Event Based Models for Disease Progression

Hongtao Hao & Joseph Austerweil

2024-10-13

Table of contents

1 Introduction

You can also read the material in PDF.

The Event-Based Model (EBM; Fonteijn et al. (2012)) is a probabilistic model that can be used to infer the order by which a disease affects the parts of a person's body. In other words, it allows us to estimate the stages by which different biological factors ("biomarkers") are affected by a disease.

For instance, Alzheimer's may have the following stages:

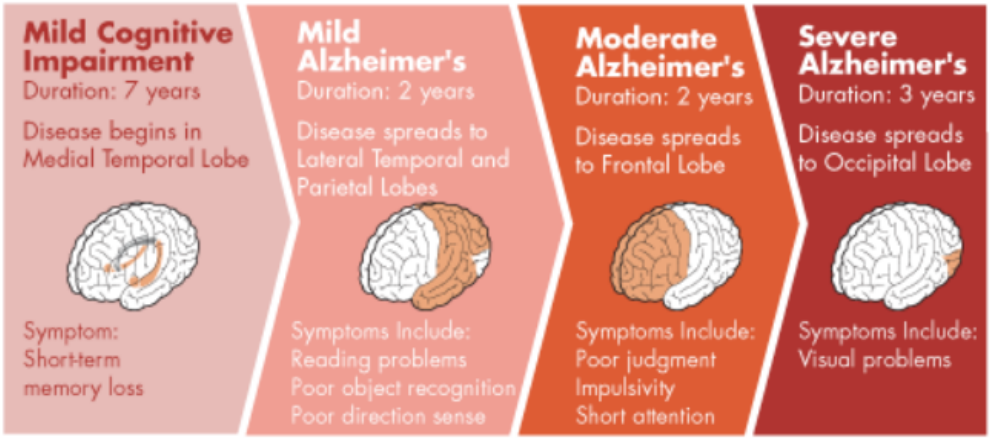


Figure 1.1: Alzheimer's Disease Progression (Credit: https://preventad.com/alzheimers-disease/)

We estimate this order based on the biomarker data from patients' visits. These data are typically results of neuropsych (e.g., MMSE) and/or biological examinations (e.g., blood pressure). Visits data can be longitudinal and/or cross-sectional, i.e., single visits from a cohort of patients.

Knowing the disease progression is important because it helps prevent and hopefully cure the disease (e.g., to identify critical points for intervention). It also helps health professionals prepare for the disease's further developments.

The EBM has been especially helpful at providing converging support for the stages of neural deterioration of neural degenerative diseases. Neural diseases are notoriously complex for many

reasons. For instance, they are difficult to study in vivo due to the challenge of direct and accurate measurement of the brain at high resolution without harming the person.

By formulating the deterioration process as a probabilistic model, the EBM affords the ability to conduct complex reasoning from noisy patient data. Given the difficulty of this task, it is unclear how much (number of participants) and what kind of data (healthy percentage) is needed for reliable estimation and how to best understand the uncertainty of model estimates.

Although prior work (e.g., Fonteijn et al. (2012), Chen et al. (2016)) has not addressed the former question in great detail, it has derived uncertainty of its estimates indirectly using bootstrapping. In this monograph, we analyze the statistical consistency of different inference methods for the EBM model. To foreshadow, our results are reassuring, yet troubling, and promising. Inference methods used in prior work work well when there is a large number of participants (~500) and the percentage of healthy participants is near 50%. However, clinical studies with such large sample sizes are uncommon and so, in typical settings, the prior approximation methods may be unreliable. We test our own method using Markov chain Monte Carlo techniques and find it provides much more accurate estimates for small sample sizes that are robust across different percentages of healthy participants. The results of our analyses come with a caveat: they are based on simulated data. We discuss this caveat, other limitations, and how to interpret our results in the monograph.

We have several assumptions in EBM:

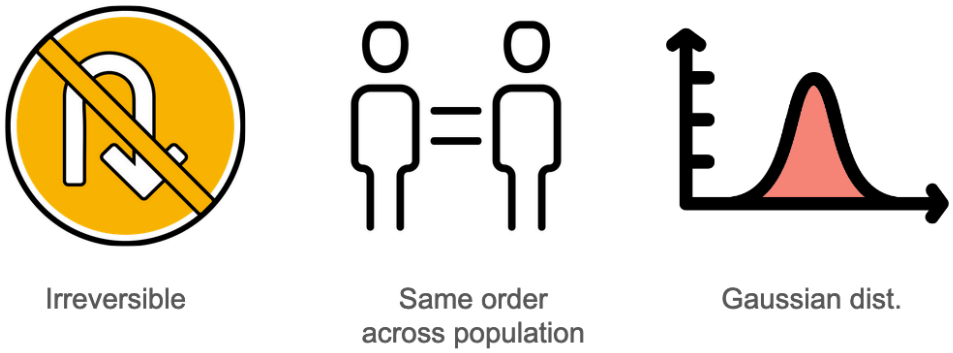


Figure 1.2: Assumptions of EBM

- The disease is irreversible (i.e., a patient cannot go from stage 2 to stage 1)
- The order in which different biomarkers get affected by the disease is the same across all patients.
- Biomarker data can be approximated by a Gaussian distribution.

Pay Attention

The third assumption, i.e., Gaussian approximation, often will be violated in raw biomarker data. For example, measurements of the concentration of amyloid proteins associated with Alzheimer's disease are necessarily non-negative. Further, their resolution has changed over the decades.
However, for the purpose of our current method, we assume this is true. There are nonparametric versions of the EBM, for example, KDE EBM

This book contains chapters that explain step by step how we use the event-based model to estimate the order of disease progression based on cross-sectional patients' biomarker data.

2 EBM Explained

2.1 Overview of Event-Based Model (EBM)

EBM provides a statistical model to understand disease progression through biomarkers. Using EBM, we can estimate the likelihood of biomarker measurements or generate synthetic data of biomarker measurements.

EBM can be used in two main ways:

1. Calculate the likelihood of biomarker measurements
2. Generate biomarker measurements

2.2 Key Concepts

Suppose the order in which a disease affects biomarkers is S. For example, $S =$ ['biomarker1', 'biomarker3', 'biomarker2'].

We also suppose biomarker measurements follow Gaussian distributions. When a biomarker is affected by the disease, its distribution parameters (mean and standard deviation) are denoted by θ. If not affected, ϕ.

The disease stage of a participant is k_j. To simplify things, let us assume the total number of disease stages is equal to the number of biomarkers.

2.3 Calculate the Likelihood of Biomarker Measurements

Suppose we have one participant's measurement data of five biomarkers:

Now, the question is:

> What is the likelihood of this participant having this sequence of biomarker data, given that we know S, θ, ϕ.

	participant	biomarker	measurement	k_j	S_n	affected_or_not	Diseased
0	67	HIP-FCI	22.478591	2	1	affected	True
1	67	HIP-GMI	-5.102497	2	3	not_affected	True
2	67	FUS-FCI	322.304772	2	5	not_affected	True
3	67	PCC-FCI	-5.690280	2	2	affected	True
4	67	FUS-GMI	6.345347	2	4	not_affected	True

Figure 2.1: Sample Data

As defined above, S is the order in which different biomarkers get affected by the disease. It is the column of S_n in the above data.

θ for each biomarker is the μ and α of normal distribution of biomarker measurement when the biomarker is affected by the disease.

ϕ for each biomarker is the μ and α of normal distribution of biomarker measurement when the biomarker is **NOT** affected by the disease.

The column of `participant` is simply this participant's identification in the data.

The column of `affected_or_not` refers to whether a biomarker is affected by the disease. It is affected if $k_j \geq S_n$; otherwise, not affected. This column is not available if we do not have access to the column of k_j, which stands for this participant's disease stage.

The column of `Diseased` refers to whether this participant is healthy (i.e., $k_j = 0$) or diseased (i.e., $k_j \geq 1$).

In the following, we explain how to calculate this likelihood in two scenarios: (1) known k_j and (2) unknown k_j.

2.3.1 Known k_j

$$p(X_j|S, z_j = 1, k_j) = \underbrace{\prod_{i=1}^{k_j} p(X_{S(i)j} \mid \theta_{S(i)})}_{\text{Affected biomarker likelihood}} \underbrace{\prod_{i=k_j+1}^{N} p(X_{S(i)j} \mid \phi_{S(i)})}_{\text{Non-affected biomarker likelihood}} \tag{2.1}$$

This equation computes the likelihood of the observed biomarker data of a specific participant, given that we know the disease stage this patient is at (k_j).

- S is an **ordered array** of biomarkers that are affected by the disease, for example, $[b, a, d, c]$. This means that biomarker b is affected at stage 1. At stage 2, biomarker b and a will be affected.

- $S(i)$ is the i^{th} biomarker according to S. For example S_1 will be biomarker b.
- k_j indicates the stage the patient is at, for example, $k_j = 2$. This means that the disease has affected biomarkers a and b. Biomarker c and d have not been affected yet.
- $\theta_{S(i)}$ is the parameters for the probability density function (PDF) of observed value of biomarker $S(i)$ when this biomarker has been affected by the disease. Let's assume this distribution is a Gaussian distribution with means of $[45, 50, 55, 60]$ and a standard deviation of 5 for biomarker b, a, d, and c.
- $\phi_{S(i)}$ is the parameters for the probability density function (PDF) of observed value of biomarker $S(i)$ when this biomarker has **NOT** been affected by the disease. Let's assume this distribution is a Gaussian distribution with means of $[25, 30, 35, 40]$ and a standard deviation of 3 for biomarker b, a, d, and c.
- X_j is an array representing the patient's observed data for all biomarkers. Assume the data is $[77, 45, 53, 90]$ for biomarkers b, a, d, and c.

We assume that the patient is at stage 2 of this disease; hence $k_j = 2$.

Next, we are going to calculate $p(X_j|S, z_j = 1, k_j)$:

When $i = 1$, we have $S_{(i)} = b$ and $X_{S_{(i)}} = X_b = 45$. So

$$p(X_{S_{(i)}}|\theta_{S(i)}) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{X_b - \mu}{\sigma}\right)^2}$$

Because $k_j = 2$, so biomarker b and a are affected. We should use the distribution of θ_b; therefore, we should plug in $\mu = 45, \sigma = 5$ in the above equation.

We can do the same for $i = 2$, 3, and 4.

So

$$p(X_j|S, k_j = 2) = p(X_b|\theta_b) \times p(X_a|\theta_a) \times p(X_d|\phi_d) \times p(X_c|\phi_c)$$

The above is the likelihood of the given biomarker data when $k_j = 2$.

Note that $p(X_b|\theta_b)$ is probability density, a value of a probability density function at a specific point; so it is not a probability itself.

Multiplying multiple probability densities will give us a likelihood.

2.3.2 Unknown k_j

$$P(X_j|S) = \sum_{k_j=0}^{N} P(k_j)p(X_j \mid S, k_j) \tag{2.2}$$

Suppose we have the same information above, except that we do not know at which disease stage the patient is, i.e., we do not know k_j. We have the observed biomarker data: $X_j = [77, 45, 53, 90]$. And I wonder: what is the likelihood of seeing this specific observed data?

We assume that all five stages (including $k_j = 0$) are equally likely.

We do not know k_j, so the best option is to calculate the "average" likelihood of all the biomarker data.

Based on Equation 2.1, we can calculate the following:

$L_1 = p(X_j|S, k_j = 1)$

$L_2 = p(X_j|S, k_j = 2)$

$L_3 = p(X_j|S, k_j = 3)$

$L_4 = p(X_j|S, k_j = 4)$

If this participant is healthy, then we know $k_j = 0$, therefore:

$$L = L_0 = p(X_j|S, k_j = 0) = p(X_b|\phi_b) \times p(X_a|\phi_a) \times p(X_d|\phi_d) \times p(X_c|\phi_c)$$

If this participant is diseased but we do not know the actual k_j, we can estimate it this way

$$L_1 = p(X_j|S, k_j = 1) = p(X_b|\theta_b) \times p(X_a|\phi_a) \times p(X_d|\phi_d) \times p(X_c|\phi_c)$$

$$L_2 = p(X_j|S, k_j = 2) = p(X_b|\theta_b) \times p(X_a|\theta_a) \times p(X_d|\phi_d) \times p(X_c|\phi_c)$$

$$L_3 = p(X_j|S, k_j = 3) = p(X_b|\theta_b) \times p(X_a|\theta_a) \times p(X_d|\theta_d) \times p(X_c|\phi_c)$$

$$L_4 = p(X_j|S, k_j = 4) = p(X_b|\theta_b) \times p(X_a|\theta_a) \times p(X_d|\theta_d) \times p(X_c|\theta_c)$$

$P(k_j)$ is the prior likelihood of being at stage k. **Event based models assume a uniform prior on** k_j. Therefore:

$P(X_j|z_j = 1, S) = \frac{1}{4}(L_1 + L_2 + L_3 + L_4)$

> **Tip**
>
> When this participant is diseased but we do not know the actual stage of this participant, the above method is useful also because it hints at the relative likelihood of each possible stage. For example, if L2 is much larger than L1, L3, and L4, then we know this participant is most likely to be at stage 2.

2.3.3 Extension

If we are more interested in the likelihood of a whole dataset consisting of all participants, we multiply all participants' likelihood: $L = L_{P_1} \times L_{P_2} \times L_{P_3} ... \times L_{P_j}$. Because this number tends to be very large, we take the natural log of L, i.e., $\ln(L)$.

2.4 EBM as A Generative Model

We can use EBM to generate synthetic biomarker data if we know:

- The order (S) in which different biomarkers get affected by the disease.
- Parameters (i.e., mean and standard deviation) of biomarkers' distribution when they are affected (θ) and not affected (ϕ) by the disease.
- Stages (k_j) that each participant is in.

Data we can generate looks like Figure 2.1.

This data is from a single participant.

As we mentioned above, to generate this data, we need to know:

- S, i.e., the order of biomarkers. In the above example, S is HIP-FCI, PCC-FCI, HIP-GMI, FUS-GMI, FUS-FCI.
- $\mathcal{N}(\theta_\mu, \theta_\sigma)$ and $\mathcal{N}(\phi_\mu, \phi_\sigma)$ for each of the five biomarkers, which are known but not shown directly here in the dataset.
- k_j, which is 2 in the above example.

We explain how this data is constructed in the following, column by column.

First, the `participant` id is 67. The `biomarker` indicates each of the five biomarkers examined and measured. The `measurement` is the biomarkers' measurement. `k_j` is the participant's stage. If this stage is above 0, it means `Diseased = True`. `S_n` indicates the n-th rank in the order. If `k_j < S_n`, it means the participant's stage hasn't reached that biomarker's rank; therefore, this biomarker is `not affected`. If `k_j >= S_n`, then this biomarker is `affected`.

If a biomarker is `affected`, then its measurement comes from $\mathcal{N}(\theta_\mu, \theta_\sigma)$ of that biomarker; if `not_affected`, $\mathcal{N}(\phi_\mu, \phi_\sigma)$.

2.4.1 Generative Process

The generative process of biomarker measurements can be described as:

$$\begin{aligned} X_{nj} \mid S, k_j, \theta_n, \phi_n \sim I(z_j = 1) \Big[& I(S(n) \leq k_j)\, p(X_{nj} \mid \theta_n) \\ & + I(S(n) > k_j)\, p(X_{nj} \mid \phi_n) \Big] \\ & + \big(1 - I(z_j = 1)\big)\, p(X_{nj} \mid \phi_n) \end{aligned} \tag{2.3}$$

This model says that given that we know $S, k_j, \theta_n,$ and ϕ_n, we can draw the biomarker measurement from a distribution.

$S \sim \text{UniformPermutation}(\cdot)$

S follows a distribution of uniform permutation. That means the ordering of biomarkers is random.

$k_j \sim \text{DiscreteUniform}(N)$

k_j follows a discrete uniform distribution, which means a participant is equally likely to fall in a progression stage (e.g., from 0 to 5, where 0 indicates this participant is healthy.)

2.4.2 Graphical Explanation

In the following, we explain the generative model in three different scenarios using graphical models: (1) All participants are healthy; (2) Both healthy and diseased participants, but all biomarkers are affected among diseased people; (3) Both healthy and diseased participants, but we do not whether biomarkers are affected or not among patients.

2.4.2.1 Scenario 1

If all participants are healthy:

$$X_{nj} \sim p(X_{nj} \mid \phi_n) \tag{2.4}$$

Where

X_{nj} indicates the measurement of biomarker n in participant j.

ϕ_n represents $\mathcal{N}(\phi_\mu, \phi_\sigma)$ for biomarker n.

The graphical model would look like:

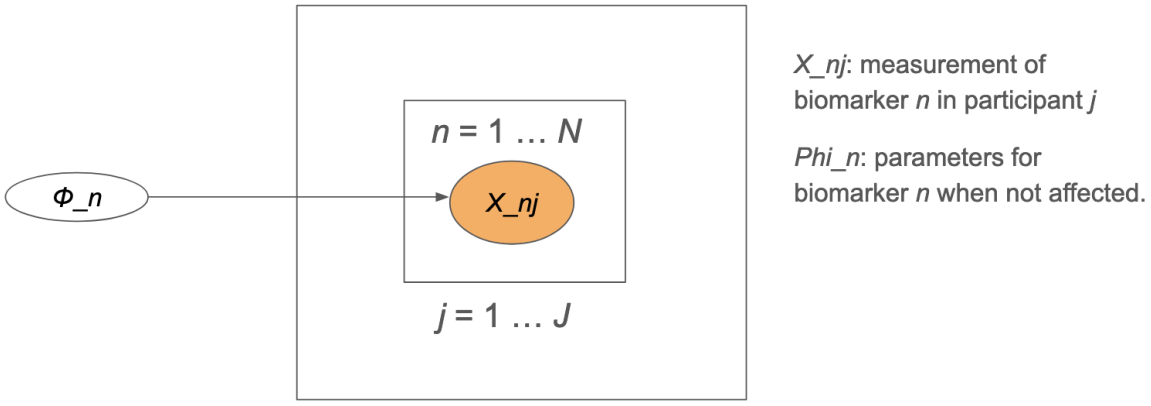


Figure 2.2: Graphical Model of Scenario 1

2.4.2.2 Scenario 2

If we have both diseased and healthy participants, and all biomarkers are affected among deceased participants.

$$X_{nj} \sim I(z_j == 1)p(X_{nj} \mid \theta_n) + (1 - I(z_j == 1))p(X_{nj} \mid \phi_n) \tag{2.5}$$

Where:

$z_j = 1$ indicates this participant is diseased and $z_j = 1$ represents a healthy participant.

$I(True) = 1$ and $I(False) = 0$.

θ_n represents $\mathcal{N}(\theta_\mu, \theta_\sigma)$ for biomarker n.

The graphical model would look like:

2.4.2.3 Scenario 3

If we have both healthy and diseased participants, but we do not know whether biomarkers are affected or not among patients, see Equation 2.3.

This is the model in usual cases.

