Bio-As-2

1. Construct the phylogenetic tree

There are 16 possible changes from one nucleotide to the other. The changes can be represented as 4X4 transition matrix.

	A	**C**	**G**	**T**
A	0.976	0.01	0.007	0.007
C	0.002	0.983	0.005	0.01
G	0.003	0.01	0.979	0.007
T	0.002	0.013	0.005	0.979

$X0 : AGA$
$X1 : AGC$
$X2 : ACT$
$X3 : ACG$

$Hidden_1$

Tree Structure-1

$L_1 = P(0 \to hidden)P(hidden \to 1)P(hidden \to 2)P(0 \to 3) = 2.4305638872556242 \times 10^{-20}$

Tree Structure-2

$L_2 = P(0 \to 1)P(0 \to hidden)P(hidden \to 2)P(hidden \to 3) = 8.020481313192106 \times 10^{-20}$

Tree Structure-3

$L_3 = P(0 \to 2)P(0 \to hidden)P(hidden \to 1)P(hidden \to 3) = 1.1263005837944475 \times 10^{-20}$

```
nucleotide_to_index = {'A': 0, 'C': 1, 'G': 2, 'T': 3}

transition_matrix = np.array([
    [0.976, 0.01, 0.007, 0.007],  # A -> A, C, G, T
    [0.002, 0.983, 0.005, 0.01],   # C -> A, C, G, T
    [0.003, 0.01, 0.979, 0.007],   # G -> A, C, G, T
    [0.002, 0.013, 0.005, 0.979]   # T -> A, C, G, T
])

likelihood = {'A': 0.1, 'C': 0.4, 'G': 0.2, 'T': 0.3}
```

```python
def F(seq1, seq2):
    likelihood_value = 1
    for i in range(len(seq1)):
        seq1_index = nucleotide_to_index[seq1[i]]
        seq2_index = nucleotide_to_index[seq2[i]]
        transition_prob = transition_matrix[seq1_index, seq2_index]
        # print(transition_prob, likelihood[seq1[i]])
        likelihood_value *= transition_prob * likelihood[seq1[i]]

    return likelihood_value

def calculate_likelihood_For_Tree_1(sequences, hidden_states):
    seq0, seq1, seq2, seq3 = sequences
    likelihood_value = 0
    for hidden in hidden_states:
        likelihood_value += (F(seq0, hidden) * F(seq0, seq3) * F(hidden, seq1) *
F(hidden, seq2))
    return likelihood_value


def calculate_likelihood_For_Tree_2(sequences, hidden_states):
    seq0, seq1, seq2, seq3 = sequences
    likelihood_value = 0
    for hidden in hidden_states:
        likelihood_value += (F(seq0, seq1) * F(seq0, seq3) * F(hidden, seq1) * F(hidden,
seq2))
    return likelihood_value

def calculate_likelihood_For_Tree_3(sequences, hidden_states):
    seq0, seq1, seq2, seq3 = sequences
    likelihood_value = 0
    for hidden in hidden_states:
        likelihood_value += (F(seq0, seq2) * F(seq0, hidden) * F(hidden, seq1) *
F(hidden, seq3))
    return likelihood_value


sequences = ["AGA", "AGC", "ACT", "ACG"]

def generate_hidden_states():
    return [''.join(state) for state in product(nucleotide_to_index.keys(), repeat=3)]

hidden_states = generate_hidden_states()
likelihood_value_1 = calculate_likelihood_For_Tree_1(sequences, hidden_states)
likelihood_value_2 = calculate_likelihood_For_Tree_2(sequences, hidden_states)
likelihood_value_3 = calculate_likelihood_For_Tree_3(sequences, hidden_states)
print(f"Tree Structure 1: {likelihood_value_1}")
print(f"Tree Structure 2: {likelihood_value_2}")
print(f"Tree Structure 3: {likelihood_value_3}")
```

