif[j][i] == 'A':
            count_A += 1
        elif cur_motif[j][i] == 'C':
            count_C += 1
        elif cur_motif[j][i] == 'G':
            count_G += 1
        elif cur_motif[j][i] == 'T':
            count_T += 1
    PWM[i][0] = (count_A + 1) / (num_genes + 4)
    PWM[i][1] = (count_C + 1) / (num_genes + 4)
    PWM[i][2] = (count_G + 1) / (num_genes + 4)
    PWM[i][3] = (count_T + 1) / (num_genes + 4)

print(PWM)
output_motif = ''
for i in range(w):
    index = np.argmax(PWM[i])
    if index == 0:
        output_motif += 'A'
    elif index == 1:
        output_motif += 'C'
    elif index == 2:
        output_motif += 'G'
    elif index == 3:
        output_motif += 'T'
print(output_motif)
```

Experiments and Result

First I run the k-means algorithm on the dataset by running clustering.ipynb with k = 100:

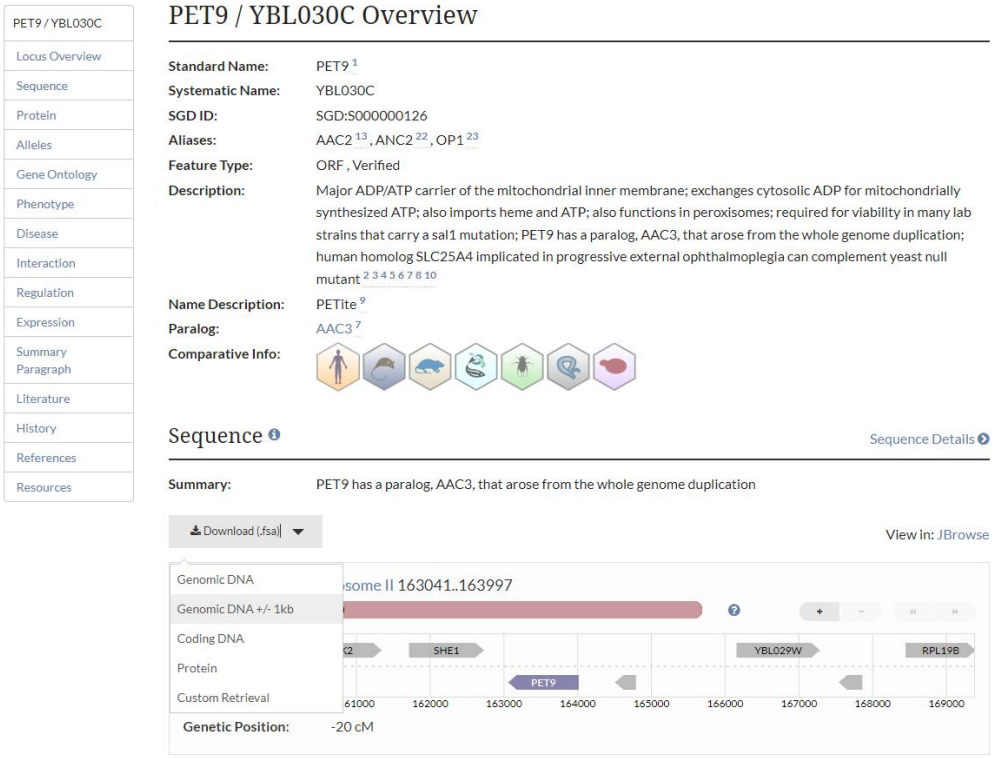
```
Cluster 0 (117 items):
tD(GUC)O ATS1 GIP4 MTW1 GEM1 PEX22 APN2 APL3 FMP23 YBR113W RAD16 SPP381 COS111 SAF1 BUD3 YCR013C POL4 PET18 MSH3 CBS1 YDL206W LRG1 YDR169C-A

Cluster 1 (25 items):
PET9 ECM33 PHO88 PGI1 HXT3 QCR7 VMA3 ERG28 QCR6 ACB1 PIL1 QCR9 HXT4 ERG20 ATP2 TEF4 SBA1 PIR1 COF1 CCW14 ERG6 PBI2 PFY1 ALD6 YPR010C-A

Cluster 2 (14 items):
snR19 snR53 snR87 ECM1 QDR3 SIT1 YHR063W-A INO1 BNA5 RNT1 AGA1 REX4 EFT1 CAR1

Cluster 3 (45 items):
TSC3 RPB5 CIT2 GGC1 RPL4B CPR5 FCY2 ALD5 TRP2 FRS2 RPL22B MET10 TRP5 HNM1 GUS1 VAS1 VMA21 YGR169C-A FOL2 YHR020W CBR1 THS1 MET5 YKL068W-A VMA