The graphical model looks like:

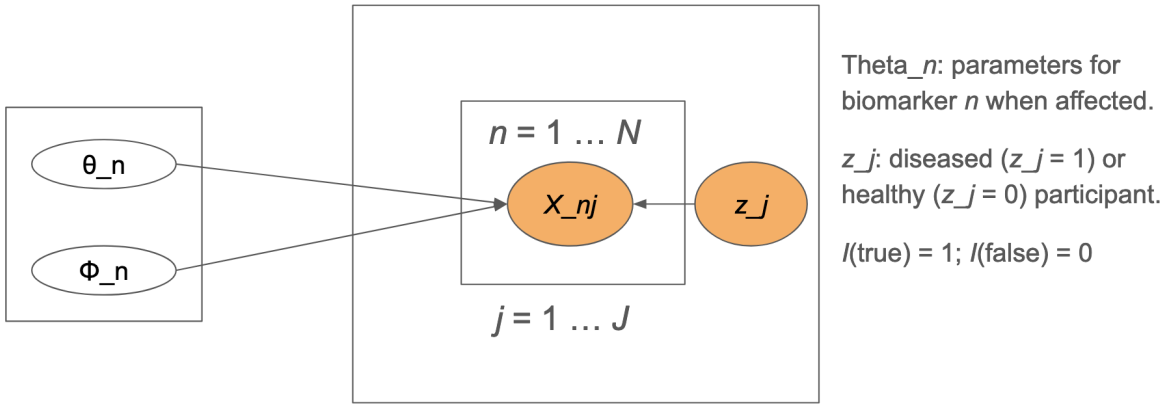


Figure 2.3: Graphical Model of Scenario 2

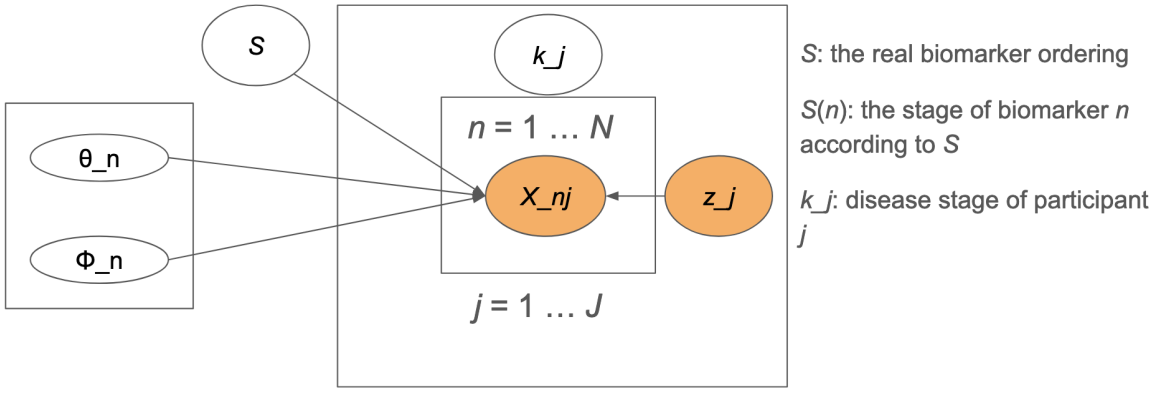


Figure 2.4: Graphical Model of Scenario 3

3 Generate Synthetic Data

In this chapter, we talk about how we generate the synthetic data of participants' biomarker measurements. These data are used to test our algorithms.

3.1 Obtain Estimated Distribution Parameters

In Section 2.4, we mentioned that EBM can be used as a generative model and we need to know S, θ, ϕ and k_j.

First, we obtained S, θ, ϕ from Chen et al. (2016):

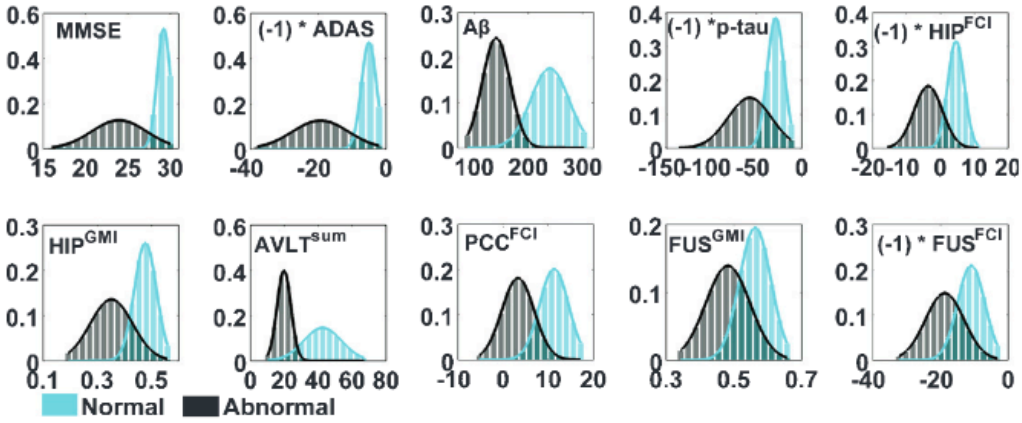


Fig. 1. Probability distributions of normal (cyan) and abnormal (black) events measured by biomarkers from the AD and CN populations. The y-axis denotes the proportion of subjects, while the x-axis indicates the detected value of each biomarker measurement. The (–1) is employed to reverse the signs of the biomarker, indicating the left distribution is an event that occurred and the right distribution is an event that did not occur.

Figure 3.1: Theta and Phi from Chen's Paper

This is our estimation:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json
```

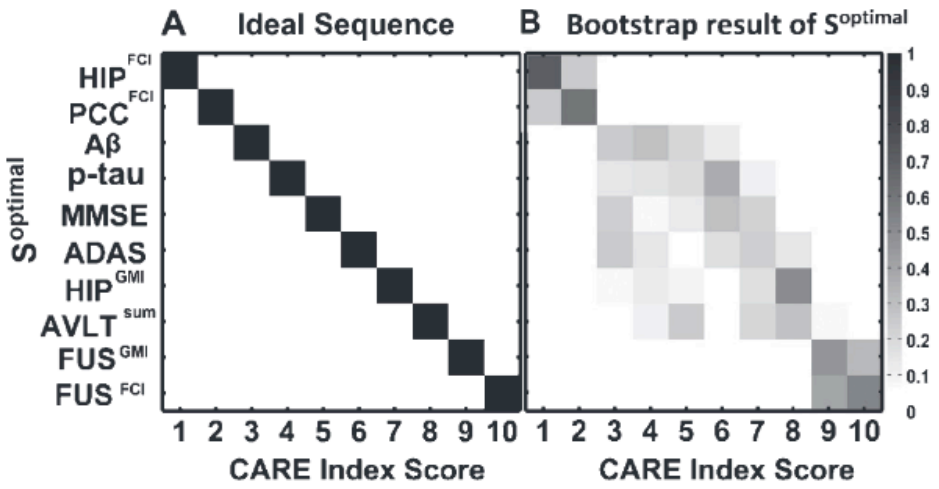


Fig. 2. Optimal temporal order, $S^{optimal}$, of the 10 AD biomarkers estimated by the EBP model. A) The y-axis shows the $S^{optimal}$ and the x-axis shows the CARE index score at which the corresponding event occurred. B) Bootstrap cross-validation of the $S^{optimal}$. Each entry in the matrix represents the proportion of the $S^{optimal}$ during 500 bootstrap samples. The proportion values range from 0 to 1 and correspond to color, from white to black. The CARE index scores with their corresponding biomarkers follow: 1, increased HIP^{FCI}; 2, decreased PCC^{FCI}; 3, decreased Aβ concentration; 4, increased p-tau concentration; 5, decreased MMSE score; 6, increased ADAS score; 7, decreased HIP^{GMI}; 8, decreased AVLT score; 9, decreased FUS^{GMI}; 10, increased FUS^{FCI}.

Figure 3.2: S from Chen's Paper

```
import scipy.stats as stats
from typing import List, Optional, Tuple, Dict
import os
import seaborn as sns
import altair as alt
```

```
all_ten_biomarker_names = np.array([
    'MMSE', 'ADAS', 'AB', 'P-Tau', 'HIP-FCI',
    'HIP-GMI', 'AVLT-Sum', 'PCC-FCI', 'FUS-GMI', 'FUS-FCI'])
# in the order above
# cyan, normal
phi_means = [28, -6, 250, -25, 5, 0.4, 40, 12, 0.6, -10]
# black, abnormal
theta_means = [22, -20, 150, -50, -5, 0.3, 20, 5, 0.5, -20]
# cyan, normal
phi_std_times_three = [2, 4, 150, 50, 5, 0.7, 45, 12, 0.2, 10]
phi_stds = [std_dev/3 for std_dev in phi_std_times_three]
# black, abnormal
theta_std_times_three = [8, 12, 50, 100, 20, 1, 20, 10, 0.2, 18]
theta_stds = [std_dev/3 for std_dev in theta_std_times_three]
```

```
# to get the real_theta_phi means and stds
hashmap_of_dicts = {}
for i, biomarker in enumerate(all_ten_biomarker_names):
    dic = {}
    # dic = {"biomarker": biomarker}
    dic['theta_mean'] = theta_means[i]
    dic['theta_std'] = theta_stds[i]
    dic['phi_mean'] = phi_means[i]
    dic['phi_std'] = phi_stds[i]
    hashmap_of_dicts[biomarker] = dic
hashmap_of_dicts

real_theta_phi = pd.DataFrame(hashmap_of_dicts).transpose().reset_index(names=['biomarker'])
real_theta_phi
```

	biomarker	theta_mean	theta_std	phi_mean	phi_std
0	MMSE	22.0	2.666667	28.0	0.666667
1	ADAS	-20.0	4.000000	-6.0	1.333333
2	AB	150.0	16.666667	250.0	50.000000
3	P-Tau	-50.0	33.333333	-25.0	16.666667
4	HIP-FCI	-5.0	6.666667	5.0	1.666667
5	HIP-GMI	0.3	0.333333	0.4	0.233333
6	AVLT-Sum	20.0	6.666667	40.0	15.000000
7	PCC-FCI	5.0	3.333333	12.0	4.000000
8	FUS-GMI	0.5	0.066667	0.6	0.066667
9	FUS-FCI	-20.0	6.000000	-10.0	3.333333

Store the parameters to a `JSON` file:

```
with open('files/real_theta_phi.json', 'w') as fp:
    json.dump(hashmap_of_dicts, fp)
```

```
biomarkers = all_ten_biomarker_names
n_biomarkers = len(biomarkers)

def plot_distribution_pair(ax, mu1, sigma1, mu2, sigma2, title):
    """mu1, sigma1: theta
    mu2, sigma2: phi
    """
```

```
    xmin = min(mu1 - 4*sigma1, mu2-4*sigma2)
    xmax = max(mu1 + 4*sigma1, mu2 + 4*sigma2)
    x = np.linspace(xmin, xmax, 1000)
    y1 = stats.norm.pdf(x, loc = mu1, scale = sigma1)
    y2 = stats.norm.pdf(x, loc = mu2, scale = sigma2)
    ax.plot(x, y1, label = "Abnormal", color = "black")
    ax.plot(x, y2, label = "Normal", color = "cyan")
    ax.fill_between(x, y1, alpha = 0.3, color = "black")
    ax.fill_between(x, y2, alpha = 0.3, color = "cyan")
    ax.set_title(title)
    ax.legend()

fig, axes = plt.subplots(2, n_biomarkers//2, figsize=(20, 10))
for i, biomarker in enumerate(biomarkers):
    ax = axes.flatten()[i]
    mu1, sigma1, mu2, sigma2 = real_theta_phi[
        real_theta_phi.biomarker == biomarker].reset_index().iloc[0, :][2:].values
    plot_distribution_pair(
        ax, mu1, sigma1, mu2, sigma2, title = biomarker)
```

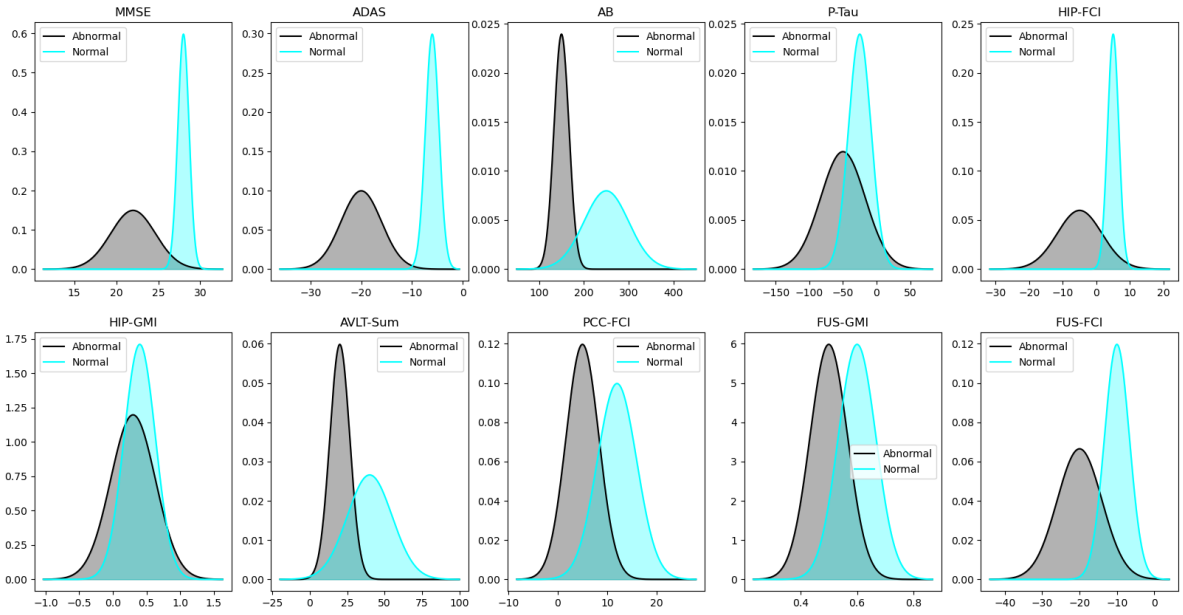


Figure 3.3: evaluate theta and phi estimations

You can compare this to Figure 3.1.

3.2 The Generating Process

In the following, we explain our data generation process.

We have the following parameters:

J: Number of participants.

R: The percentage of healthy participants.

M: Number of datasets per combination of j and r.

We set these parameters:

```
js = [50, 200, 500]
rs = [0.1, 0.25, 0.5, 0.75, 0.9]
num_of_datasets_per_combination = 50
```

So, there will be $3 \times 5 \times 50 = 750$ datasets to be generated.

We define our `generate_data_from_ebm` function:

```
def generate_data_from_ebm(
    n_participants: int,
    S_ordering: List[str],
    real_theta_phi_file: str,
    healthy_ratio: float,
    output_dir: str,
    m,  # combstr_m
    seed: Optional[int] = 0
) -> pd.DataFrame:
    """
    Simulate an Event-Based Model (EBM) for disease progression.

    Args:
    n_participants (int): Number of participants.
    S_ordering (List[str]): Biomarker names ordered according to the order
        in which each of them get affected by the disease.
    real_theta_phi_file (str): Directory of a JSON file which contains
        theta and phi values for all biomarkers.
        See real_theta_phi.json for example format.
    output_dir (str): Directory where output files will be saved.
    healthy_ratio (float): Proportion of healthy participants out of n_participants.
    seed (Optional[int]): Seed for the random number generator for reproducibility.
```

```
    Returns:
    pd.DataFrame: A DataFrame with columns 'participant', "biomarker", 'measurement',
        'diseased'.
    """
    # Parameter validation
    assert n_participants > 0, "Number of participants must be greater than 0."
    assert 0 <= healthy_ratio <= 1, "Healthy ratio must be between 0 and 1."

    # Set the seed for numpy's random number generator
    rng = np.random.default_rng(seed)

    # Load theta and phi values from the JSON file
    try:
        with open(real_theta_phi_file) as f:
            real_theta_phi = json.load(f)
    except FileNotFoundError:
        raise FileNotFoundError(f"File {real_theta_phi} not fount")
    except json.JSONDecodeError:
        raise ValueError(
            f"File {real_theta_phi_file} is not a valid JSON file.")

    n_biomarkers = len(S_ordering)
    n_stages = n_biomarkers + 1

    n_healthy = int(n_participants * healthy_ratio)
    n_diseased = int(n_participants - n_healthy)

    # Generate disease stages
    kjs = np.concatenate((np.zeros(n_healthy, dtype=int),
                          rng.integers(1, n_stages, n_diseased)))
    # shuffle so that it's not 0s first and then disease stages bur all random
    rng.shuffle(kjs)

    # Initiate biomarker measurement matrix (J participants x N biomarkers) with None
    X = np.full((n_participants, n_biomarkers), None, dtype=object)

    # Create distributions for each biomarker
    theta_dist = {biomarker: stats.norm(
        real_theta_phi[biomarker]['theta_mean'],
        real_theta_phi[biomarker]['theta_std']
    ) for biomarker in S_ordering}
```

```
    phi_dist = {biomarker: stats.norm(
        real_theta_phi[biomarker]['phi_mean'],
        real_theta_phi[biomarker]['phi_std']
    ) for biomarker in S_ordering}

    # Populate the matrix with biomarker measurements
    for j in range(n_participants):
        for n, biomarker in enumerate(S_ordering):
            # because for each j, we generate X[j, n] in the order of S_ordering,
            # the final dataset will have this ordering as well.
            k_j = kjs[j]
            S_n = n + 1

            # Assign biomarker values based on the participant's disease stage
            # affected, or not_affected, is regarding the biomarker, not the participant
            if k_j >= 1:
                if k_j >= S_n:
                    # rvs() is affected by np.random()
                    X[j, n] = (
                        j, biomarker, theta_dist[biomarker].rvs(random_state=rng), k_j, S_n,
                else:
                    X[j, n] = (j, biomarker, phi_dist[biomarker].rvs(random_state=rng),
                               k_j, S_n, 'not_affected')
            # if the participant is healthy
            else:
                X[j, n] = (j, biomarker, phi_dist[biomarker].rvs(random_state=rng),
                           k_j, S_n, 'not_affected')

    df = pd.DataFrame(X, columns=S_ordering)
    # make this dataframe wide to long
    df_long = df.melt(var_name="Biomarker", value_name="Value")
    data = df_long['Value'].apply(pd.Series)
    data.columns = ['participant', "biomarker",
                    'measurement', 'k_j', 'S_n', 'affected_or_not']

    # biomarker_name_change_dic = dict(
    #     zip(S_ordering, range(1, n_biomarkers + 1)))
    data['diseased'] = data.apply(lambda row: row.k_j > 0, axis=1)
    # data.drop(['k_j', 'S_n', 'affected_or_not'], axis=1, inplace=True)
    # data['biomarker'] = data.apply(
    #     lambda row: f"{row.biomarker} ({biomarker_name_change_dic[row.biomarker]})", axis=
```

```
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    filename = f"{int(healthy_ratio*n_participants)}|{n_participants}_{m}"
    data.to_csv(f'{output_dir}/{filename}.csv', index=False)
    # print("Data generation done! Output saved to:", filename)
    return data
```

```
S_ordering = np.array([
    'HIP-FCI', 'PCC-FCI', 'AB', 'P-Tau', 'MMSE', 'ADAS',
    'HIP-GMI', 'AVLT-Sum', 'FUS-GMI', 'FUS-FCI'
])

# where the generated data will be saved
output_dir = 'data'

# We run the following only once; once the data is generated, we no longer run it
# We still show the codes to present our generation process
torun = False
```

```
if torun:
    real_theta_phi_file = 'files/real_theta_phi.json'
    for j in js:
        for r in rs:
            for m in range(0, num_of_datasets_per_combination):
                generate_data_from_ebm(
                    n_participants=j,
                    S_ordering=S_ordering,
                    real_theta_phi_file=real_theta_phi_file,
                    healthy_ratio=r,
                    output_dir=output_dir,
                    m=m,
                    seed = int(j*10 + (r * 100) + m),
                )
        print(f'Done for J={j}')
```

3.3 Visualize Synthetic Data

Above, we have generated 750 datasets, named in the fashion of `150|200_3`, which means the third dataset when $j = 200$ and $r = 0.75$.

Next, we try to visualize this dataset.