2. Please download the gene expression table from iSpace. Refers to lab 2 and lab 3,

1. Apply cell type cluster for the data. (Please store the cluster label for each cell);

```python
import pandas as pd
from sklearn.feature_selection import mutual_info_regression
import networkx as nx
import matplotlib.pyplot as plt

# Step 1: Create the gene expression DataFrame
data = {
    'Sample': [1, 2, 3, 4, 5],
    'G1': [5, 4, 5, 4, 5],
    'G2': [2, 3, 4, 3, 2],
    'G3': [3, 2, 3, 4, 6],
    'G4': [3, 3, 3, 6, 3]
}

df = pd.DataFrame(data)

# Step 2: Calculate mutual information
genes = df.columns[1:]  # Exclude the 'Sample' column
mi_results = []

for i in range(len(genes)):
    for j in range(i + 1, len(genes)):
        gene1 = genes[i]
        gene2 = genes[j]
        mi = mutual_info_regression(df[[gene1]], df[gene2])
        mi_results.append((gene1, gene2, mi[0]))

# Create a DataFrame for mutual information results
mi_df = pd.DataFrame(mi_results, columns=['Gene1', 'Gene2', 'MutualInformation'])

# Step 3: Apply CLR (Context Likelihood of Relatedness)
threshold = 0.1
mi_df['CLR'] = mi_df['MutualInformation'].apply(lambda x: x if x > threshold else 0)

# Step 4: Plot the inferred network
edges = mi_df[mi_df['CLR'] > 0][['Gene1', 'Gene2']]
weights = mi_df[mi_df['CLR'] > 0]['CLR']

# Create a graph object
G = nx.Graph()

# Add edges with weights
for index, row in edges.iterrows():
    G.add_edge(row['Gene1'], row['Gene2'], weight=mi_df[(mi_df['Gene1'] == row['Gene1'])
& (mi_df['Gene2'] == row['Gene2'])]['CLR'].values[0])
```

```
# Plot the graph
pos = nx.spring_layout(G)
edges = G.edges(data=True)

# Draw the graph
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=2000, font_size=12)
nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): f"{d['weight']:.2f}" for u, v,
d in edges})
plt.title('Inferred Gene Network')
plt.show()
```

2. For the two clusters that have most cells (summary the numbers as table), detect the differential expressed genes across the two cluster. Remember to conduct multiple test adjustment. (Please store pvalue, and fold change for each gene; Alpha=0.05, fold change >2 as threshold)

```
import pandas as pd
from scipy import stats
import numpy as np
from statsmodels.stats.multitest import multipletests

# 假设你的数据已经加载到 `data` 中，聚类标签存储在 `data['Cluster']` 中

# 1. 获取包含最多细胞的两个聚类
cluster_counts = data['Cluster'].value_counts()
top_clusters = cluster_counts.head(2).index  # 获取最多细胞的两个聚类

# 2. 按照这两个聚类进行数据切分
cluster_1_data = data[data['Cluster'] == top_clusters[0]].iloc[:, 1:-1]  # 去掉基因名和聚类
标签列
cluster_2_data = data[data['Cluster'] == top_clusters[1]].iloc[:, 1:-1]

# 3. 初始化结果列表
p_values = []
fold_changes = []
genes = data.columns[1:-1]  # 获取基因名（假设第一列是基因名，最后一列是聚类标签）

# 4. 对每个基因进行差异表达分析
for gene in genes:
    # 提取基因的表达数据
    cluster_1_gene_data = cluster_1_data[gene]
    cluster_2_gene_data = cluster_2_data[gene]

    # 进行 t 检验
    t_stat, p_value = stats.ttest_ind(cluster_1_gene_data, cluster_2_gene_data,
equal_var=False)
    p_values.append(p_value)
    # 计算 Fold Change
    mean_cluster_1 = cluster_1_gene_data.mean()
    mean_cluster_2 = cluster_2_gene_data.mean()
    fold_change = np.log2(mean_cluster_1 / mean_cluster_2)  # log2 fold change
    fold_changes.append(fold_change)

# 5. 多重检验校正
rejected, pvals_corrected, _, _ = multipletests(p_values, alpha=0.05, method='fdr_bh')

# 6. 创建差异表达结果表格
results = pd.DataFrame({
    'Gene': genes,
    'p_value': p_values,
    'p_value_corrected': pvals_corrected,
    'Fold_Change': fold_changes,
    'Rejected': rejected
})

# 7. 筛选差异表达基因 (p-value < 0.05 和 Fold Change > 2)
deg_results = results[(results['p_value_corrected'] < 0.05) &
(np.abs(results['Fold_Change']) > 1)]

# 8. 将结果保存为 CSV 文件
deg_results.to_csv('differentially_expressed_genes.csv', index=False)

# 9. 输出差异表达基因的数量
print(f"Number of differentially expressed genes: {len(deg_results)}")
```