Cluster 4 (128 items):
snR38 snR41 LDB7 RIB1 YBL039W-B RCR1 POA1 FAT1 RDH54 ALG1 SHE3 YPC1 MBA1 RIM2 DAD3 RSC6 PAT1 RTK1 CDC36 PPH22 RCR2 YDR029W YDR061W TAF12 TRS2

Cluster 5 (109 items):
tR(UCU)K FRT2 MST28 YBR072C-A VPS15 ATG14 PCH2 YBR232C THI2 YBR292C YCR085W GIT1 NDE2 YDL118W YDL196W GUD1 YDR008C YDR018C YDR102C SPO71 ARP1

Cluster 6 (82 items):
IRT1 tE(UUC)K tG(GCC)G1 tH(GUG)E1 tK(UUU)O YAL016C-B YAR035C-A YBL012C REG2 YBR134W YBR184W YBR221W-A YBR226C SPS22 YCL074W YCL075W YCR006C B

Cluster 7 (34 items):
SHP1 PRX1 YSA1 TPS1 GLK1 YCP4 MRP10 INH1 GIM4 YER062C ERV29 RPN1 FSH1 YHR138C SPO12 RPN2 REE1 VPS55 ARC19 UGP1 GFA1 MRP8 YPT52 GLO1 MRPL39 SU

Cluster 8 (124 items):
...

Cluster 99 (8 items):
RPL41B RPL41A TDH3 ENO2 FBA1 PDC1 CCW12 RPS31
```

I picked a cluster with decent size to analyze, here I pick cluster 1 with a size of 25 genes

```
PET9 ECM33 PHO88 PGI1 HXT3 QCR7 VMA3 ERG28 QCR6 ACB1 PIL1 QCR9 HXT4 ERG20 ATP2 TEF4 SBA1 PIR1 COF1 CCW14 ERG6 PBI2 PFY1 ALD6 YPR010C-A
```

For each gene in the cluster, I downloaded the sequences from https://www.yeastgenome.org/

This process is done manually 25 times, and all the files are stored in the genes folder.



I run the motif finding algorithm using Gibbs Sampling by running motif_finding.ipynb . For this example, I use motif width w = 15 and get the sequence motif “TCCCCTGCGCATATT”

```
Iterations: 131
[[0.03448276 0.03448276 0.27586207 0.65517241]
 [0.20689655 0.4137931  0.13793103 0.24137931]
 [0.03448276 0.62068966 0.03448276 0.31034483]
 [0.03448276 0.75862069 0.03448276 0.17241379]
 [0.20689655 0.51724138 0.10344828 0.17241379]
 [0.06896552 0.10344828 0.06896552 0.75862069]
 [0.31034483 0.03448276 0.44827586 0.20689655]
 [0.06896552 0.86206897 0.03448276 0.03448276]
 [0.10344828 0.4137931  0.44827586 0.03448276]
 [0.24137931 0.65517241 0.06896552 0.03448276]
 [0.34482759 0.13793103 0.17241379 0.34482759]
 [0.06896552 0.17241379 0.10344828 0.65517241]
 [0.37931034 0.17241379 0.13793103 0.31034483]
 [0.03448276 0.17241379 0.06896552 0.72413793]
 [0.27586207 0.10344828 0.03448276 0.5862069 ]]
TCCCCTGCGCATATT
```

Then I use the online tool TOMTOM to validate and judge the quality of the motif that I found.

QUERY MOTIFS

Database	ID	Alt. ID	Preview	Matches
query_motifs	1	TCCCCTGCGCATATT		28

This process can be done several times with varying number of clusters, width of motif, and multiple instances to get different results.

Discussions

There are several challenges that I experienced through this project. First is determining the parameters of my algorithm, which is the amount of cluster k and the motif length w. I don't really know the amount of cluster and the length of the motif I'm supposed to find so it's assigned somewhat arbitrarily. Different assignment of the parameter values will produce different results.

Selecting which cluster to analyze is also selected based on a whim. I tried to select a cluster with a size that is not too big and not too small. However, other clusters may contain more significant information compared to the cluster I chose to analyze. There is also the issue that I chose to download the sequence of each gene in the cluster manually and not done automatically. This means that if I want to choose to analyze a different cluster, I need to do this again manually and if the cluster size is very big, it will become very tedious.

There is also a problem with Gibbs Sampling producing different results on each run even when the motif length that I chose to find is unchanged. It seems that the search space I'm looking for has many local maximas and the algorithm converges to them very easily. This means that the motif that the algorithm outputs is very dependant on the random initialization of the motif in each sequence.

On an example run demonstrated above, the algorithm finds the motif TCCCCTGCGCATATT of length 15. Using the validation tool TOMTOM, it finds 28 out of 732 matches for this motif, which may or may not be a statistically significant number.