```
df = pd.read_csv(f"{output_dir}/150|200_3.csv")
df.head()
```

	participant	biomarker	measurement	k_j	S_n	affected_or_not	diseased
0	0	HIP-FCI	3.135981	0	1	not_affected	False
1	1	HIP-FCI	12.593704	2	1	affected	True
2	2	HIP-FCI	6.220776	0	1	not_affected	False
3	3	HIP-FCI	3.545100	0	1	not_affected	False
4	4	HIP-FCI	3.966541	0	1	not_affected	False

```
df.shape
```

```
(2000, 7)
```

This dataset has 2000 rows because we have 200 participants and 10 biomarkers.

3.3.1 Distribution of all biomarker values

```
alt.renderers.enable('png')
alt.Chart(df).transform_density(
    'measurement',
    as_=['measurement', 'Density'],
    groupby=['biomarker']
).mark_area().encode(
    x="measurement:Q",
    y="Density:Q",
    facet = alt.Facet(
        "biomarker:N",
        columns = 5
    ),
    color=alt.Color(
        'biomarker:N'
    )
).properties(
    width= 100,
    height = 180,
```

```
).properties(
    title='Distribution of biomarker measurments'
)
```

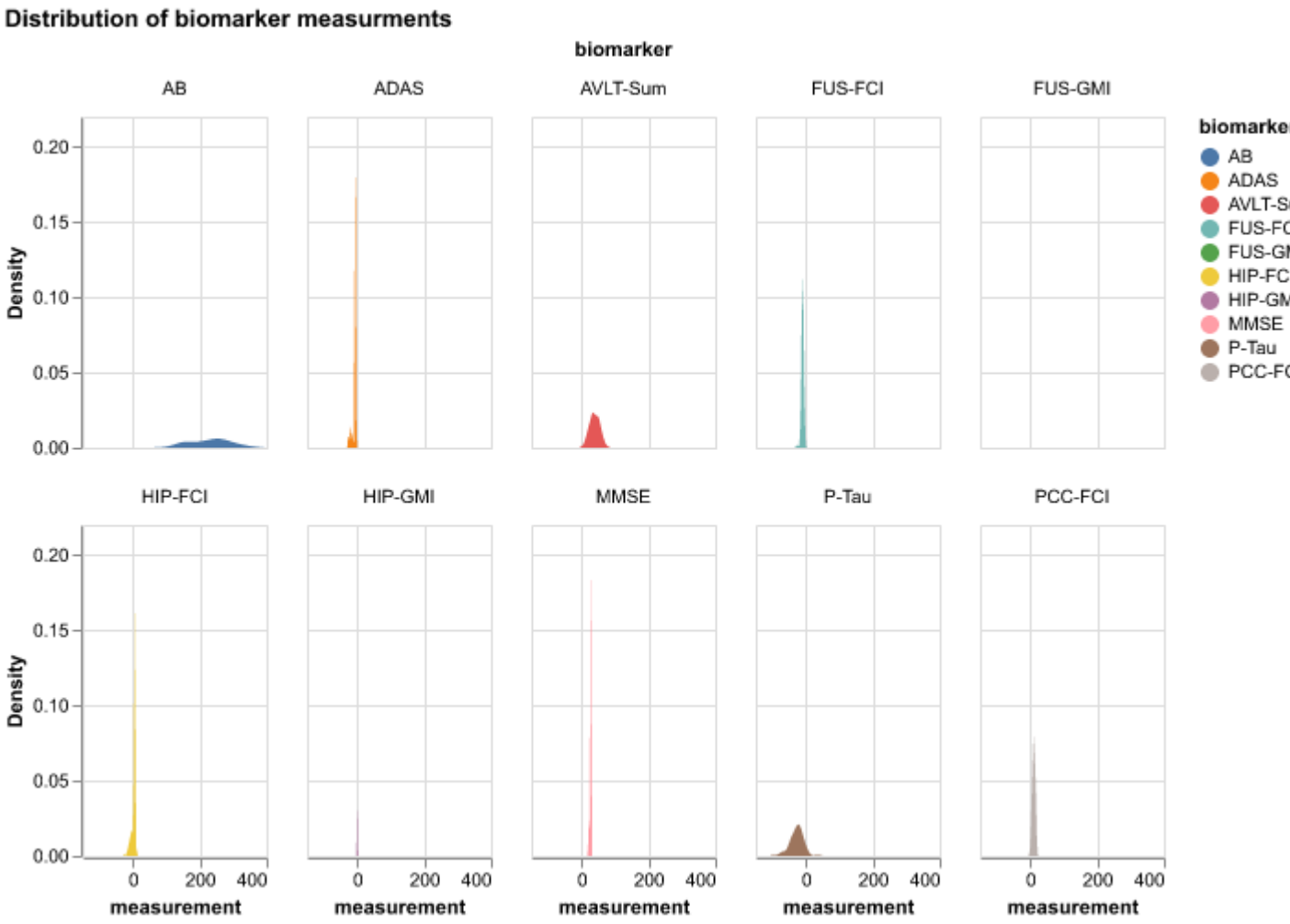


Figure 3.4: Distribution of biomarker measurments

3.3.2 Distribution of A Specific Biomarker

```
idx = 1
biomarkers = df.biomarker.unique()
bio_data = df[df.biomarker==biomarkers[idx]]
```

```
alt.Chart(bio_data).transform_density(
    'measurement',
    as_=['measurement', 'Density'],
    groupby=['affected_or_not']
).mark_area().encode(
    x="measurement:Q",
    y="Density:Q",
    facet = alt.Facet(
        "affected_or_not:N",
    ),
    color=alt.Color(
        'affected_or_not:N'
    )
).properties(
    width= 240,
    height = 200,
).properties(
    title=f'Distribution of {biomarker} measurements'
)
```

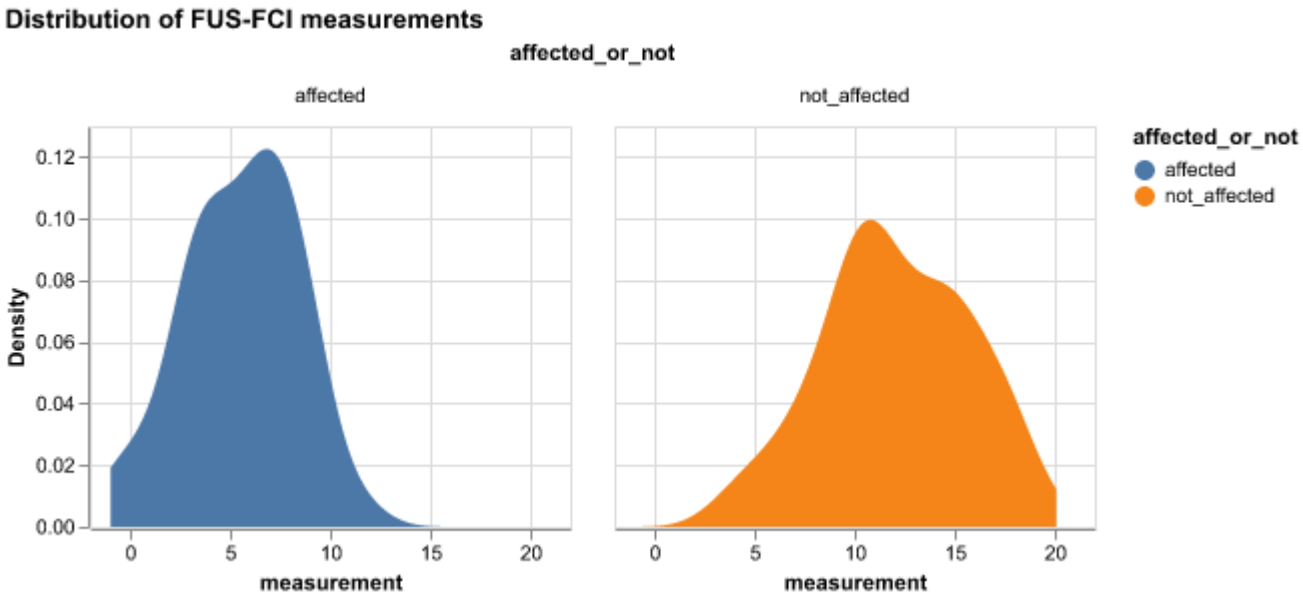


Figure 3.5: Distribution of HIP-FCI measurements, compring bewteen affected and non-affected group

3.3.3 Looking into A Specific Participant

```
pidx = 1
p_data = df[df.participant == pidx]
p_data
```

	participant	biomarker	measurement	k_j	S_n	affected_or_not	diseased
1	1	HIP-FCI	12.593704	2	1	affected	True
201	1	PCC-FCI	7.164017	2	2	affected	True
401	1	AB	182.033823	2	3	not_affected	True
601	1	P-Tau	-25.345325	2	4	not_affected	True
801	1	MMSE	27.600823	2	5	not_affected	True
1001	1	ADAS	-4.920415	2	6	not_affected	True
1201	1	HIP-GMI	0.099052	2	7	not_affected	True
1401	1	AVLT-Sum	30.270797	2	8	not_affected	True
1601	1	FUS-GMI	0.658954	2	9	not_affected	True
1801	1	FUS-FCI	-11.701559	2	10	not_affected	True

```
pidx =1 # participant index
p_data = df[df.participant == pidx]
alt.Chart(p_data).mark_bar().encode(
    x='biomarker',
    y='measurement',
    color=alt.Color(
        'affected_or_not:N'
    ),
    tooltip=['biomarker', 'affected_or_not', 'measurement']
).interactive().properties(
    title=f'Distribution of biomarker measurements for participant #{idx} (k_j = {p_data.k_j
)
```

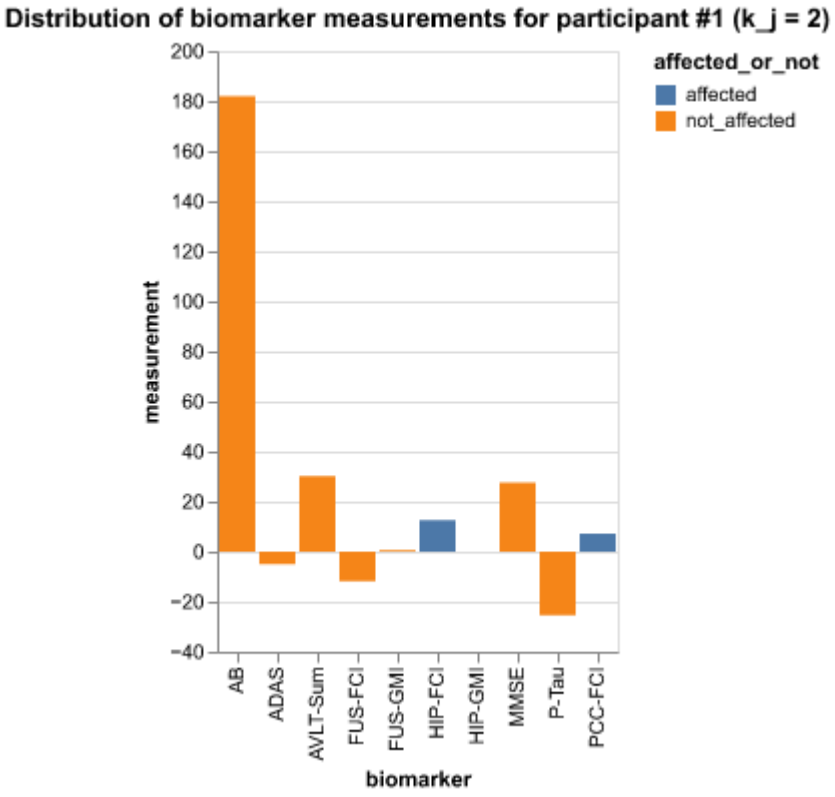


Figure 3.6: Distribution of biomarker measurements for a specific participant

4 Estimate Distribution Parameters

Given S, and a biomarker's measurements, how can we estimate $\mathcal{N}(\theta_\mu, \theta_\sigma)$ and $\mathcal{N}(\phi_\mu, \phi_\sigma)$?

```
import pandas as pd
import numpy as np
import altair as alt
import math
from scipy.stats import norm
from sklearn.cluster import AgglomerativeClustering
from typing import Dict
import json
from sklearn.cluster import KMeans
from collections import defaultdict
from scipy.stats import mode
```

```
output_dir = 'data'
df = pd.read_csv(f"{output_dir}/150|200_3.csv")
biomarkers = df.biomarker.unique()
idx = 1
biomarker_df = df[df.biomarker==biomarkers[idx]]
biomarker_df.sample(10)
```

	participant	biomarker	measurement	k_j	S_n	affected_or_not	diseased
307	107	PCC-FCI	2.875619	10	2	affected	True
360	160	PCC-FCI	14.765535	0	2	not_affected	False
365	165	PCC-FCI	8.913005	0	2	not_affected	False
354	154	PCC-FCI	11.380722	0	2	not_affected	False
200	0	PCC-FCI	6.567299	0	2	not_affected	False
285	85	PCC-FCI	11.371021	0	2	not_affected	False
375	175	PCC-FCI	3.771693	9	2	affected	True
203	3	PCC-FCI	14.266635	0	2	not_affected	False
295	95	PCC-FCI	4.949612	0	2	not_affected	False
299	99	PCC-FCI	17.490539	0	2	not_affected	False

```
biomarker_df.shape
```

```
(200, 7)
```

4.1 Hard K-Means

> **Tip**
>
> To use this algorithm, we only need to know (1) whether this participant is diseased; and (2) each biomarker measurement.

The first method we can use is hard K-Means. We clustering a certain biomarker's measurements into two clusters. A clustering is successful if:

- There are two, and only two clusters.
- Each clustes has more than one element (This is to make sure that the standard deviation of this biomarker's theta or phi is non-zero)

Ideally, we wanted all healthy participants to be grouped into a single cluster, which is why we initially tried using the constrained K-Means algorithm implemented by Babaki (2017). However, the algorithm did not work as intended.

We therefore designed a hard k-means algorithm to satisfy our needs:

- We try hard K-Means multiple times; If the two above mentioned requirements are not met, then
- We group the measurements into two random clusters; If the two above mentioned requirements are still not met, then raise an error and stop.

```
def compute_theta_phi_for_biomarker(biomarker_df, max_attempt = 50, seed = None):
    """get theta and phi parameters for this biomarker
    input:
        - biomarker_df: a pd.dataframe of a specific biomarker
    output:
        - a tuple: theta_mean, theta_std, phi_mean, phi_std
    """
    if seed is not None:
        # Set the seed for numpy's random number generator
        rng = np.random.default_rng(seed)
    else:
        rng = np.random
```

```
n_clusters = 2
measurements = np.array(biomarker_df['measurement']).reshape(-1, 1)
healthy_df = biomarker_df[biomarker_df['diseased'] == False]

# clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward')
# predictions = clustering.fit_predict(measurements)

# # Verify that AgglomerativeClustering produced exactly 2 clusters with more than 1 memb
# cluster_counts = np.bincount(predictions) # array([3, 2])
# if len(cluster_counts) != n_clusters or any(c <= 1 for c in cluster_counts):
#     print("AgglomerativeClustering did not yield the required clusters, switching to KM
# # If AgglomerativeClustering fails, attempt KMeans with a max_attempt limit

curr_attempt = 0
n_init_value = 10
clustering_setup = KMeans(n_clusters=n_clusters, n_init=n_init_value)

while curr_attempt < max_attempt:
    clustering_result = clustering_setup.fit(measurements)
    predictions = clustering_result.labels_
    cluster_counts = np.bincount(predictions) # array([3, 2])

    if len(cluster_counts) == n_clusters and all(c > 1 for c in cluster_counts):
        break
    curr_attempt += 1
else:
    print(f"KMeans failed. Try randomizing the predictions")
    predictions = rng.choice([0, 1], size=len(measurements))
    cluster_counts = np.bincount(predictions)
    if len(cluster_counts) != n_clusters or not all(c > 1 for c in cluster_counts):
        raise ValueError(f"KMeans clustering failed to find valid clusters within max_att

healthy_predictions = predictions[healthy_df.index]
mode_result = mode(healthy_predictions, keepdims=False).mode
phi_cluster_idx = mode_result[0] if isinstance(mode_result, np.ndarray) else mode_result
theta_cluster_idx = 1 - phi_cluster_idx

# two empty clusters to strore measurements
clustered_measurements = [[] for _ in range(2)]
# Store measurements into their cluster
for i, prediction in enumerate(predictions):
    clustered_measurements[prediction].append(measurements[i][0])
```

```
     # Calculate means and standard deviations
    theta_mean, theta_std = np.mean(
        clustered_measurements[theta_cluster_idx]), np.std(
            clustered_measurements[theta_cluster_idx])
    phi_mean, phi_std = np.mean(
        clustered_measurements[phi_cluster_idx]), np.std(
            clustered_measurements[phi_cluster_idx])

    # Check for invalid values
    if any(np.isnan(v) or v == 0 for v in [theta_std, phi_std, theta_mean, phi_mean]):
        raise ValueError("One of the calculated values is invalid (0 or NaN).")

    return theta_mean, theta_std, phi_mean, phi_std

def get_theta_phi_estimates(
    data: pd.DataFrame,
) -> Dict[str, Dict[str, float]]:
    """
    Obtain theta and phi estimates (mean and standard deviation) for each biomarker.

    Args:
    data (pd.DataFrame): DataFrame containing participant data with columns 'participant',
        'biomarker', 'measurement', and 'diseased'.
    # biomarkers (List[str]): A list of biomarker names.

    Returns:
    Dict[str, Dict[str, float]]: A dictionary where each key is a biomarker name,
        and each value is another dictionary containing the means and standard deviations
        for theta and phi of that biomarker, with keys 'theta_mean', 'theta_std', 'phi_mean'
        and 'phi_std'.
    """
    # empty hashmap of dictionaries to store the estimates
    estimates = {}
    biomarkers = data.biomarker.unique()
    for biomarker in biomarkers:
        # Filter data for the current biomarker
        # reset_index is necessary here because we will use healthy_df.index later
        biomarker_df = data[data['biomarker']
                            == biomarker].reset_index(drop=True)
        theta_mean, theta_std, phi_mean, phi_std = compute_theta_phi_for_biomarker(
            biomarker_df)
        estimates[biomarker] = {
```

```
            'theta_mean': theta_mean,
            'theta_std': theta_std,
            'phi_mean': phi_mean,
            'phi_std': phi_std
        }
    return estimates
```

```
hard_kmeans_estimates = get_theta_phi_estimates(data = df)
hard_kmeans_estimates_df = pd.DataFrame.from_dict(
    hard_kmeans_estimates, orient='index')
hard_kmeans_estimates_df.reset_index(names = 'biomarker', inplace=True)
hard_kmeans_estimates_df
```

	biomarker	theta_mean	theta_std	phi_mean	phi_std
0	HIP-FCI	-8.587833	5.365053	4.903437	2.008974
1	PCC-FCI	7.288870	2.797150	14.569768	2.274322
2	AB	173.199732	33.778905	277.009573	34.503492
3	P-Tau	-49.329924	15.988080	-15.568346	12.688703
4	MMSE	22.272803	1.624078	28.011392	0.805532
5	ADAS	-20.582091	3.853234	-6.002048	1.487443
6	HIP-GMI	0.576991	0.157259	0.192231	0.128850
7	AVLT-Sum	27.808002	8.561387	52.524815	7.989840
8	FUS-GMI	0.519974	0.047474	0.628440	0.038051
9	FUS-FCI	-7.061708	1.928477	-12.647345	2.927590

```
with open('files/real_theta_phi.json', 'r') as f:
    truth = json.load(f)
truth_df = pd.DataFrame.from_dict(truth, orient='index')
truth_df.reset_index(names = 'biomarker', inplace=True)
truth_df
```

	biomarker	theta_mean	theta_std	phi_mean	phi_std
0	MMSE	22.0	2.666667	28.0	0.666667
1	ADAS	-20.0	4.000000	-6.0	1.333333
2	AB	150.0	16.666667	250.0	50.000000
3	P-Tau	-50.0	33.333333	-25.0	16.666667
4	HIP-FCI	-5.0	6.666667	5.0	1.666667
5	HIP-GMI	0.3	0.333333	0.4	0.233333
6	AVLT-Sum	20.0	6.666667	40.0	15.000000

	biomarker	theta_mean	theta_std	phi_mean	phi_std
7	PCC-FCI	5.0	3.333333	12.0	4.000000
8	FUS-GMI	0.5	0.066667	0.6	0.066667
9	FUS-FCI	-20.0	6.000000	-10.0	3.333333

Now let's compare the results using plots:

```
def obtain_theta_phi_params(biomarker, estimate_df, truth):
    '''This is to obtain both true and estimated theta and phi params for each biomarker '''
    biomarker_data_est = estimate_df[estimate_df.biomarker == biomarker].reset_index()
    biomarker_data = truth[truth.biomarker == biomarker].reset_index()
    # theta for affected
    theta_mean_est = biomarker_data_est.theta_mean[0]
    theta_std_est = biomarker_data_est.theta_std[0]

    theta_mean = biomarker_data.theta_mean[0]
    theta_std = biomarker_data.theta_std[0]

    # phi for not affected
    phi_mean_est = biomarker_data_est.phi_mean[0]
    phi_std_est = biomarker_data_est.phi_std[0]

    phi_mean = biomarker_data.phi_mean[0]
    phi_std = biomarker_data.phi_std[0]

    return theta_mean, theta_std, theta_mean_est, theta_std_est, phi_mean, phi_std, phi_mean_

def make_chart(biomarkers, estimate_df, truth, title):
    alt.renderers.enable('png')
    charts = []
    for biomarker in biomarkers:
        theta_mean, theta_std, theta_mean_est, theta_std_est, phi_mean, phi_std, phi_mean_est
        biomarker, estimate_df, truth)
        mean1, std1 = theta_mean, theta_std
        mean2, std2 = theta_mean_est, theta_std_est