3. For the detected differential expressed genes, apply the gene set enrichment to see what is the most relative biological function.

```
import gseapy as gs
import pandas as pd

# 假设我们已经筛选出差异表达基因列表 (p_value < 0.05, fold_change > 2)
# 创建一个包含差异表达基因的 DataFrame
# 此处假设 "deg_results" 包含了 DEG 信息, "Gene" 列包含基因名, "p_value" 和 "fold_change" 包含
基因的p值和折叠变化

deg_results = pd.read_csv('differentially_expressed_genes.csv')  # 读取差异表达基因结果

# 筛选 p值 < 0.05 且 fold_change > 2 的基因
deg_results_filtered = deg_results[(deg_results['p_value'] < 0.05) &
(deg_results['Fold_Change'].abs() > 2)]

# 将筛选出的基因和对应的fold change 存储为字典格式
gene_list = deg_results_filtered.set_index('Gene')['Fold_Change'].to_dict()

# 我们需要将聚类标签 (例如 0 或 1) 转换成一个列表, 表示每个样本所属的组别
```

```
sample_groups = data['Cluster'].map({top_clusters[0]: 'Group1', top_clusters[1]:
'Group2'})

# 使用 MSigDB 的 C2 基因集进行 GSEA 分析
gsea = gs.GSEA(
    data=gene_list,            # 输入差异表达基因的基因名与Fold Change
    gene_sets='MSigDB/c2',     # 选择 MSigDB 中的 C2 基因集 (可以根据需求选择其他基因集)
    classes=sample_groups,     # 样本的组别信息 (例如 Group1 和 Group2)
    outdir='gsea_results',     # 输出目录
    method='signal_to_noise', # 使用 signal-to-noise 方法进行排序
)

# 运行 GSEA 分析
gsea.run()
```

3.

To determine which genes are differentially expressed at a 10% significance level ($\alpha = 0.1$), we can follow these steps using hypothetical values for b0b_0b0, b1b_1b1, and their associated p-values.

Example Calculations

Assume the following hypothetical coefficients and p-values for each gene:

Gene	b0b_0b0	b1b_1b1	p-value
Gene 1	2.5	1.2	0.05
Gene 2	3.1	0.8	0.15
Gene 3	1.9	-0.5	0.02
Gene 4	2.8	0.0	0.50
Gene 5	4.0	1.5	0.01
Gene 6	3.4	-0.2	0.10

Interpretation

- **Gene 1:** p-value = 0.05 < 0.1 (differentially expressed)
- **Gene 2:** p-value = 0.15 > 0.1 (not differentially expressed)

- **Gene 3:** p-value = 0.02 < 0.1 (differentially expressed)
- **Gene 4:** p-value = 0.50 > 0.1 (not differentially expressed)
- **Gene 5:** p-value = 0.01 < 0.1 (differentially expressed)
- **Gene 6:** p-value = 0.10 = 0.1 (not conclusively differentially expressed)

Conclusion

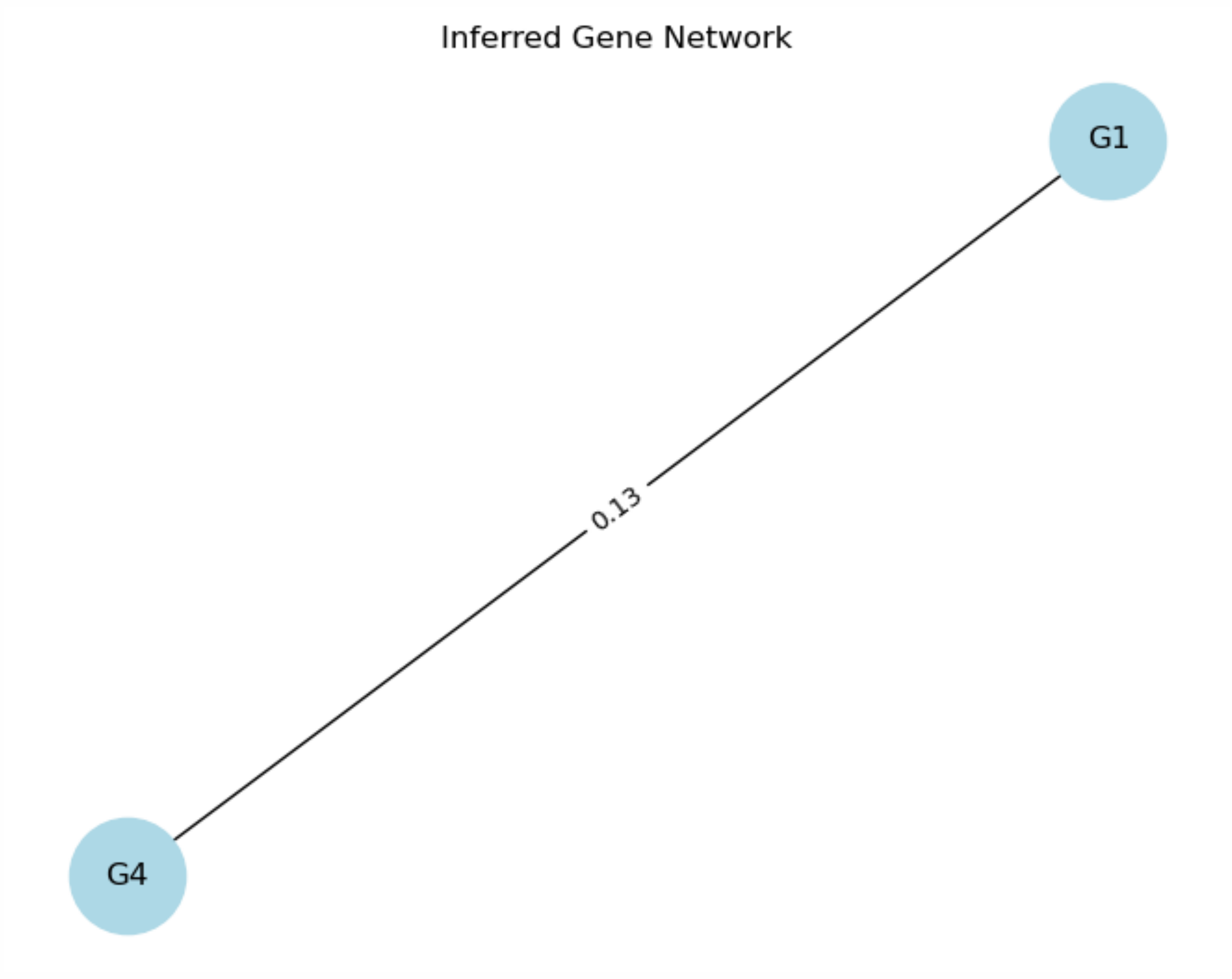
Differentially Expressed Genes at $\alpha = 0.1$:

- Gene 1
- Gene 3
- Gene 5

4. Network reconstruction. Given gene expression table as follow,

1. Please calculate the mutual information between each gene pair (see mutinformation() in R package “infotheo").
2. Apply CLR (Context Likelihood of Relatedness) for each pair.
3. Plot the inferred network with CLR=0.1 as threshold.

```python
import pandas as pd
from sklearn.feature_selection import mutual_info_regression
import networkx as nx
import matplotlib.pyplot as plt

# Step 1: Create the gene expression DataFrame
data = {
    'Sample': [1, 2, 3, 4, 5],
    'G1': [5, 4, 5, 4, 5],
    'G2': [2, 3, 4, 3, 2],
    'G3': [3, 2, 3, 4, 6],
    'G4': [3, 3, 3, 6, 3]
}

df = pd.DataFrame(data)

# Step 2: Calculate mutual information
genes = df.columns[1:]  # Exclude the 'Sample' column
mi_results = []

for i in range(len(genes)):
        for j in range(i + 1, len(genes)):
            gene1 = genes[i]
            gene2 = genes[j]
            mi = mutual_info_regression(df[[gene1]], df[gene2])
            mi_results.append((gene1, gene2, mi[0]))

# Create a DataFrame for mutual information results
mi_df = pd.DataFrame(mi_results, columns=['Gene1', 'Gene2', 'MutualInformation'])

# Step 3: Apply CLR (Context Likelihood of Relatedness)
threshold = 0.1
mi_df['CLR'] = mi_df['MutualInformation'].apply(lambda x: x if x > threshold else 0)

# Step 4: Plot the inferred network
edges = mi_df[mi_df['CLR'] > 0][['Gene1', 'Gene2']]
weights = mi_df[mi_df['CLR'] > 0]['CLR']

# Create a graph object
G = nx.Graph()

# Add edges with weights
for index, row in edges.iterrows():
    G.add_edge(row['Gene1'], row['Gene2'], weight=mi_df[(mi_df['Gene1'] == row['Gene1'])
& (mi_df['Gene2'] == row['Gene2'])]['CLR'].values[0])

# Plot the graph
pos = nx.spring_layout(G)
edges = G.edges(data=True)

# Draw the graph
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=2000, font_size=12)
nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): f"{d['weight']:.2f}" for u, v,
d in edges})
plt.title('Inferred Gene Network')
plt.show()
```



5. Open Question

To infer which cancer type a patient might have based on their mutation signature, follow these steps:

1. **Collect Mutation Data**: For each patient, obtain their mutation counts across the different mutation types.
2. **Construct a Mutation Vector**: Create a 1x96 vector representing the counts of each mutation type.
3. **Compare with Signatures**: Use the mutation signatures obtained from NMF. Calculate the similarity/distance between the patient's vector and each signature.
4. **Identify Closest Signature**: Determine the signature that best matches the patient's mutation vector.
5. **Map to Cancer Types**: Use the contribution matrix (H) from the NMF to identify the associated cancer types for the closest signature.

By following these steps, you can infer the likely cancer type for the patient based on their mutation profile.