        # Generating points on the x axis
        x_thetas = np.linspace(min(mean1 - 3*std1, mean2 - 3*std2),
                               max(mean1 + 3*std1, mean2 + 3*std2), 1000)

        # Creating DataFrames for each distribution
```

```
        df1 = pd.DataFrame({'x': x_thetas, 'pdf': norm.pdf(x_thetas, mean1, std1), 'Distribut
        df2 = pd.DataFrame({'x': x_thetas, 'pdf': norm.pdf(x_thetas, mean2, std2), 'Distribut

        # Combining the DataFrames
        df3 = pd.concat([df1, df2])

        # Altair plot
        chart_theta = alt.Chart(df3).mark_line().encode(
            x='x',
            y='pdf',
            color=alt.Color('Distribution:N', legend=alt.Legend(title="Theta"))
        ).properties(
            title=f'{biomarker}, Theta',
            width=100,
            height=100
            )

        mean1, std1 = phi_mean, phi_std
        mean2, std2 = phi_mean_est, phi_std_est

        # Generating points on the x axis
        x_phis = np.linspace(min(mean1 - 3*std1, mean2 - 3*std2),
                             max(mean1 + 3*std1, mean2 + 3*std2), 1000)

        # Creating DataFrames for each distribution
        df1 = pd.DataFrame({'x': x_phis, 'pdf': norm.pdf(x_phis, mean1, std1), 'Distribution
        df2 = pd.DataFrame({'x': x_phis, 'pdf': norm.pdf(x_phis, mean2, std2), 'Distribution

        # Combining the DataFrames
        df3 = pd.concat([df1, df2])

        # Altair plot
        chart_phi = alt.Chart(df3).mark_line().encode(
            x='x',
            y='pdf',
            color=alt.Color('Distribution:N', legend=alt.Legend(title="Phi"))
        ).properties(
            title=f'{biomarker}, Phi',
            width=100,
            height=100
            )
```

```
        # Concatenate theta and phi charts horizontally
        hconcat_chart = alt.hconcat(chart_theta, chart_phi).resolve_scale(color="independent")

        # Append the concatenated chart to the list of charts
        charts.append(hconcat_chart)
    # Concatenate all the charts vertically
    final_chart = alt.vconcat(*charts).properties(title = title)

    # Display the final chart
    final_chart.display()
```

```
make_chart(
    biomarkers[0:5],
    hard_kmeans_estimates_df,
    truth_df,
    title = "Comparing Theta and Phi Distributions Using hard k-means"
)
```

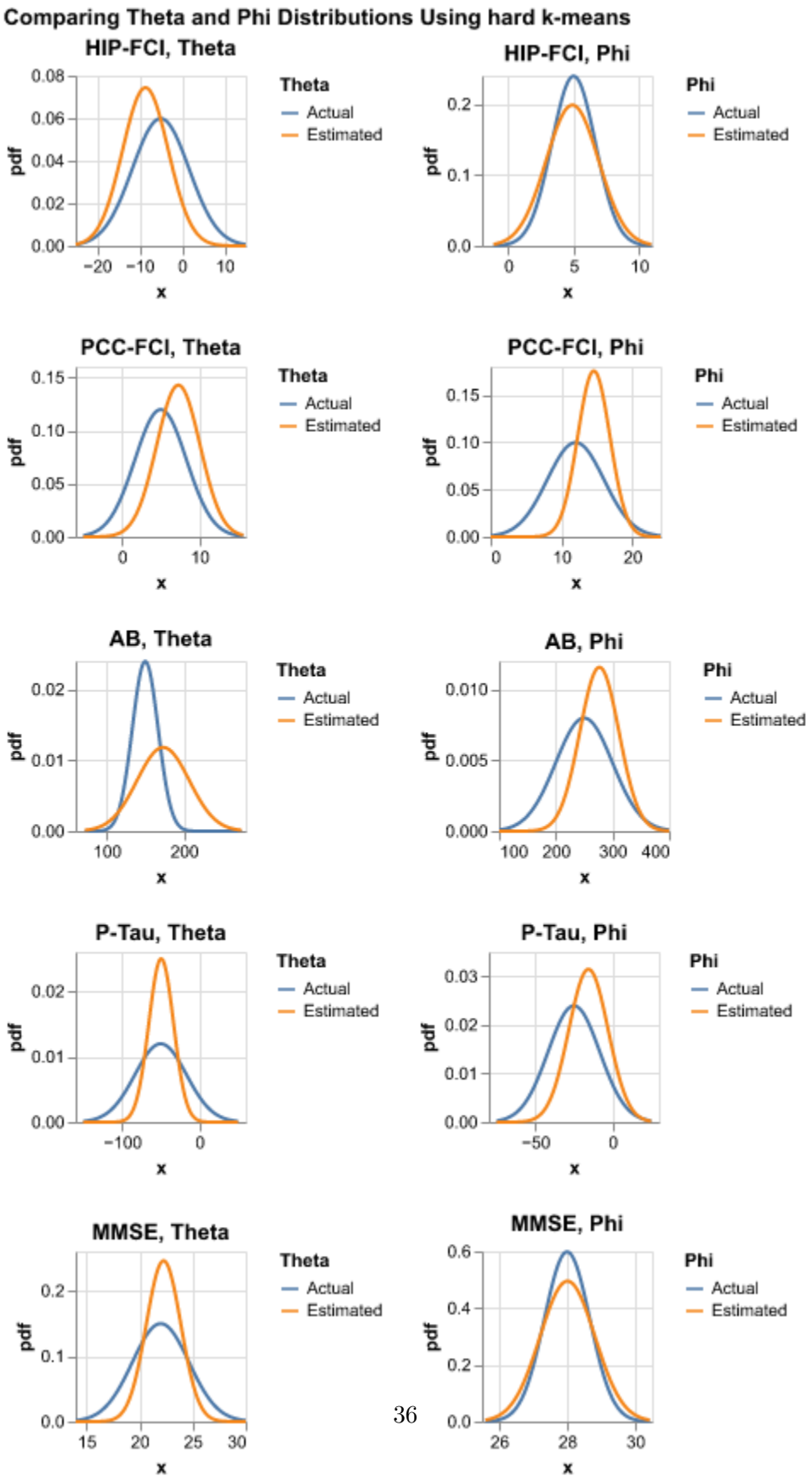


Figure 4.1: Comparing Theta and Phi Distributions Using Simple Clusering

It turns out the result is not very desriable.

4.2 Conjugate Priors

The second method we may utilize is conjugate priors. Conjugacy occurs when the posterior distribution is in the same family of distribution as the prior distribution, but with new parameter values.

Why conjugacy is important? Because without it, one has to do the integration, which oftentimes is hard.

Three major conjugate families:

- Beta-Binomial
- Gamma-Poisson
- Normal-Normal

In our example, we assume that the measurement data for each biomarker follows a normal distribution; however, we do not know the exact μ and σ. Our job is to estimate the two parameters for each biomarker based on the data we have.

According to *An Introduction to Bayesian Thinking* by Clyde et al. (2022), if the data comes from a normal distribution with unknown μ and σ, the conjugate prior for μ has a normal distribution with mean m_0 and variance $\frac{\sigma^2}{n_0}$. The conjugate prior for $\frac{1}{\sigma^2}$ has a Gamma distribution with shape $\frac{v_0}{2}$ and rate $\frac{v_0 s_0^2}{2}$ where

- m_0: prior estimate of μ.
- n_0: how strongly is the prior belief in m_0 is held.
- s_0^2: prior estimate of σ^2.
- v_0: prior degress of freedome, influencing the certainty of s_0^2.

That is to say:

$$\mu|\sigma^2 \sim \mathcal{N}(m_0, \sigma^2/n_0)$$

$$1/\sigma^2 \sim Gamma\left(\frac{v_0}{2}, \frac{v_0 s_0^2}{2}\right)$$

Combined, we have:

$$(\mu, 1/\sigma^2) \sim NormalGamma(m_0, n_0, s_0^2, v_0)$$

The posterior also follows a Normal-Gamma distribution:

$$(\mu, 1/\sigma^2)|data \sim NormalGamma(m_n, n_n, s_n^2, v_n)$$

More specifically

$$1/\sigma^2|data \sim Gamma(v_n/2, s_n^2 v_n/2)$$

$$\mu|data, \sigma^2 \sim \mathcal{N}(m_n, \sigma^2/n_n)$$

Based on the above two equations, we know that the mean of posterior mean is m_n and the mean of the posterior variance is $(s_n^2 v_n/2)/(v_n/2)$. This is beceause the expected value of $Gamma(\alpha, \beta)$ is $\frac{\alpha}{\beta}$.

where

- m_n: posterior mean, mode, and median for μ
- n_n: posterior sample size
- s_n^2: posterior variance
- v_n: posterior degrees of freedome

The updating rules to get the new hyper-parameters:

$$m_n = \frac{n}{n+n_0}\bar{y} + \frac{n_0}{n+n_0}m_0$$

$$n_n = n_0 + n$$

$$v_n = v_0 + n$$

$$s_n^2 = \frac{1}{v_n}\left[s^2(n-1) + s_0^2 v_0 + \frac{n_0 n}{n_n}(\bar{y} - m_0)^2\right]$$

where

- n: sample size
- $\bar{y}$: sample mean
- s^2: sample variance

> **Tip**
>
> To apply the algorithm of conjugate priors, we assume we already know S and k_j, alongside biomarker measurement (X_{nj}). Based on S and k_j, we can infer whether a biomarker is affected by the disease or not.

```
def estimate_params_exact(m0, n0, s0_sq, v0, data):
    '''This is to estimate means and vars based on conjugate priors
    Inputs:
        - data: a vector of measurements
        - m0: prior estimate of $\mu$.
        - n0: how strongly is the prior belief in $m_0$ is held.
        - s0_sq: prior estimate of $\sigma^2$.
        - v0: prior degress of freedome, influencing the certainty of $s_0^2$.

    Outputs:
        - mu estiate, std estimate
    '''
    # Data summary
    sample_mean = np.mean(data)
    sample_size = len(data)
    sample_var = np.var(data, ddof=1)  # ddof=1 for unbiased estimator

    # Update hyperparameters for the Normal-Inverse Gamma posterior
    updated_m0 = (n0 * m0 + sample_size * sample_mean) / (n0 + sample_size)
    updated_n0 = n0 + sample_size
    updated_v0 = v0 + sample_size
    updated_s0_sq = (1 / updated_v0) * ((sample_size - 1) * sample_var + v0 * s0_sq +
                                        (n0 * sample_size / updated_n0) * (sample_mean - m0)
    updated_alpha = updated_v0/2
    updated_beta = updated_v0*updated_s0_sq/2

    # Posterior estimates
    mu_posterior_mean = updated_m0
    sigma_squared_posterior_mean = updated_beta/updated_alpha

    mu_estimation = mu_posterior_mean
    std_estimation = np.sqrt(sigma_squared_posterior_mean)

    return mu_estimation, std_estimation

def get_theta_phi_conjugate_priors(biomarkers, data_we_have, theta_phi_kmeans):
```

```
    '''To get estimated parameters, returns a hashmap
    Input:
    - biomarkers: biomarkers
    - data_we_have: participants data filled with initial or updated participant_stages
    - theta_phi_kmeans: a hashmap of dicts, which are the prior theta and phi values
        obtained from the initial hard k-means algorithm

    Output:
    - a hashmap of dictionaries. Key is biomarker name and value is a dictionary.
    Each dictionary contains the theta and phi mean/std values for a specific biomarker.
    '''
    # empty list of dictionaries to store the estimates
    hashmap_of_means_stds_estimate_dicts = {}

    for biomarker in biomarkers:
        # Initialize dictionary outside the inner loop
        dic = {'biomarker': biomarker}
        for affected in ['affected', 'not_affected']:
            data_full = data_we_have[(data_we_have.biomarker == biomarker) & (
                data_we_have.affected_or_not == affected)]
            if len(data_full) > 1:
                measurements = data_full.measurement
                s0_sq = np.var(measurements, ddof=1)
                m0 = np.mean(measurements)
                mu_estimate, std_estimate = estimate_params_exact(
                    m0=m0, n0=1, s0_sq=s0_sq, v0=1, data=measurements)
                if affected == 'affected':
                    dic['theta_mean'] = mu_estimate
                    dic['theta_std'] = std_estimate
                else:
                    dic['phi_mean'] = mu_estimate
                    dic['phi_std'] = std_estimate
            # If there is only one observation or not observation at all, resort to theta_phi
            # YES, IT IS POSSIBLE THAT DATA_FULL HERE IS NULL
            # For example, if a biomarker indicates stage of (num_biomarkers), but all partic
            # are smaller than that stage; so that for all participants, this biomarker is no
            else:
                print('not enough data here, so we have to use theta phi estimates from hard
                # print(theta_phi_kmeans)
                if affected == 'affected':
                    dic['theta_mean'] = theta_phi_kmeans[biomarker]['theta_mean']
                    dic['theta_std'] = theta_phi_kmeans[biomarker]['theta_std']
```

```
                else:
                    dic['phi_mean'] = theta_phi_kmeans[biomarker]['phi_mean']
                    dic['phi_std'] = theta_phi_kmeans[biomarker]['phi_std']
        # print(f"biomarker {biomarker} done!")
        hashmap_of_means_stds_estimate_dicts[biomarker] = dic
    return hashmap_of_means_stds_estimate_dicts
```

```
conjugate_prior_theta_phi = get_theta_phi_conjugate_priors(
    biomarkers = biomarkers,
    data_we_have = df,
    theta_phi_kmeans = hard_kmeans_estimates
)
cp_df = pd.DataFrame.from_dict(conjugate_prior_theta_phi, orient='index')
cp_df.reset_index(drop=True, inplace=True)
cp_df
```

	biomarker	theta_mean	theta_std	phi_mean	phi_std
0	HIP-FCI	-5.378366	7.233991	5.092800	1.514402
1	PCC-FCI	5.521792	2.777207	12.071769	3.671679
2	AB	151.143708	14.806694	251.973564	51.382188
3	P-Tau	-41.768257	34.857945	-24.739527	14.928907
4	MMSE	23.122406	2.446874	28.049683	0.718493
5	ADAS	-19.633304	4.582900	-5.902198	1.278311
6	HIP-GMI	0.425625	0.272876	0.379542	0.235348
7	AVLT-Sum	21.664360	3.755735	40.700638	14.480463
8	FUS-GMI	0.482745	0.055585	0.590434	0.063730
9	FUS-FCI	-18.566905	5.781937	-9.648705	3.099195

> **Note**
>
> When we estimate θ and ϕ using conjugate priors, we need to use the result from hard k-means as a fall back because it is possible that for a specific biomarker, either the `affected` or the `not_affected` group is empty. If that is the case, we are not able to estimate relevant parameters and have to resort to the fallback result.

```
make_chart(
    biomarkers[0:5],
    cp_df,
    truth_df,
```

```
    title = "Comparing Theta and Phi Distributions Using Conjugate Priors"
)
```

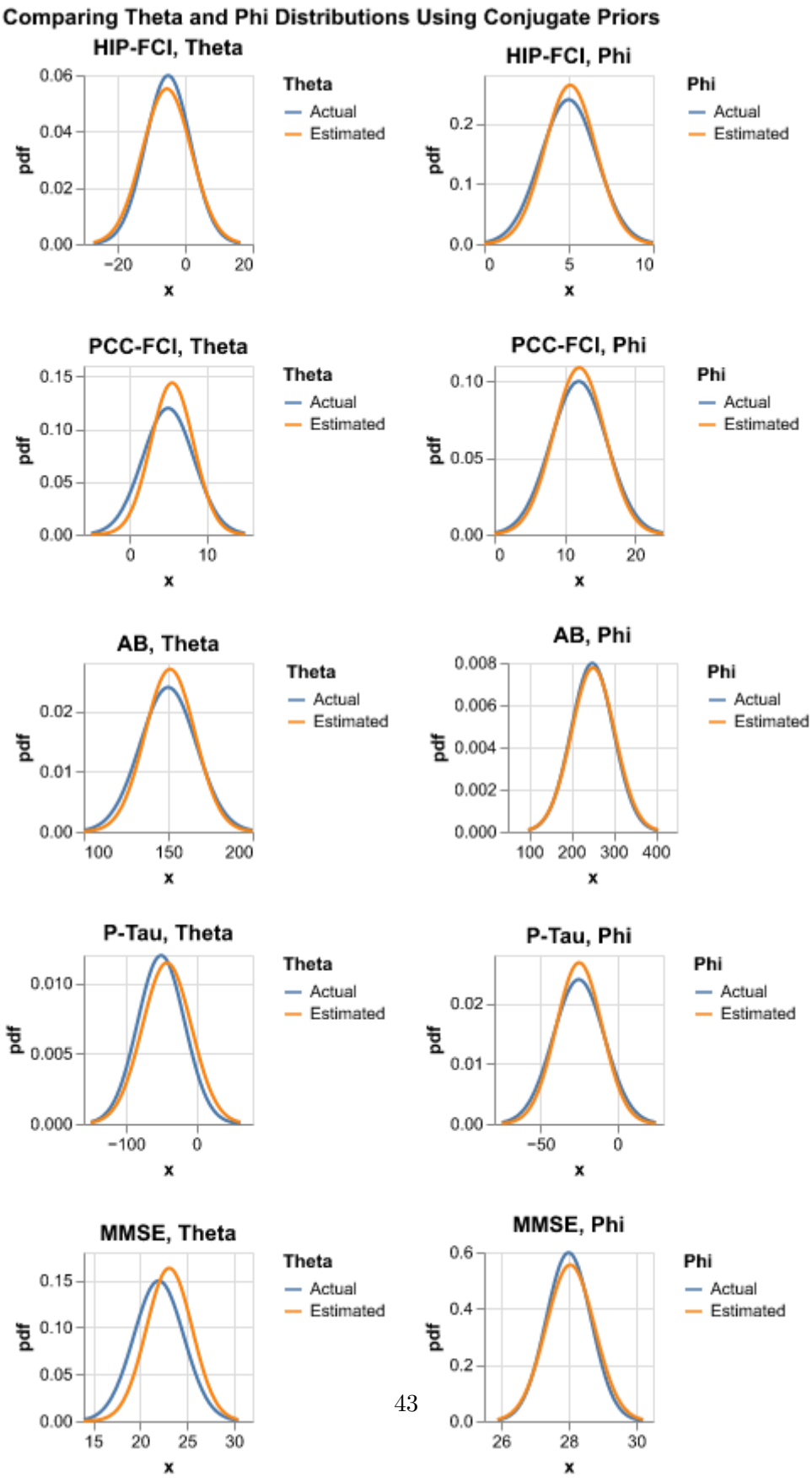


Figure 4.2: Comparing Theta and Phi Distributions Using Conjugate Prior

4.3 Soft K-Means

Conjugate Priors assumes we know k_j, which often times is not already known. Our hard k-means algorithm is only taking advantage of X_{nj} and whether participants are diseased or not, leaving S, which is known to us, unexploited.

Soft K-Means is a good alternative to these two because it utilizes S while at the same time do not assume we know k_j.

The logic of soft-kmeans is this;

1. If a participant is diseased, we iterate through all possible disease stages, and calculate the associated likelihood using Equation 2.1. We then normalize these likelihoods to obtain the estimated probability of this participant being at each stage. For example, if there are three possible stages, and the associated likelihoods are `[1, 3, 6]`, then the normalized likelihoods would be `[0.1, 0.3, 0.6]`.

> **Tip**
>
> You may wonder how we can use Equation 2.1 when we do not know θ and ϕ yet (which is exactly what we are trying to do!). If you notice this, it is a very keen observation!. If fact, we are going to use the estimated θ and ϕ we obtained above using hard k-means.

2. For each biomarker n, we obtain S_n based on S. Then we iterate through all participants. If this participant is healthy, we include their biomarker measurement in `cluster_phi`. If this participant is diseased, we compare between P_θ and P_ϕ. If $S_n = 2$, then $P_\theta = 0.1+0.3 = 0.4$ and $P_\phi = 0.6$. Because P_ϕ is larger, we include this participant's biomarker measurement in `cluster_phi`. When the iteration through participants is done, we can calculate the mean and standard deviation of each cluster.

> **Tip**
>
> If $P_\theta = P_\phi$, we randomly assign this participant's biomarker measurement to a cluster.

```
def compute_single_measurement_likelihood(theta_phi, biomarker, affected, measurement):
    '''Computes the likelihood of the measurement value of a single biomarker

    We know the normal distribution defined by either theta or phi
    and we know the measurement. This will give us the probability
    of this given measurement value.

    input:
    - theta_phi: the dictionary containing theta and phi values for each biomarker
```

```
    - biomarker: an integer between 0 and 9
    - affected: boolean
    - measurement: the observed value for a biomarker in a specific participant

    output: a scalar
    '''
    biomarker_dict = theta_phi[biomarker]
    mu = biomarker_dict['theta_mean'] if affected else biomarker_dict['phi_mean']
    std = biomarker_dict['theta_std'] if affected else biomarker_dict['phi_std']
    var = std**2
    if var <= int(0) or np.isnan(measurement) or np.isnan(mu):
        print(f"Invalid values: measurement: {measurement}, mu: {mu}, var: {var}")
        likelihood = np.exp(-(measurement - mu)**2 /
                            (2 * var)) / np.sqrt(2 * np.pi * var)
    else:
        likelihood = np.exp(-(measurement - mu)**2 /
                            (2 * var)) / np.sqrt(2 * np.pi * var)
    return likelihood

def fill_up_kj_and_affected(pdata, k_j):
    '''Fill up a single participant's data using k_j; basically add two columns:
    k_j and affected
    Note that this function assumes that pdata already has the S_n column

    Input:
    - pdata: a dataframe of ten biomarker values for a specific participant
    - k_j: a scalar
    '''
    data = pdata.copy()
    data['k_j'] = k_j
    data['affected'] = data.apply(lambda row: row.k_j >= row.S_n, axis=1)
    return data

def compute_likelihood(pdata, k_j, theta_phi):
    '''
    This function computes the likelihood of seeing this sequence of biomarker values
    for a specific participant, assuming that this participant is at stage k_j
    '''
    data = fill_up_kj_and_affected(pdata, k_j)
    likelihood = 1
    for i, row in data.iterrows():
        biomarker = row['biomarker']
```

```
        measurement = row['measurement']
        affected = row['affected']
        likelihood *= compute_single_measurement_likelihood(
            theta_phi, biomarker, affected, measurement)
    return likelihood

def obtain_participants_hashmap(
        data,
        prior_theta_phi_estimates,
):
    """
    Input:
        - data: a pd.dataframe. For exrample, 150|200_3.csv
        - prior_theta_phi_estimates, a hashmap of dicts.
            This is the result from hard k-means

    Output:
        - hashmap: a dictionary whose key is participant id
            and value value is a dict whose key is stage
            and value is normalized likelihood
    """
    # initialize hashmap_of_normalized_stage_likelihood_dicts
    participants_hashmap = {}
    non_diseased_participants = data[
        data.diseased == False]['participant'].unique()
    disease_stages = data.S_n.unique()
    for p in data.participant.unique():
        dic = defaultdict(int)
        pdata = data[data.participant == p].reset_index(drop = True)
        if p in non_diseased_participants:
            dic[0] = 1
        else:
            for k_j in disease_stages:
                kj_ll = compute_likelihood(pdata, k_j, prior_theta_phi_estimates)
                dic[k_j] = kj_ll
            # likelihood sum
            sum_ll = sum(dic.values())
            epsilon = 1e-10
            if sum_ll == 0:
                sum_ll = epsilon
            normalized_lls = [l/sum_ll for l in dic.values()]
            normalized_ll_dict = dict(zip(disease_stages, normalized_lls))
```

```
            participants_hashmap[p] = normalized_ll_dict
    return participants_hashmap

def calc_soft_kmeans_for_biomarker(
        data,
        biomarker,
        participants_hashmap

):
    """obtain theta, phi estimates using soft kmeans for a single biomarker
    Inputs:
        - data: a pd.dataframe. For example, 150|200_3.csv
        - biomarker: a str, a certain biomarker name
        - hashmap: a dict, returned result of obtain_hashmap()
    Outputs:
        - theta_mean, theta_std, phi_mean, phi_std, a tuple of floats
    """
    non_diseased_participants = data[
        data.diseased == False]['participant'].unique()
    disease_stages = data.S_n.unique()
     # DataFrame for this biomarker
    biomarker_df = data[
        data['biomarker'] == biomarker].reset_index(
            drop=True).sort_values(
                by = 'participant', ascending = True)
    # Extract measurements
    measurements = np.array(biomarker_df['measurement'])

    this_biomarker_order = biomarker_df.S_n[0]

    affected_cluster = []
    non_affected_cluster = []

    for p in data.participant.unique():
        if p in non_diseased_participants:
            non_affected_cluster.append(measurements[p])
        else:
            normalized_ll_dict = participants_hashmap[p]
            affected_prob = sum(
                normalized_ll_dict[
                    kj] for kj in disease_stages if kj >= this_biomarker_order)
            non_affected_prob = sum(
```

```
                normalized_ll_dict[
                    kj] for kj in disease_stages if kj < this_biomarker_order)
            if affected_prob > non_affected_prob:
                    affected_cluster.append(measurements[p])
            elif affected_prob < non_affected_prob:
                non_affected_cluster.append(measurements[p])
            else:
                # Assign to either cluster randomly if probabilities are equal
                if np.random.random() > 0.5:
                    affected_cluster.append(measurements[p])
                else:
                    non_affected_cluster.append(measurements[p])
    # Compute means and standard deviations
    theta_mean = np.mean(affected_cluster) if affected_cluster else np.nan
    theta_std = np.std(affected_cluster) if affected_cluster else np.nan
    phi_mean = np.mean(
        non_affected_cluster) if non_affected_cluster else np.nan
    phi_std = np.std(non_affected_cluster) if non_affected_cluster else np.nan
    return theta_mean, theta_std, phi_mean, phi_std

def cal_soft_kmeans_for_biomarkers(
        data,
        participants_hashmap,
        prior_theta_phi_estimates,
):
    soft_kmeans_estimates = {}
    biomarkers = data.biomarker.unique()
    for biomarker in biomarkers:
        dic = {'biomarker': biomarker}
        prior = prior_theta_phi_estimates[biomarker]
        theta_mean, theta_std, phi_mean, phi_std = calc_soft_kmeans_for_biomarker(
            data, biomarker, participants_hashmap
        )
        if theta_std == 0 or math.isnan(theta_std):
            theta_mean = prior['theta_mean']
            theta_std = prior['theta_std']
        if phi_std == 0 or math.isnan(phi_std):
            phi_mean = prior['phi_mean']
            phi_std = prior['phi_std']
        dic['theta_mean'] = theta_mean
        dic['theta_std'] = theta_std
        dic['phi_mean'] = phi_mean
```

```
        dic['phi_std'] = phi_std
        soft_kmeans_estimates[biomarker] = dic
    return soft_kmeans_estimates
```

```
participants_hashmap = obtain_participants_hashmap(
    data = df,
    prior_theta_phi_estimates = hard_kmeans_estimates,
)

soft_kmeans_estimates = cal_soft_kmeans_for_biomarkers(
        data = df,
        participants_hashmap = participants_hashmap,
        prior_theta_phi_estimates = hard_kmeans_estimates,
)
```

```
soft_kmeans_estimates_df = pd.DataFrame.from_dict(
    soft_kmeans_estimates, orient='index')
soft_kmeans_estimates_df.reset_index(drop=True, inplace=True)
soft_kmeans_estimates_df
```

	biomarker	theta_mean	theta_std	phi_mean	phi_std
0	HIP-FCI	-5.378366	7.232544	5.092800	1.514369
1	PCC-FCI	5.710289	2.864833	12.098428	3.688180
2	AB	153.648672	18.041801	253.189546	51.038112
3	P-Tau	-42.235505	34.734178	-24.626343	14.860556
4	MMSE	23.122406	2.445751	28.049683	0.718480
5	ADAS	-18.374956	5.810630	-5.889272	1.283121
6	HIP-GMI	0.442551	0.261505	0.373983	0.234485
7	AVLT-Sum	23.819301	6.942910	41.371798	14.381565
8	FUS-GMI	0.496420	0.052929	0.595958	0.060867
9	FUS-FCI	-6.326653	1.695438	-10.205164	3.733813

```
make_chart(
    biomarkers[0:5],
    soft_kmeans_estimates_df,
    truth_df,
    title = "Comparing Theta and Phi Distributions Using Soft K-Means"
)
```

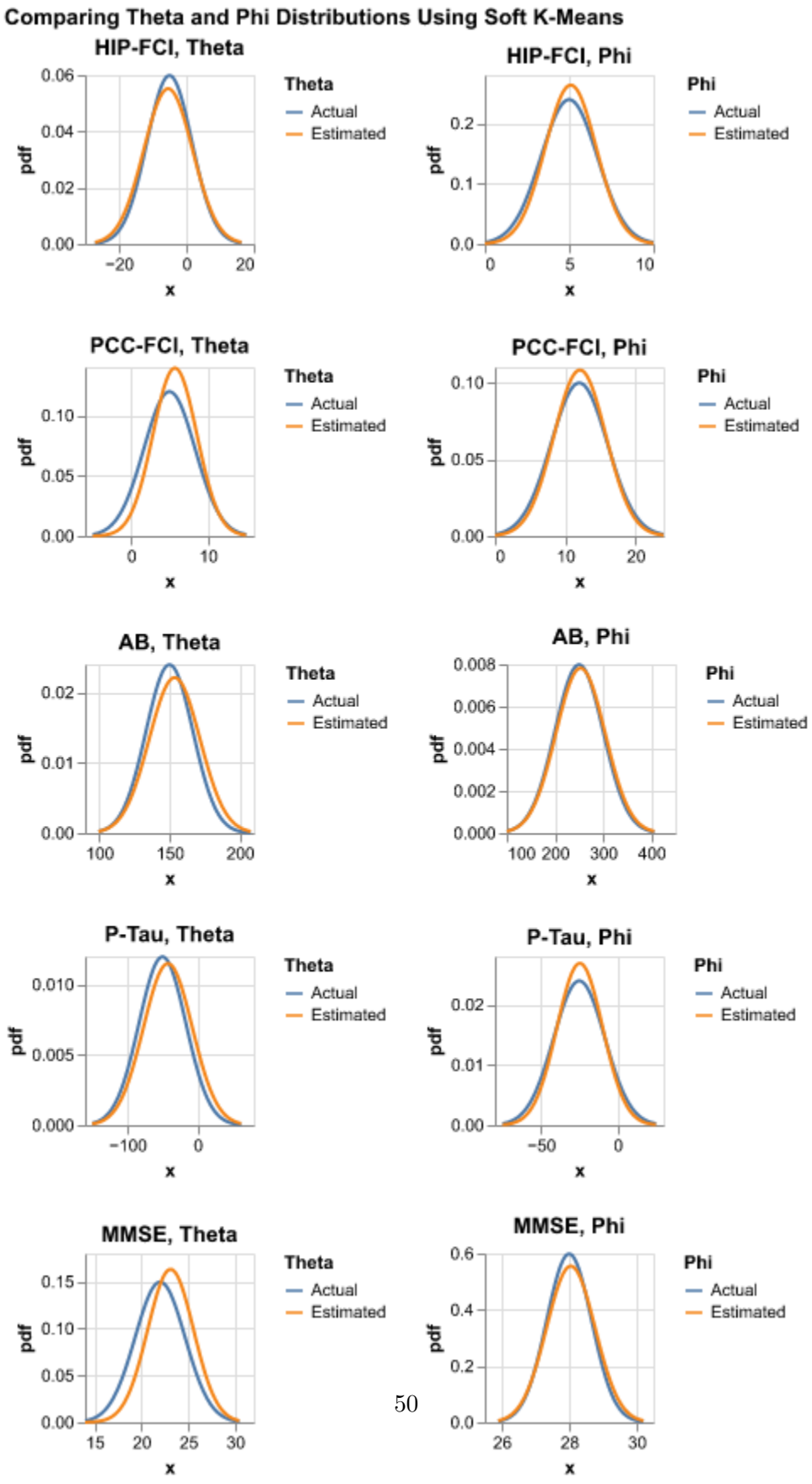


Figure 4.3: Comparing Theta and Phi Distributions Using Soft K-Mean

4.4 Conclusion

We compare the above three methods. Hard k-means has the least number of prerequisites: it only needs to know whether participants are healthy or not and biomarker measurements. However, the drawback is that it might not be very accurate. Conjugate priors are extremely accurate; however, it requires knowledge of almost everything: besides what is required by hard k-means, it also requires S and k_j. Soft k-kmeans does not require the knowledge of k_j and is an improvement over hard k-means.

We also noticed that both conjugate priors and soft k-means need to use the result from hard k-means as a fallback.

5 Estimate Participant Stages

In this chapter, we will do some exercise to have a deeper understanding of the math equations in Section 2.3.

5.1 Challenge

Suppos we know S, θ, ϕ. How could we estimate participant stages?

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json
from collections import Counter
```

This is the data we have. And we want to know fill the missing column of `k_j`.

```
output_dir = 'data'
df = pd.read_csv(f"{output_dir}/100|200_3.csv")
real_stages_dic = dict(zip(df.participant, df.k_j))
df.drop(['k_j', 'affected_or_not'], axis = 1, inplace=True)
df.head()
```

	participant	biomarker	measurement	S_n	diseased
0	0	HIP-FCI	-8.908479	1	True
1	1	HIP-FCI	-1.095464	1	False
2	2	HIP-FCI	0.470754	1	True
3	3	HIP-FCI	2.633455	1	True
4	4	HIP-FCI	4.070208	1	False

5.2 Solution

One possible solution looks like this:

- For each diseased participant, we iterate through all possible disease stages and calculate the likelihood using Equation 2.1.
- We normalize all the likelihoods, construct an array, and randomly sample one possible stage according to that array.
- Run multiple times, for each diseased participant, the mode of the sampled stages will be their stage.

```
def compute_single_measurement_likelihood(
        theta_phi,
        biomarker,
        affected,
        measurement):
    '''Computes the likelihood of the measurement value of a single biomarker

    We know the normal distribution defined by either theta or phi
    and we know the measurement. This will give us the probability
    of this given measurement value.

    input:
    - theta_phi: the dictionary containing theta and phi values for each biomarker
    - biomarker: an integer between 0 and 9
    - affected: boolean
    - measurement: the observed value for a biomarker in a specific participant

    output: a scalar
    '''
    biomarker_dict = theta_phi[biomarker]
    mu = biomarker_dict['theta_mean'] if affected else biomarker_dict['phi_mean']
    std = biomarker_dict['theta_std'] if affected else biomarker_dict['phi_std']
    var = std**2
    if var <= int(0) or np.isnan(measurement) or np.isnan(mu):
        print(f"Invalid values: measurement: {measurement}, mu: {mu}, var: {var}")
        likelihood = np.exp(-(measurement - mu)**2 /
                            (2 * var)) / np.sqrt(2 * np.pi * var)
    else:
        likelihood = np.exp(-(measurement - mu)**2 /
                            (2 * var)) / np.sqrt(2 * np.pi * var)
```

```
    return likelihood

def fill_up_kj_and_affected(pdata, k_j):
    '''Fill up a single participant's data using k_j; basically add two columns:
    k_j and affected
    Note that this function assumes that pdata already has the S_n column

    Input:
    - pdata: a dataframe of ten biomarker values for a specific participant
    - k_j: a scalar
    '''
    data = pdata.copy()
    data['k_j'] = k_j
    data['affected'] = data.apply(lambda row: row.k_j >= row.S_n, axis=1)
    return data

def compute_likelihood(pdata, k_j, theta_phi):
    '''
    This function computes the likelihood of seeing this sequence of biomarker values
    for a specific participant, assuming that this participant is at stage k_j
    '''
    data = fill_up_kj_and_affected(pdata, k_j)
    likelihood = 1
    for i, row in data.iterrows():
        biomarker = row['biomarker']
        measurement = row['measurement']
        affected = row['affected']
        likelihood *= compute_single_measurement_likelihood(
            theta_phi, biomarker, affected, measurement)
    return likelihood
```

We first look at the known θ, ϕ:

```
with open('files/real_theta_phi.json', 'r') as f:
    truth = json.load(f)
truth_df = pd.DataFrame.from_dict(truth, orient='index')
truth_df.reset_index(names = 'biomarker', inplace=True)
truth_df
```

	biomarker	theta_mean	theta_std	phi_mean	phi_std
0	MMSE	22.0	2.666667	28.0	0.666667
1	ADAS	-20.0	4.000000	-6.0	1.333333
2	AB	150.0	16.666667	250.0	50.000000
3	P-Tau	-50.0	33.333333	-25.0	16.666667
4	HIP-FCI	-5.0	6.666667	5.0	1.666667
5	HIP-GMI	0.3	0.333333	0.4	0.233333
6	AVLT-Sum	20.0	6.666667	40.0	15.000000
7	PCC-FCI	5.0	3.333333	12.0	4.000000
8	FUS-GMI	0.5	0.066667	0.6	0.066667
9	FUS-FCI	-20.0	6.000000	-10.0	3.333333

5.3 Implementation

We then implement the algorithm mentioned above:

```
theta_phi_estimates = truth.copy()
disease_stages = df.S_n.unique()
diseased_participants = df[df.diseased==True]['participant'].unique()
```

```
def update_participant_stages_dic(
        data,
        p,
        disease_stages,
        theta_phi_estimates,
        # participant stage dic:
        psdic,
        sample_iterations = 20
):
    """
    Inputs:
        - data: pd.dataframe, e.g., 100|200_3.csv
        - p: int
        - disease_stages: a list of integers
        - theta_phi_estimates: a hashmap of dictionaries
        - psdic: a dictionary
        - sample_iteration: int. How many times we sample
    Output:
        no outputs. Simply update psdic
    """
```

```
    pdata = data[data.participant == p]
    stage_likelihood_dict = {}
    for k_j in disease_stages:
        kj_likelihood = compute_likelihood(
            pdata, k_j, theta_phi_estimates)
        # update each stage likelihood for this participant
        stage_likelihood_dict[k_j] = kj_likelihood
    # Add a small epsilon to avoid division by zero
    likelihood_sum = sum(stage_likelihood_dict.values())
    epsilon = 1e-10
    if likelihood_sum == 0:
        # print("Invalid likelihood_sum: zero encountered.")
        likelihood_sum = epsilon  # Handle the case accordingly
    normalized_stage_likelihood = [
        l/likelihood_sum for l in stage_likelihood_dict.values()]
    sampled_stages = np.random.choice(
        disease_stages,
        size = sample_iterations,
        p=normalized_stage_likelihood,
        replace=True
    )
    mode_result = Counter(sampled_stages).most_common(1)[0][0]
    psdic[p] = mode_result
```

```
participants = df.participant.unique()
psdic = {}
for p in participants:
    if p not in diseased_participants:
        psdic[p] = 0
    else:
        update_participant_stages_dic(
            df,
            p,
            disease_stages,
            theta_phi_estimates,
            # participant stage dic:
            psdic,
            sample_iterations = 10
    )
```

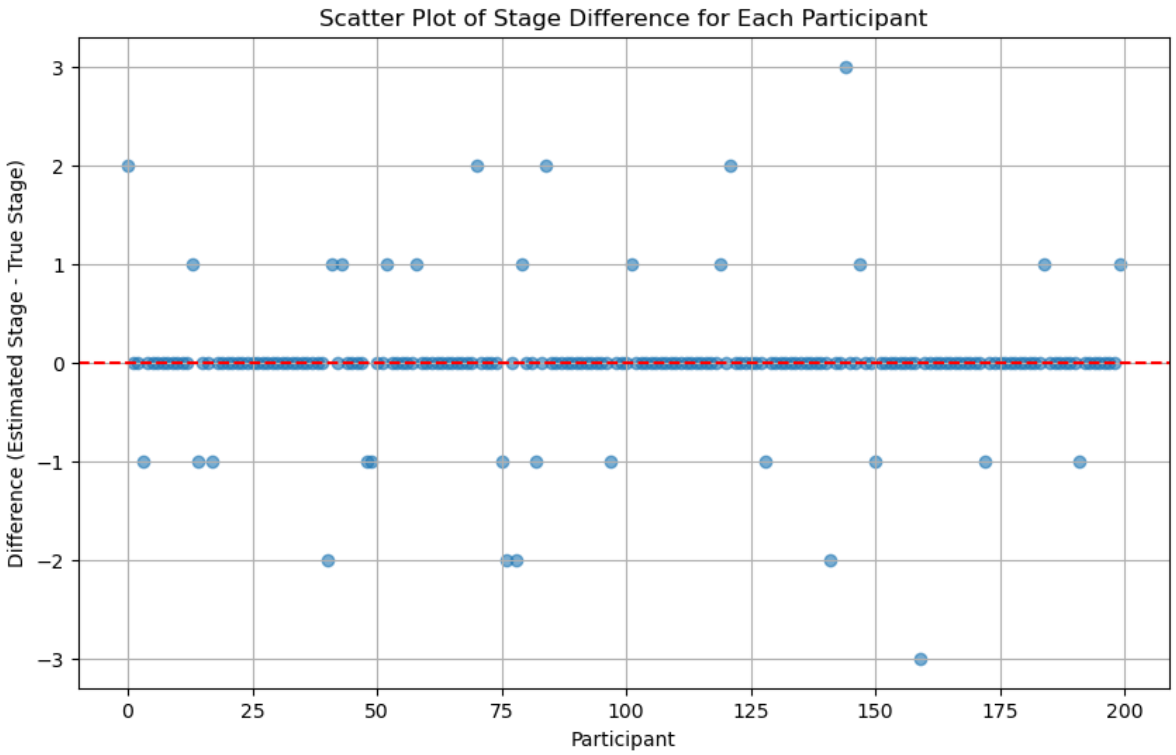
5.4 Result

Then we compare our results with the actual participants' stages:

```
diff = np.array(list(psdic.values())) - np.array(list(real_stages_dic.values()))
```

```
def scatter_plot_of_stage_differences(stage_differences):
    '''Scatter Plot of the Difference at each index
    Input:
    - stage_differences: estimated_stages - actual stages. Result should be a 1-dim np array
    '''
    plt.figure(figsize=(10, 6))
    plt.scatter(range(len(diff)), stage_differences, alpha=0.6)
    plt.axhline(y=0, color='red', linestyle='--')
    plt.title("Scatter Plot of Stage Difference for Each Participant")
    plt.xlabel("Participant")
    plt.ylabel("Difference (Estimated Stage - True Stage)")
    plt.grid(True)
    plt.show()
```

```
scatter_plot_of_stage_differences(diff)
```



5.5 Discussion

From the above result, we can see how challenging it is to accurately estimate participant stages, **even if** we know exactly the θ and ϕ.

> **Tip**
>
> What if we know only S, but not θ or ϕ?
> The first step for us is to estimate θ, ϕ and then follow the above procedures. To do that, refer back to Chapter 4.

6 Estimate S

Now we come to the core part of our project: how can we estimate S, without knowing θ, ϕ, k_j?

Basically, this is the data we have:

```
import pandas as pd
output_dir = 'data'
df = pd.read_csv(f"{output_dir}/150|200_3.csv").drop(
    ['k_j', 'S_n', 'affected_or_not'], axis = 1)
df.head()
```

	participant	biomarker	measurement	diseased
0	0	HIP-FCI	3.135981	False
1	1	HIP-FCI	12.593704	True
2	2	HIP-FCI	6.220776	False
3	3	HIP-FCI	3.545100	False
4	4	HIP-FCI	3.966541	False

The main idea is this:

- We need to know θ, ϕ first before we can estimate likelihoods. To estimate θ, ϕ, there are three approaches covered in Chapter 4.
- We try many different S and calculate its associated likelihood. We either accept or reject this S according to Metropolis–Hastings algorithm.

In the following, We will cover three different approaches to estimate S:

- Hard K-Means
- Soft K-Means
- Conjugate Priors

7 Estimate S with Hard KMeans

The basic idea of using hard K Means to estimate S is:

- We first estimate distribution parameters using hard K Means, the exact procedure we covered in Section 4.1.
- We use Metropolis–Hastings algorithm to accept or reject a proposed S.

Algorithm 1 Metropolis-Hastings Hard K-Means

Input: data with participant id, biomarker, and its measurement; n_{shuffle}: number of elements to shuffle
Output: all_accepted_orders
$\theta, \phi \leftarrow$ get_theta_phi_using_hard_kmeans()
all_accepted_orders $\leftarrow$ []
current_order $\leftarrow$ permutation(np.arange(1, n_{stages}))
current_ll $\leftarrow -\infty$
for i in range(iterations) **do**
    new_order $\leftarrow$ current_order.copy()
    shuffle_order(new_order, n_{shuffle})
    all_ln_ll $\leftarrow$ compute_ll(data, theta, phi, new_order)
    prob_of_accepting_new_order $\leftarrow$ exp(all_ln_ll − current_ll)
    if np.random.rand() < prob_of_accepting_new_order **then**
        current_order $\leftarrow$ new_order
        current_ll $\leftarrow$ all_ln_ll
        all_accepted_orders.append(current_order)
    end if
end for
return all_accepted_orders

Figure 7.1: Hard K-Means Algorithm

7.1 Implementation

```
import numpy as np
import utils
import json
import pandas as pd
import utils
from scipy.stats import kendalltau
import sys
import os
```

```
def calculate_all_participant_ln_likelihood(
        iteration,
        data_we_have,
        current_order_dict,
        n_participants,
        non_diseased_participant_ids,
        theta_phi_estimates,
        diseased_stages,
):
    data = data_we_have.copy()
    data['S_n'] = data.apply(
        lambda row: current_order_dict[row['biomarker']], axis=1)
    all_participant_ln_likelihood = 0

    for p in range(n_participants):
        pdata = data[data.participant == p].reset_index(drop=True)
        if p in non_diseased_participant_ids:
            this_participant_likelihood = utils.compute_likelihood(
                pdata, k_j=0, theta_phi=theta_phi_estimates)
            this_participant_ln_likelihood = np.log(
                this_participant_likelihood + 1e-10)
        else:
            # normalized_stage_likelihood_dict = None
            # initiaze stage_likelihood
            stage_likelihood_dict = {}
            for k_j in diseased_stages:
                kj_likelihood = utils.compute_likelihood(
                    pdata, k_j, theta_phi_estimates)
                # update each stage likelihood for this participant
                stage_likelihood_dict[k_j] = kj_likelihood
```

```
            # Add a small epsilon to avoid division by zero
            likelihood_sum = sum(stage_likelihood_dict.values())

            # calculate weighted average
            this_participant_likelihood = np.mean(likelihood_sum)
            this_participant_ln_likelihood = np.log(
                this_participant_likelihood + 1e-10)
        all_participant_ln_likelihood += this_participant_ln_likelihood
    return all_participant_ln_likelihood

def metropolis_hastings_hard_kmeans(
    data_we_have,
    iterations,
    n_shuffle,
):
    '''Implement the metropolis-hastings algorithm using simple clustering
    Inputs:
        - data: data_we_have
        - iterations: number of iterations
        - log_folder_name: the folder where log files locate

    Outputs:
        - best_order: a numpy array
        - best_likelihood: a scalar
    '''
    n_participants = len(data_we_have.participant.unique())
    biomarkers = data_we_have.biomarker.unique()
    n_biomarkers = len(biomarkers)
    n_stages = n_biomarkers + 1
    non_diseased_participant_ids = data_we_have.loc[
        data_we_have.diseased == False].participant.unique()
    diseased_stages = np.arange(start=1, stop=n_stages, step=1)
    # obtain the iniial theta and phi estimates
    theta_phi_estimates = utils.get_theta_phi_estimates(
        data_we_have)

    # initialize empty lists
    acceptance_count = 0
    all_current_accepted_order_dicts = []

    current_accepted_order = np.random.permutation(np.arange(1, n_stages))
    current_accepted_order_dict = dict(zip(biomarkers, current_accepted_order))
```

```
    current_accepted_likelihood = -np.inf

    for _ in range(iterations):
        # in each iteration, we have updated current_order_dict and theta_phi_estimates

        new_order = current_accepted_order.copy()
        utils.shuffle_order(new_order, n_shuffle)
        current_order_dict = dict(zip(biomarkers, new_order))
        all_participant_ln_likelihood = calculate_all_participant_ln_likelihood(
                _,
                data_we_have,
                current_order_dict,
                n_participants,
                non_diseased_participant_ids,
                theta_phi_estimates,
                diseased_stages,
            )


        # Log-Sum-Exp Trick
        max_likelihood = max(all_participant_ln_likelihood,
                             current_accepted_likelihood)
        prob_of_accepting_new_order = np.exp(
            (all_participant_ln_likelihood - max_likelihood) -
            (current_accepted_likelihood - max_likelihood)
        )

        # prob_of_accepting_new_order = np.exp(
        #     all_participant_ln_likelihood - current_accepted_likelihood)

        # np.exp(a)/np.exp(b) = np.exp(a - b)
        # if a > b, then np.exp(a - b) > 1

        # it will definitly update at the first iteration
        if np.random.rand() < prob_of_accepting_new_order:
            acceptance_count += 1
            current_accepted_order = new_order
            current_accepted_likelihood = all_participant_ln_likelihood
            current_accepted_order_dict = current_order_dict

        acceptance_ratio = acceptance_count*100/(_+1)
        all_current_accepted_order_dicts.append(current_accepted_order_dict)
```

```
        if (_+1) % 10 == 0:
            formatted_string = (
                f"iteration {_ + 1} done, "
                f"current accepted likelihood: {current_accepted_likelihood}, "
                f"current acceptance ratio is {acceptance_ratio:.2f} %, "
                f"current accepted order is {current_accepted_order_dict.values()}, "
            )
            print(formatted_string)

    # print("done!")
    return all_current_accepted_order_dicts
```

```
n_shuffle = 2
iterations = 10
burn_in = 2
thining = 2

base_dir = os.getcwd()
print(f"Current working directory: {base_dir}")
data_dir = os.path.join(base_dir, "data")

cop_kmeans_dir = os.path.join(base_dir, 'hard_kmeans')
temp_results_dir = os.path.join(cop_kmeans_dir, "temp_json_results")
img_dir = os.path.join(cop_kmeans_dir, 'img')
results_file = os.path.join(cop_kmeans_dir, "results.json")

os.makedirs(cop_kmeans_dir, exist_ok=True)
os.makedirs(temp_results_dir, exist_ok=True)
os.makedirs(img_dir, exist_ok=True)

print(f"Data directory: {data_dir}")
print(f"Temp results directory: {temp_results_dir}")
print(f"Image directory: {img_dir}")

if __name__ == "__main__":

    # Read parameters from command line arguments
    j = 200
    r = 0.75
    m = 3
```

```
    print(f"Processing with j={j}, r={r}, m={m}")

    combstr = f"{int(j*r)}|{j}"
    heatmap_folder = img_dir

    img_filename = f"{int(j*r)}-{j}_{m}"
    filename = f"{combstr}_{m}"
    data_file = f"{data_dir}/{filename}.csv"
    data_we_have = pd.read_csv(data_file)
    n_biomarkers = len(data_we_have.biomarker.unique())

    if not os.path.exists(data_file):
        print(f"Data file not found: {data_file}")
        sys.exit(1)  # Exit early if the file doesn't exist
    else:
        print(f"Data file found: {data_file}")

    # Define the temporary result file
    temp_result_file = os.path.join(temp_results_dir, f"temp_results_{j}_{r}_{m}.json")

    dic = {}

    if combstr not in dic:
        dic[combstr] = []

    accepted_order_dicts = metropolis_hastings_hard_kmeans(
        data_we_have,
        iterations,
        n_shuffle,
    )

    utils.save_heatmap(
        accepted_order_dicts,
        burn_in,
        thining,
        folder_name=heatmap_folder,
        file_name=f"{img_filename}",
        title=f'heatmap of {filename}')

    most_likely_order_dic = utils.obtain_most_likely_order_dic(
        accepted_order_dicts, burn_in, thining)
    most_likely_order = list(most_likely_order_dic.values())
```

```
    tau, p_value = kendalltau(most_likely_order, range(1, n_biomarkers + 1))

    dic[combstr].append(tau)

    # Write the results to a unique temporary file inside the temp folder
    with open(temp_result_file, "w") as file:
        json.dump(dic, file, indent=4)
    print(f"{filename} is done! Results written to {temp_result_file}")
```

```
Current working directory: /Users/hongtaoh/Desktop/github/ebmBook
Data directory: /Users/hongtaoh/Desktop/github/ebmBook/data
Temp results directory: /Users/hongtaoh/Desktop/github/ebmBook/hard_kmeans/temp_json_results
Image directory: /Users/hongtaoh/Desktop/github/ebmBook/hard_kmeans/img
Processing with j=200, r=0.75, m=3
Data file found: /Users/hongtaoh/Desktop/github/ebmBook/data/150|200_3.csv
iteration 10 done, current accepted likelihood: -4491.553963056519, current acceptance ratio
150|200_3 is done! Results written to /Users/hongtaoh/Desktop/github/ebmBook/hard_kmeans/temp
```

7.2 Result

We plot the resulting S probalistically using a heatmap. We also quantify the difference between our result with the real S using Kendall's Tau. It ranges from -1 (completely different) to 1 (exactly the same). 0 indicate complete randomness.

```
dic
```

```
{'150|200': [0.3333333333333333]}
```

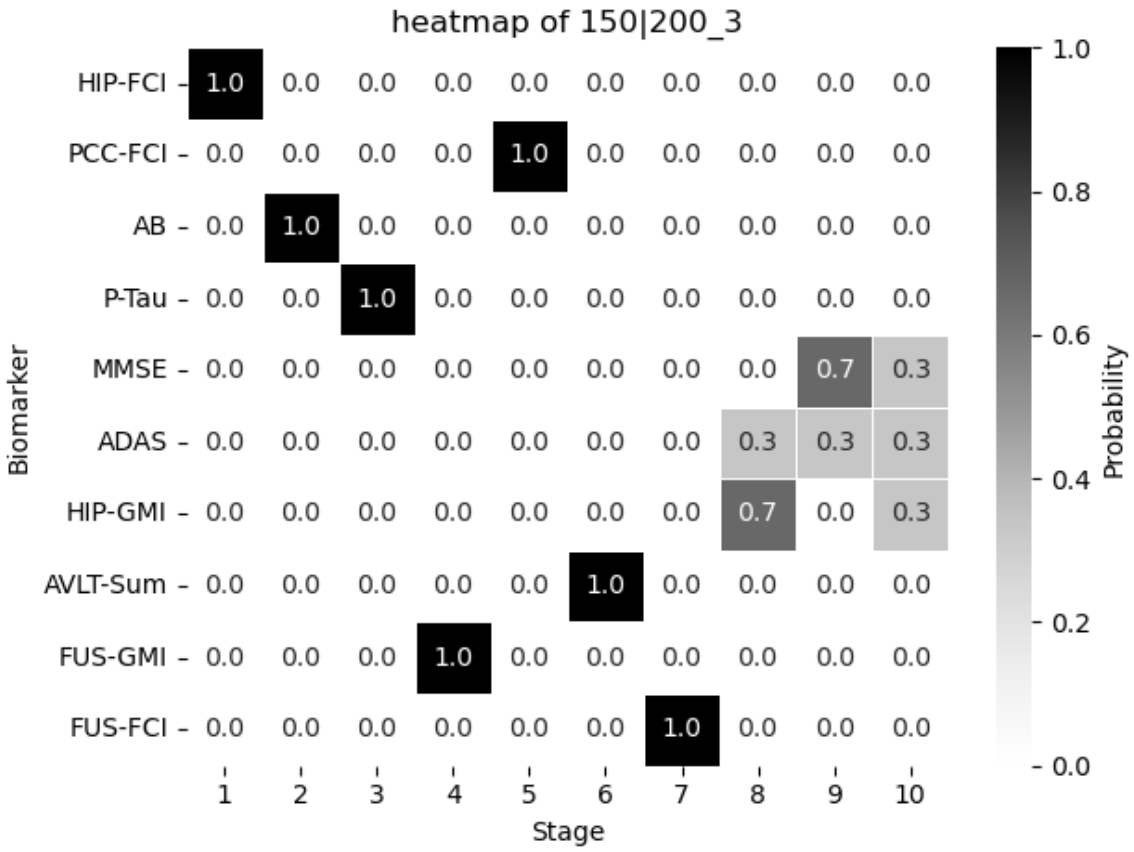


Figure 7.2: Result of Hard K-Means

8 Estimate S with Soft K-Means

The basic idea of using Soft K-Means to estimate S is:

- We first estimate distribution parameters using hard K-Means, the exact procedure we covered in Section 4.1.
- In each iteration, we use Soft Kmeans algorithm to update θ, ϕ (refer to Section 4.3) and use Metropolis–Hastings algorithm to accept or reject a proposed S.

Algorithm 2 Metropolis-Hastings Soft K-Means

Input: data with participant id, biomarker, and its measurement; n_{shuffle}: number of elements to shuffle
Output: all_accepted_orders
$\theta, \phi \leftarrow$ get_theta_phi_using_clustering()
all_accepted_orders $\leftarrow$ []
current_order $\leftarrow$ permutation(np.arange(1, n_{stages}))
current_ll $\leftarrow -\infty$
for i in range(iterations) **do**
    new_order $\leftarrow$ current_order.copy()
    shuffle_order(new_order, n_{shuffle})
    (all_ln_ll, hashmap_of_marginalized_ll_dics) $\leftarrow f$(data, θ, ϕ, new_order)
    prob_of_accepting_new_order $\leftarrow$ exp(all_ln_ll − current_ll)
    if np.random.rand() < prob_of_accepting_new_order **then**
        current_order $\leftarrow$ new_order
        current_ll $\leftarrow$ all_ln_ll
        all_accepted_orders.append(current_order)
    end if
    $\theta, \phi \leftarrow$ theta_phi_soft_kmeans(data, hashmap_of_marginalized_ll_dics)
end for
return all_accepted_orders

Figure 8.1: Soft K-Means Algorithm

8.1 Implementation

```
import numpy as np
import utils
import json
import pandas as pd
import utils
from scipy.stats import kendalltau
import sys
import os
import math
```

```
def calculate_soft_kmeans_for_biomarker(
        data,
        biomarker,
        order_dict,
        n_participants,
        non_diseased_participants,
        hashmap_of_normalized_stage_likelihood_dicts,
        diseased_stages,
        seed=None
):
    """
    Calculate mean and std for both the affected and non-affected clusters for a single bioma

    Parameters:
        data (pd.DataFrame): The data containing measurements.
        biomarker (str): The biomarker to process.
        order_dict (dict): Dictionary mapping biomarkers to their order.
        n_participants (int): Number of participants in the study.
        non_diseased_participants (list): List of non-diseased participants.
        hashmap_of_normalized_stage_likelihood_dicts (dict): Hash map of
            dictionaries containing stage likelihoods for each participant.
        diseased_stages (list): List of diseased stages.
        seed (int, optional): Random seed for reproducibility.

    Returns:
        tuple: Means and standard deviations for affected and non-affected clusters.
    """
    if seed is not None:
        # Set the seed for numpy's random number generator
```

```
        rng = np.random.default_rng(seed)
    else:
        rng = np.random

    # DataFrame for this biomarker
    biomarker_df = data[
        data['biomarker'] == biomarker].reset_index(
            drop=True).sort_values(
                by = 'participant', ascending = True)
    # Extract measurements
    measurements = np.array(biomarker_df['measurement'])

    this_biomarker_order = order_dict[biomarker]

    affected_cluster = []
    non_affected_cluster = []

    for p in range(n_participants):
        if p in non_diseased_participants:
            non_affected_cluster.append(measurements[p])
        else:
            if this_biomarker_order == 1:
                affected_cluster.append(measurements[p])
            else:
                normalized_stage_likelihood_dict = hashmap_of_normalized_stage_likelihood_di
                    p]
                # Calculate probabilities for affected and non-affected states
                affected_prob = sum(
                    normalized_stage_likelihood_dict[s] for s in diseased_stages if s >= this
                )
                non_affected_prob = sum(
                    normalized_stage_likelihood_dict[s] for s in diseased_stages if s < this_
                )
                if affected_prob > non_affected_prob:
                    affected_cluster.append(measurements[p])
                elif affected_prob < non_affected_prob:
                    non_affected_cluster.append(measurements[p])
                else:
                    # Assign to either cluster randomly if probabilities are equal
                    if rng.random() > 0.5:
                        affected_cluster.append(measurements[p])
                    else:
```

```
                        non_affected_cluster.append(measurements[p])

    # Compute means and standard deviations
    theta_mean = np.mean(affected_cluster) if affected_cluster else np.nan
    theta_std = np.std(affected_cluster) if affected_cluster else np.nan
    phi_mean = np.mean(
        non_affected_cluster) if non_affected_cluster else np.nan
    phi_std = np.std(non_affected_cluster) if non_affected_cluster else np.nan
    return theta_mean, theta_std, phi_mean, phi_std

def soft_kmeans_theta_phi_estimates(
        iteration,
        prior_theta_phi_estimates,
        data_we_have,
        biomarkers,
        order_dict,
        n_participants,
        non_diseased_participants,
        hashmap_of_normalized_stage_likelihood_dicts,
        diseased_stages,
        seed=None):
    """
    Get the DataFrame of theta and phi using the soft K-means algorithm for all biomarkers.

    Parameters:
        data_we_have (pd.DataFrame): DataFrame containing the data.
        biomarkers (list): List of biomarkers in string.
        order_dict (dict): Dictionary mapping biomarkers to their order.
        n_participants (int): Number of participants in the study.
        non_diseased_participants (list): List of non-diseased participants.
        hashmap_of_normalized_stage_likelihood_dicts (dict): Hash map of dictionaries contai
        diseased_stages (list): List of diseased stages.
        seed (int, optional): Random seed for reproducibility.

    Returns:
        a dictionary containing the means and standard deviations for theta and phi for each
    """
    # List of dicts to store the estimates
    # In each dic, key is biomarker, and values are theta and phi params
    hashmap_of_means_stds_estimate_dicts = {}
    for biomarker in biomarkers:
        dic = {'biomarker': biomarker}
```

```
        prior_theta_phi_estimates_biomarker = prior_theta_phi_estimates[biomarker]
        theta_mean, theta_std, phi_mean, phi_std = calculate_soft_kmeans_for_biomarker(
            data_we_have,
            biomarker,
            order_dict,
            n_participants,
            non_diseased_participants,
            hashmap_of_normalized_stage_likelihood_dicts,
            diseased_stages,
            seed
        )
        if theta_std == 0 or math.isnan(theta_std):
            theta_mean = prior_theta_phi_estimates_biomarker['theta_mean']
            theta_std = prior_theta_phi_estimates_biomarker['theta_std']
        if phi_std == 0 or math.isnan(phi_std):
            phi_mean = prior_theta_phi_estimates_biomarker['phi_mean']
            phi_std = prior_theta_phi_estimates_biomarker['phi_std']
        dic['theta_mean'] = theta_mean
        dic['theta_std'] = theta_std
        dic['phi_mean'] = phi_mean
        dic['phi_std'] = phi_std
        hashmap_of_means_stds_estimate_dicts[biomarker] = dic
    return hashmap_of_means_stds_estimate_dicts

def calculate_all_participant_ln_likelihood_and_update_hashmap(
        iteration,
        data_we_have,
        current_order_dict,
        n_participants,
        non_diseased_participant_ids,
        theta_phi_estimates,
        diseased_stages,
):
    data = data_we_have.copy()
    data['S_n'] = data.apply(
        lambda row: current_order_dict[row['biomarker']], axis=1)
    all_participant_ln_likelihood = 0
    # key is participant id
    # value is normalized_stage_likelihood_dict
    hashmap_of_normalized_stage_likelihood_dicts = {}
    for p in range(n_participants):
        pdata = data[data.participant == p].reset_index(drop=True)
```

```
        if p in non_diseased_participant_ids:
            this_participant_likelihood = utils.compute_likelihood(
                pdata, k_j=0, theta_phi=theta_phi_estimates)
            this_participant_ln_likelihood = np.log(
                this_participant_likelihood + 1e-10)
        else:
            # normalized_stage_likelihood_dict = None
            # initiaze stage_likelihood
            stage_likelihood_dict = {}
            for k_j in diseased_stages:
                kj_likelihood = utils.compute_likelihood(
                    pdata, k_j, theta_phi_estimates)
                # update each stage likelihood for this participant
                stage_likelihood_dict[k_j] = kj_likelihood
            # Add a small epsilon to avoid division by zero
            likelihood_sum = sum(stage_likelihood_dict.values())
            epsilon = 1e-10
            if likelihood_sum == 0:
                # print("Invalid likelihood_sum: zero encountered.")
                likelihood_sum = epsilon  # Handle the case accordingly
            normalized_stage_likelihood = [
                l/likelihood_sum for l in stage_likelihood_dict.values()]
            normalized_stage_likelihood_dict = dict(
                zip(diseased_stages, normalized_stage_likelihood))
            hashmap_of_normalized_stage_likelihood_dicts[p] = normalized_stage_likelihood_di

            # calculate weighted average
            this_participant_likelihood = np.mean(likelihood_sum)
            this_participant_ln_likelihood = np.log(
                this_participant_likelihood)
        all_participant_ln_likelihood += this_participant_ln_likelihood
    return all_participant_ln_likelihood, hashmap_of_normalized_stage_likelihood_dicts


def metropolis_hastings_soft_kmeans(
    data_we_have,
    iterations,
    n_shuffle,
):
    '''Implement the metropolis-hastings algorithm using soft kmeans
    Inputs:
        - data: data_we_have
```

```
        - iterations: number of iterations
        - log_folder_name: the folder where log files locate

    Outputs:
        - best_order: a numpy array
        - best_likelihood: a scalar
    '''
    n_participants = len(data_we_have.participant.unique())
    biomarkers = data_we_have.biomarker.unique()
    n_biomarkers = len(biomarkers)
    n_stages = n_biomarkers + 1
    non_diseased_participant_ids = data_we_have.loc[
        data_we_have.diseased == False].participant.unique()
    diseased_stages = np.arange(start=1, stop=n_stages, step=1)
    # obtain the iniial theta and phi estimates
    prior_theta_phi_estimates = utils.get_theta_phi_estimates(
        data_we_have)
    theta_phi_estimates = prior_theta_phi_estimates.copy()

    # initialize empty lists
    acceptance_count = 0
    all_current_accepted_order_dicts = []

    current_accepted_order = np.random.permutation(np.arange(1, n_stages))
    current_accepted_order_dict = dict(zip(biomarkers, current_accepted_order))
    current_accepted_likelihood = -np.inf

    for _ in range(iterations):
        # in each iteration, we have updated current_order_dict and theta_phi_estimates

        new_order = current_accepted_order.copy()
        utils.shuffle_order(new_order, n_shuffle)
        current_order_dict = dict(zip(biomarkers, new_order))
        all_participant_ln_likelihood, \
            hashmap_of_normalized_stage_likelihood_dicts = calculate_all_participant_ln_likel
                _,
                data_we_have,
                current_order_dict,
                n_participants,
                non_diseased_participant_ids,
                theta_phi_estimates,
                diseased_stages,
```

```
        )

    # Now, update theta_phi_estimates using soft kmeans
    # based on the updated hashmap of normalized stage likelihood dicts
    theta_phi_estimates = soft_kmeans_theta_phi_estimates(
        _,
        prior_theta_phi_estimates,
        data_we_have,
        biomarkers,
        current_order_dict,
        n_participants,
        non_diseased_participant_ids,
        hashmap_of_normalized_stage_likelihood_dicts,
        diseased_stages,
        seed=None,
    )

    # Log-Sum-Exp Trick
    max_likelihood = max(all_participant_ln_likelihood,
                         current_accepted_likelihood)
    prob_of_accepting_new_order = np.exp(
        (all_participant_ln_likelihood - max_likelihood) -
        (current_accepted_likelihood - max_likelihood)
    )

    # prob_of_accepting_new_order = np.exp(
    #     all_participant_ln_likelihood - current_accepted_likelihood)

    # np.exp(a)/np.exp(b) = np.exp(a - b)
    # if a > b, then np.exp(a - b) > 1

    # it will definitly update at the first iteration
    if np.random.rand() < prob_of_accepting_new_order:
        acceptance_count += 1
        current_accepted_order = new_order
        current_accepted_likelihood = all_participant_ln_likelihood
        current_accepted_order_dict = current_order_dict

    acceptance_ratio = acceptance_count*100/(_+1)
    all_current_accepted_order_dicts.append(current_accepted_order_dict)

    if (_+1) % 10 == 0:
```

```
            formatted_string = (
                f"iteration {_ + 1} done, "
                f"current accepted likelihood: {current_accepted_likelihood}, "
                f"current acceptance ratio is {acceptance_ratio:.2f} %, "
                f"current accepted order is {current_accepted_order_dict.values()}, "
            )
            print(formatted_string)

    # print("done!")
    return all_current_accepted_order_dicts
```

```
n_shuffle = 2
iterations = 10
burn_in = 2
thining = 2

base_dir = os.getcwd()
print(f"Current working directory: {base_dir}")
data_dir = os.path.join(base_dir, "data")

soft_kmeans_dir = os.path.join(base_dir, 'soft_kmeans')
temp_results_dir = os.path.join(soft_kmeans_dir, "temp_json_results")
img_dir = os.path.join(soft_kmeans_dir, 'img')
results_file = os.path.join(soft_kmeans_dir, "results.json")

os.makedirs(soft_kmeans_dir, exist_ok=True)
os.makedirs(temp_results_dir, exist_ok=True)
os.makedirs(img_dir, exist_ok=True)

print(f"Data directory: {data_dir}")
print(f"Temp results directory: {temp_results_dir}")
print(f"Image directory: {img_dir}")

if __name__ == "__main__":

    # Read parameters from command line arguments
    j = 200
    r = 0.75
    m = 3

    print(f"Processing with j={j}, r={r}, m={m}")
```

```
    combstr = f"{int(j*r)}|{j}"
    heatmap_folder = img_dir

    img_filename = f"{int(j*r)}-{j}_{m}"
    filename = f"{combstr}_{m}"
    data_file = f"{data_dir}/{filename}.csv"
    data_we_have = pd.read_csv(data_file)
    n_biomarkers = len(data_we_have.biomarker.unique())

    if not os.path.exists(data_file):
        print(f"Data file not found: {data_file}")
        sys.exit(1)  # Exit early if the file doesn't exist
    else:
        print(f"Data file found: {data_file}")

    # Define the temporary result file
    temp_result_file = os.path.join(temp_results_dir, f"temp_results_{j}_{r}_{m}.json")

    dic = {}

    if combstr not in dic:
        dic[combstr] = []

    accepted_order_dicts = metropolis_hastings_soft_kmeans(
        data_we_have,
        iterations,
        n_shuffle,
    )

    utils.save_heatmap(
        accepted_order_dicts,
        burn_in,
        thining,
        folder_name=heatmap_folder,
        file_name=f"{img_filename}",
        title=f'heatmap of {filename}')

    most_likely_order_dic = utils.obtain_most_likely_order_dic(
        accepted_order_dicts, burn_in, thining)
    most_likely_order = list(most_likely_order_dic.values())
    tau, p_value = kendalltau(most_likely_order, range(1, n_biomarkers + 1))
```

```
    dic[combstr].append(tau)

    # Write the results to a unique temporary file inside the temp folder
    with open(temp_result_file, "w") as file:
        json.dump(dic, file, indent=4)
    print(f"{filename} is done! Results written to {temp_result_file}")
```

```
Current working directory: /Users/hongtaoh/Desktop/github/ebmBook
Data directory: /Users/hongtaoh/Desktop/github/ebmBook/data
Temp results directory: /Users/hongtaoh/Desktop/github/ebmBook/soft_kmeans/temp_json_results
Image directory: /Users/hongtaoh/Desktop/github/ebmBook/soft_kmeans/img
Processing with j=200, r=0.75, m=3
Data file found: /Users/hongtaoh/Desktop/github/ebmBook/data/150|200_3.csv

/var/folders/wx/xz5y_06d15q5pgl_mhv76c8r0000gn/T/ipykernel_12345/1007084789.py:261: RuntimeWa
  prob_of_accepting_new_order = np.exp(

iteration 10 done, current accepted likelihood: -4558.471198243365, current acceptance ratio
150|200_3 is done! Results written to /Users/hongtaoh/Desktop/github/ebmBook/soft_kmeans/temp
```

8.2 Result

We plot the resulting S probalistically using a heatmap.

```
dic
```

```
{'150|200': [-0.15555555555555553]}
```

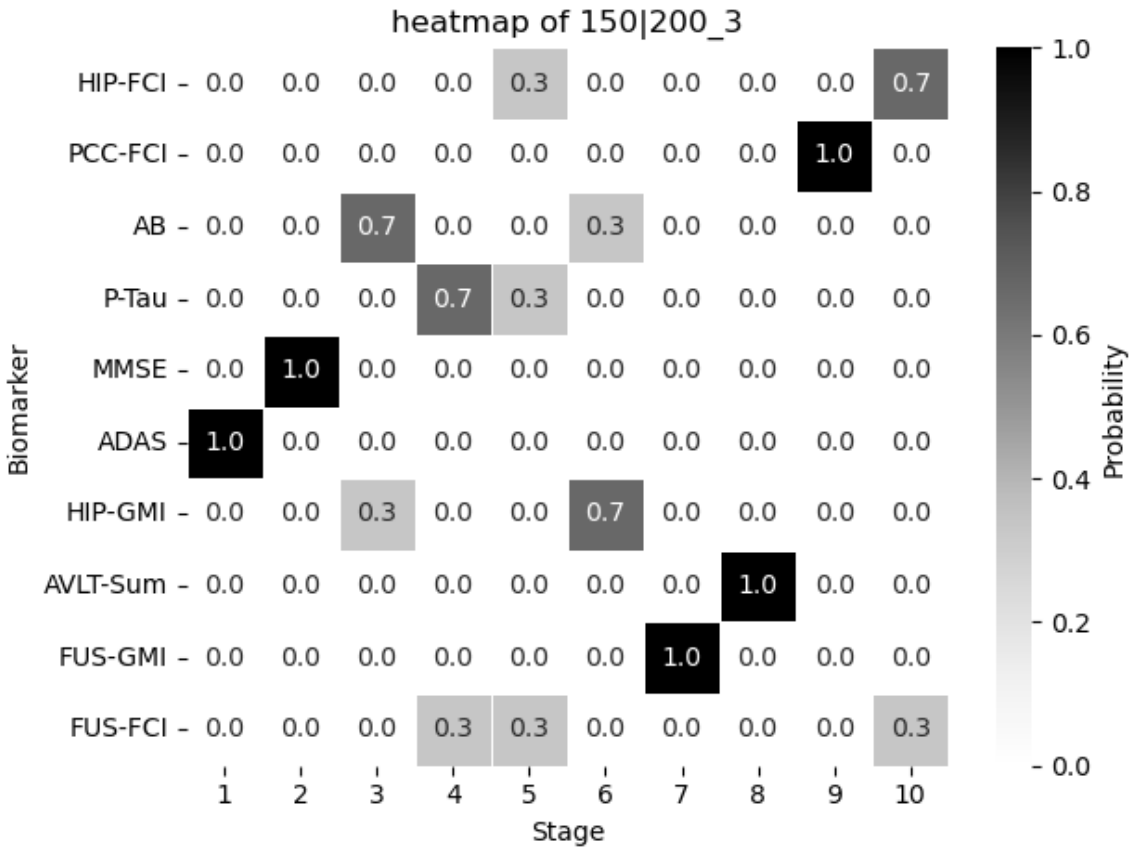


Figure 8.2: Result of Soft K-Means

9 Estimate S with Conjugate Priors

The basic idea of using conjugate Priors to estimate S is:

- We first estimate distribution parameters using hard K-Means, the exact procedure we covered in Section 4.1.
- We first randomly assign each diseased participant a stage. Then, in each iteration, we use the conjugate priors algorithm to update θ, ϕ (refer to Section 4.2) and also k_j. We use Metropolis–Hastings algorithm to accept or reject a proposed S.

9.1 Implementation

```
import numpy as np
import utils
import json
import pandas as pd
import utils
from scipy.stats import kendalltau
import sys
import os
import math
import random
```

```
def estimate_params_exact(m0, n0, s0_sq, v0, data):
    '''This is to estimate means and vars based on conjugate priors
    Inputs:
        - data: a vector of measurements
        - m0: prior estimate of $\mu$.
        - n0: how strongly is the prior belief in $m_0$ is held.
        - s0_sq: prior estimate of $\sigma^2$.
        - v0: prior degress of freedome, influencing the certainty of $s_0^2$.

    Outputs:
        - mu estiate, std estimate
```

Algorithm 3 Metropolis-Hastings Conjugate Priors

Input: data with participant id, biomarker, and its measurement; n_{shuffle}: number of elements to shuffle
Output: all_accepted_orders
all_accepted_orders $\leftarrow$ []
current_order $\leftarrow$ permutation(np.arange(1, n_{stages}))
current_ll $\leftarrow -\infty$
participant_stages $\leftarrow$ randomized_participant_stages()
for i in range(iterations) **do**
    new_order $\leftarrow$ current_order.copy()
    shuffle_order(new_order, n_{shuffle})
    data $\leftarrow$ update_data(data, new_order)
    $(\theta_0, \phi_0) \leftarrow$ get_prior_theta_phi(data)
    $(\theta, \phi) \leftarrow$ conjugate_prior_theta_phi(data, θ_0, ϕ_0)
    (all_ln_ll, participant_stages) $\leftarrow f$(data, participant_stages, θ, ϕ)
    prob_of_accepting_new_order $\leftarrow$ exp(all_ln_ll $-$ current_ll)
    if np.random.rand() $<$ prob_of_accepting_new_order **then**
        current_order $\leftarrow$ new_order
        current_ll $\leftarrow$ all_ln_ll
        all_accepted_orders.append(current_order)
    end if
end for
return all_accepted_orders

Figure 9.1: Conjugate Priors Algorithm

```
    '''
    # Data summary
    sample_mean = np.mean(data)
    sample_size = len(data)
    sample_var = np.var(data, ddof=1)  # ddof=1 for unbiased estimator

    # Update hyperparameters for the Normal-Inverse Gamma posterior
    updated_m0 = (n0 * m0 + sample_size * sample_mean) / (n0 + sample_size)
    updated_n0 = n0 + sample_size
    updated_v0 = v0 + sample_size
    updated_s0_sq = (1 / updated_v0) * ((sample_size - 1) * sample_var + v0 * s0_sq +
                                        (n0 * sample_size / updated_n0) * (sample_mean - m0)**
    updated_alpha = updated_v0/2
    updated_beta = updated_v0*updated_s0_sq/2

    # Posterior estimates
    mu_posterior_mean = updated_m0
    sigma_squared_posterior_mean = updated_beta/updated_alpha

    mu_estimation = mu_posterior_mean
    std_estimation = np.sqrt(sigma_squared_posterior_mean)

    return mu_estimation, std_estimation

def get_theta_phi_conjugate_priors(biomarkers, data_we_have, theta_phi_kmeans):
    '''To get estimated parameters, returns a hashmap
    Input:
    - biomarkers: biomarkers
    - data_we_have: participants data filled with initial or updated participant_stages
    - theta_phi_kmeans: a hashmap of dicts, which are the prior theta and phi values
        obtained from the initial constrained kmeans algorithm

    Output:
    - a hashmap of dictionaries. Key is biomarker name and value is a dictionary.
    Each dictionary contains the theta and phi mean/std values for a specific biomarker.
    '''
    # empty list of dictionaries to store the estimates
    hashmap_of_means_stds_estimate_dicts = {}

    for biomarker in biomarkers:
        # Initialize dictionary outside the inner loop
        dic = {'biomarker': biomarker}
```

```
        for affected in [True, False]:
            data_full = data_we_have[(data_we_have.biomarker == biomarker) & (
                data_we_have.affected == affected)]
            if len(data_full) > 1:
                measurements = data_full.measurement
                s0_sq = np.var(measurements, ddof=1)
                m0 = np.mean(measurements)
                mu_estimate, std_estimate = estimate_params_exact(
                    m0=m0, n0=1, s0_sq=s0_sq, v0=1, data=measurements)
                if affected:
                    dic['theta_mean'] = mu_estimate
                    dic['theta_std'] = std_estimate
                else:
                    dic['phi_mean'] = mu_estimate
                    dic['phi_std'] = std_estimate
            # If there is only one observation or not observation at all, resort to theta_phi
            # YES, IT IS POSSIBLE THAT DATA_FULL HERE IS NULL
            # For example, if a biomarker indicates stage of (num_biomarkers), but all partic
            # are smaller than that stage; so that for all participants, this biomarker is no
            else:
                # print(theta_phi_kmeans)
                if affected:
                    dic['theta_mean'] = theta_phi_kmeans[biomarker]['theta_mean']
                    dic['theta_std'] = theta_phi_kmeans[biomarker]['theta_std']
                else:
                    dic['phi_mean'] = theta_phi_kmeans[biomarker]['phi_mean']
                    dic['phi_std'] = theta_phi_kmeans[biomarker]['phi_std']
        # print(f"biomarker {biomarker} done!")
        hashmap_of_means_stds_estimate_dicts[biomarker] = dic
    return hashmap_of_means_stds_estimate_dicts

def compute_all_participant_ln_likelihood_and_update_participant_stages(
        n_participants,
        data,
        non_diseased_participant_ids,
        estimated_theta_phi,
        disease_stages,
        participant_stages,
):
    all_participant_ln_likelihood = 0
    for p in range(n_participants):
        # this participant data
```

```
        pdata = data[data.participant == p].reset_index(drop=True)

        """If this participant is not diseased (i.e., if we know k_j is equal to 0)
        We still need to compute the likelihood of this participant seeing this sequence of b
        but we do not need to estimate k_j like below

        We still need to compute the likelihood because we need to add it to all_participant_
        """
        if p in non_diseased_participant_ids:
            this_participant_likelihood = utils.compute_likelihood(
                pdata, k_j=0, theta_phi=estimated_theta_phi)
            this_participant_ln_likelihood = np.log(
                this_participant_likelihood + 1e-10)
        else:
            # initiaze stage_likelihood
            stage_likelihood_dict = {}
            for k_j in disease_stages:
                # even though data above has everything, it is filled up by random stages
                # we don't like it and want to know the true k_j. All the following is to upd
                participant_likelihood = utils.compute_likelihood(
                    pdata, k_j, estimated_theta_phi)
                # update each stage likelihood for this participant
                stage_likelihood_dict[k_j] = participant_likelihood
            likelihood_sum = sum(stage_likelihood_dict.values())
            normalized_stage_likelihood = [
                l/likelihood_sum for l in stage_likelihood_dict.values()]
            sampled_stage = np.random.choice(
                disease_stages, p=normalized_stage_likelihood)
            participant_stages[p] = sampled_stage

            # use weighted average likelihood because we didn't know the exact participant st
            # all above to calculate participant_stage is only for the purpous of calculate t
            this_participant_likelihood = np.mean(likelihood_sum)
            this_participant_ln_likelihood = np.log(
                this_participant_likelihood + 1e-10)
        """
        All the codes in between are calculating this_participant_ln_likelihood.
        If we already know kj=0, then
        it's very simple. If kj is unknown, we need to calculate the likelihood of seeing
        this sequence of biomarker
        data at different stages, and get the relative likelihood before
        we get a sampled stage (this is for estimating theta and phi).
```

```
        Then we calculate this_participant_ln_likelihood using average likelihood.
        """
        all_participant_ln_likelihood += this_participant_ln_likelihood
    return all_participant_ln_likelihood

def update_data_by_the_new_participant_stages(data, participant_stages, n_participants):
    '''This is to fill up data_we_have.
    Basically, add two columns: k_j, affected, and modify diseased column
    based on the initial or updated participant_stages
    Note that we assume here we've already got S_n

    Inputs:
        - data_we_have
        - participant_stages: np array
        - participants: 0-99
    '''
    participant_stage_dic = dict(
        zip(np.arange(0, n_participants), participant_stages))
    data['k_j'] = data.apply(
        lambda row: participant_stage_dic[row.participant], axis=1)
    data['diseased'] = data.apply(lambda row: row.k_j > 0, axis=1)
    data['affected'] = data.apply(lambda row: row.k_j >= row.S_n, axis=1)
    return data

"""The version without reverting back to the max order
"""
def metropolis_hastings_with_conjugate_priors(
    data_we_have,
    iterations,
    n_shuffle
):
    n_participants = len(data_we_have.participant.unique())
    biomarkers = data_we_have.biomarker.unique()
    n_biomarkers = len(biomarkers)
    n_stages = n_biomarkers + 1
    diseased_stages = np.arange(start=1, stop=n_stages, step=1)

    non_diseased_participant_ids = data_we_have.loc[
        data_we_have.diseased == False].participant.unique()

    # initialize empty lists
    acceptance_count = 0
```

```
    all_current_accepted_order_dicts = []

    # initialize an ordering and likelihood
    # note that it should be a random permutation of numbers 1-10
    current_accepted_order = np.random.permutation(np.arange(1, n_stages))
    current_accepted_order_dict = dict(zip(biomarkers, current_accepted_order))
    current_accepted_likelihood = -np.inf

    participant_stages = np.zeros(n_participants)
    for idx in range(n_participants):
        if idx not in non_diseased_participant_ids:
            # 1-len(diseased_stages), inclusive on both ends
            participant_stages[idx] = random.randint(1, len(diseased_stages))

    for _ in range(iterations):
        new_order = current_accepted_order.copy()
        utils.shuffle_order(new_order, n_shuffle)
        current_order_dict = dict(zip(biomarkers, new_order))

        # copy the data to avoid modifying the original
        data = data_we_have.copy()
        data['S_n'] = data.apply(
            lambda row: current_order_dict[row['biomarker']], axis=1)
        # add kj and affected for the whole dataset based on participant_stages
        # also modify diseased col (because it will be useful for the new theta_phi_kmeans)
        data = update_data_by_the_new_participant_stages(
            data, participant_stages, n_participants)
        # should be inside the for loop because once the participant_stages change,
        # the diseased column changes as well.
        theta_phi_kmeans = utils.get_theta_phi_estimates(
            data_we_have,
        )
        estimated_theta_phi = get_theta_phi_conjugate_priors(
            biomarkers, data, theta_phi_kmeans)

        all_participant_ln_likelihood = compute_all_participant_ln_likelihood_and_update_part
            n_participants,
            data,
            non_diseased_participant_ids,
            estimated_theta_phi,
            diseased_stages,
            participant_stages,
```

```
        )

        # ratio = likelihood/best_likelihood
        # because we are using np.log(likelihood) and np.log(best_likelihood)
        # np.exp(a)/np.exp(b) = np.exp(a - b)
        # if a > b, then np.exp(a - b) > 1

        # Log-Sum-Exp Trick
        max_likelihood = max(all_participant_ln_likelihood,
                             current_accepted_likelihood)
        prob_of_accepting_new_order = np.exp(
            (all_participant_ln_likelihood - max_likelihood) -
            (current_accepted_likelihood - max_likelihood)
        )

        # it will definitly update at the first iteration
        if np.random.rand() < prob_of_accepting_new_order:
            acceptance_count += 1
            current_accepted_order = new_order
            current_accepted_likelihood = all_participant_ln_likelihood
            current_accepted_order_dict = current_order_dict

        acceptance_ratio = acceptance_count*100/(_+1)
        all_current_accepted_order_dicts.append(current_accepted_order_dict)

        # if _ >= burn_in and _ % thining == 0:
        if (_+1) % 10 == 0:
            formatted_string = (
                f"iteration {_ + 1} done, "
                f"current accepted likelihood: {current_accepted_likelihood}, "
                f"current acceptance ratio is {acceptance_ratio:.2f} %, "
                f"current accepted order is {current_accepted_order_dict.values()}, "
            )

    return all_current_accepted_order_dicts
```

```
n_shuffle = 2
iterations = 10
burn_in = 2
thining = 2

base_dir = os.getcwd()
```

```
print(f"Current working directory: {base_dir}")
data_dir = os.path.join(base_dir, "data")
conjugate_priors_dir = os.path.join(base_dir, 'conjugate_priors')
temp_results_dir = os.path.join(conjugate_priors_dir, "temp_json_results")
img_dir = os.path.join(conjugate_priors_dir, 'img')
results_file = os.path.join(conjugate_priors_dir, "results.json")

os.makedirs(conjugate_priors_dir, exist_ok=True)
os.makedirs(temp_results_dir, exist_ok=True)
os.makedirs(img_dir, exist_ok=True)

print(f"Data directory: {data_dir}")
print(f"Temp results directory: {temp_results_dir}")
print(f"Image directory: {img_dir}")

if __name__ == "__main__":

    # Read parameters from command line arguments
    j = 200
    r = 0.75
    m = 3

    print(f"Processing with j={j}, r={r}, m={m}")

    combstr = f"{int(j*r)}|{j}"
    heatmap_folder = img_dir

    img_filename = f"{int(j*r)}-{j}_{m}"
    filename = f"{combstr}_{m}"
    data_file = f"{data_dir}/{filename}.csv"
    data_we_have = pd.read_csv(data_file)
    n_biomarkers = len(data_we_have.biomarker.unique())

    if not os.path.exists(data_file):
        print(f"Data file not found: {data_file}")
        sys.exit(1)  # Exit early if the file doesn't exist
    else:
        print(f"Data file found: {data_file}")

    # Define the temporary result file
    temp_result_file = os.path.join(temp_results_dir, f"temp_results_{j}_{r}_{m}.json")
```

```
    # temp_result_file = f"{temp_results_dir}/temp_results_{j}_{r}_{m}.json"

    dic = {}

    if combstr not in dic:
        dic[combstr] = []

    accepted_order_dicts = metropolis_hastings_with_conjugate_priors(
        data_we_have,
        iterations,
        n_shuffle,
    )

    utils.save_heatmap(
        accepted_order_dicts,
        burn_in,
        thining,
        folder_name=heatmap_folder,
        file_name=f"{img_filename}",
        title=f'heatmap of {filename}')

    most_likely_order_dic = utils.obtain_most_likely_order_dic(
        accepted_order_dicts, burn_in, thining)
    most_likely_order = list(most_likely_order_dic.values())
    tau, p_value = kendalltau(most_likely_order, range(1, n_biomarkers + 1))

    dic[combstr].append(tau)

    # Write the results to a unique temporary file inside the temp folder
    with open(temp_result_file, "w") as file:
        json.dump(dic, file, indent=4)
    print(f"{filename} is done! Results written to {temp_result_file}")
```

```
Current working directory: /Users/hongtaoh/Desktop/github/ebmBook
Data directory: /Users/hongtaoh/Desktop/github/ebmBook/data
Temp results directory: /Users/hongtaoh/Desktop/github/ebmBook/conjugate_priors/temp_json_res
Image directory: /Users/hongtaoh/Desktop/github/ebmBook/conjugate_priors/img
Processing with j=200, r=0.75, m=3
Data file found: /Users/hongtaoh/Desktop/github/ebmBook/data/150|200_3.csv
150|200_3 is done! Results written to /Users/hongtaoh/Desktop/github/ebmBook/conjugate_priors
```

9.2 Result

We plot the resulting S probalistically using a heatmap.

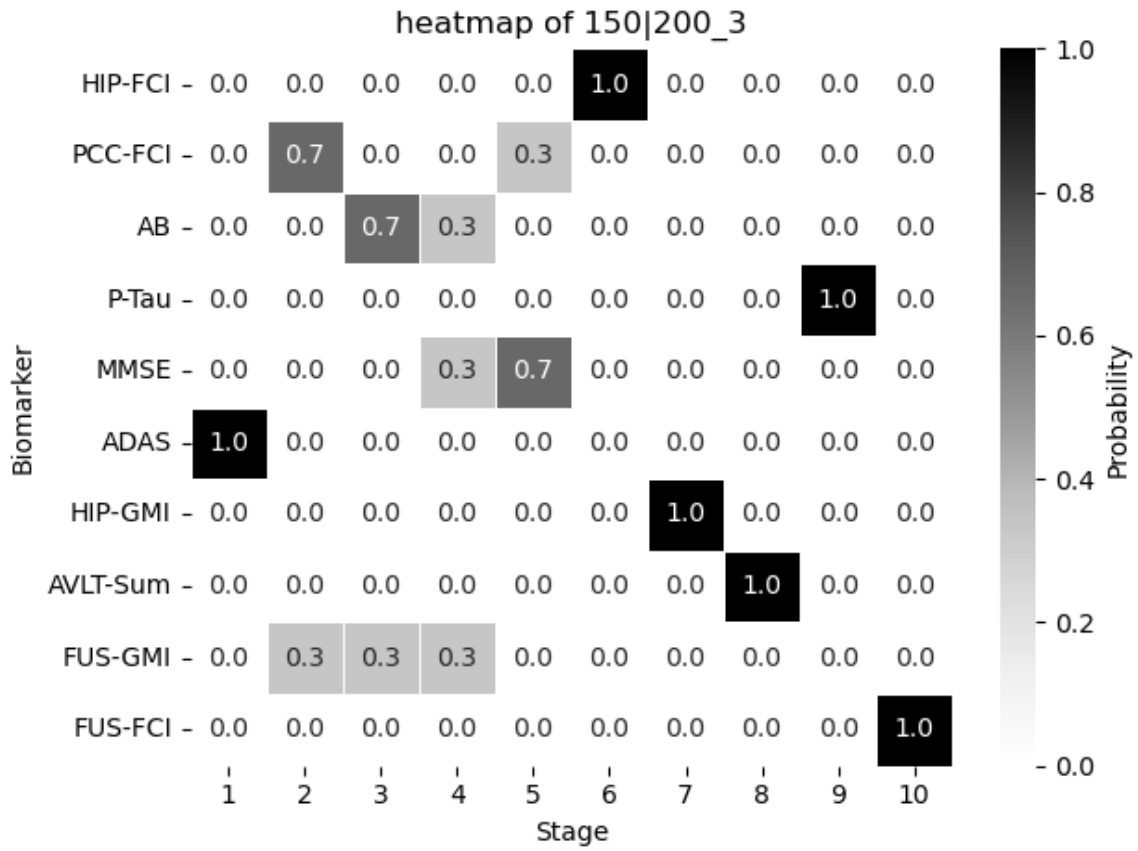


Figure 9.2: Result of Conjugate Priors

```
dic
```

```
{'150|200': [0.28888888888888886]}
```

10 Final Results

We only used 10 iterations in the previous three chapters to demonstrate how our algorithm works. However, to get them to really work, we need to run several thousand iterations.

Also, to cancel out noises and randomness, we need to use all the fifty variations of data.

With the help of The Center for High Throughput Computing at the University of Wisconsin-Madison, we were able to run these tests. In the following, we present our results.

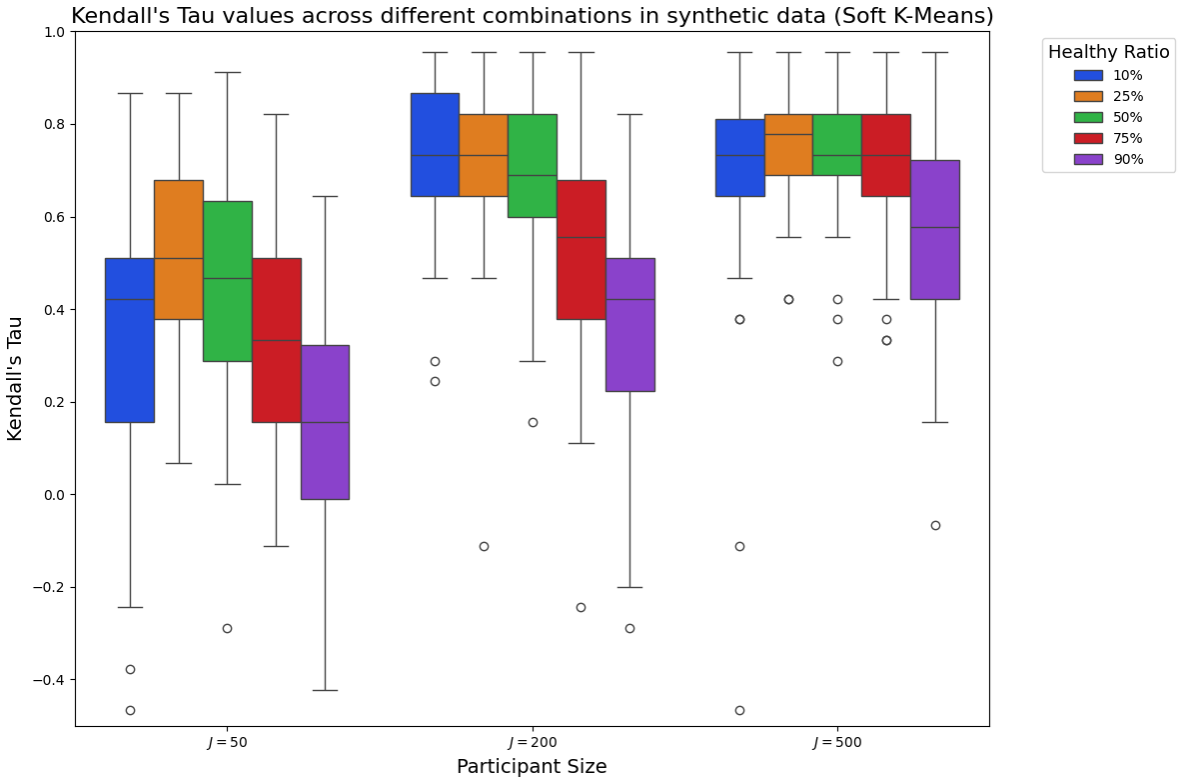
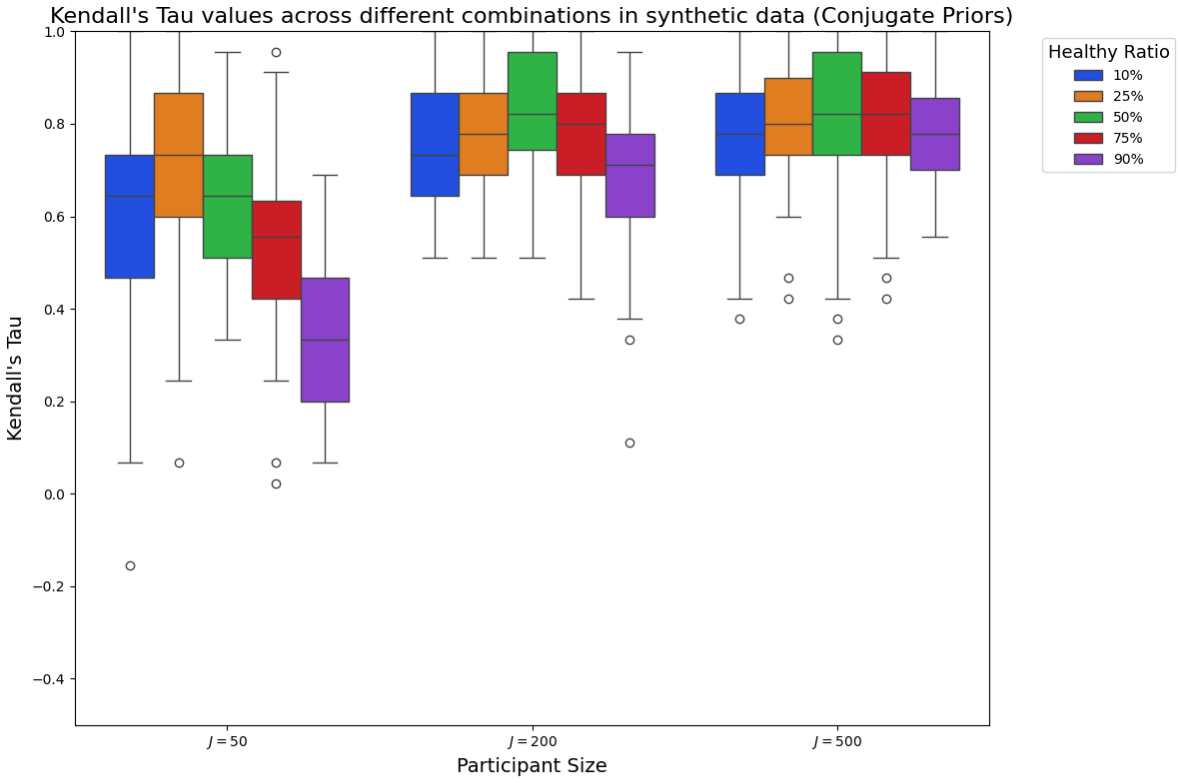


Figure 10.1: Results of Soft K-Means



Notes:

This figure shows Kendall's Tau for different combinations of participant size and healthy ratio.
Each bombination has 50 variants of datasets
The results are derived from our own implementation of soft kmeans based on synthetic data with 10 biomarkers.
Number of iterations is 10000.

Figure 10.2: Results of Conjugate Priors

From the two results, we are able to see that it is better to have more participants, because that offers more information for our models. In terms of healthy ratio, it seems 50% is a sweet spot.

Also, we notice that conjugate priors perform better than soft K-Means.

We also tested the two algorithms developed by UCL POND group with the same 750 datasets: EBM Basic and KDE EBM. See the following for results:

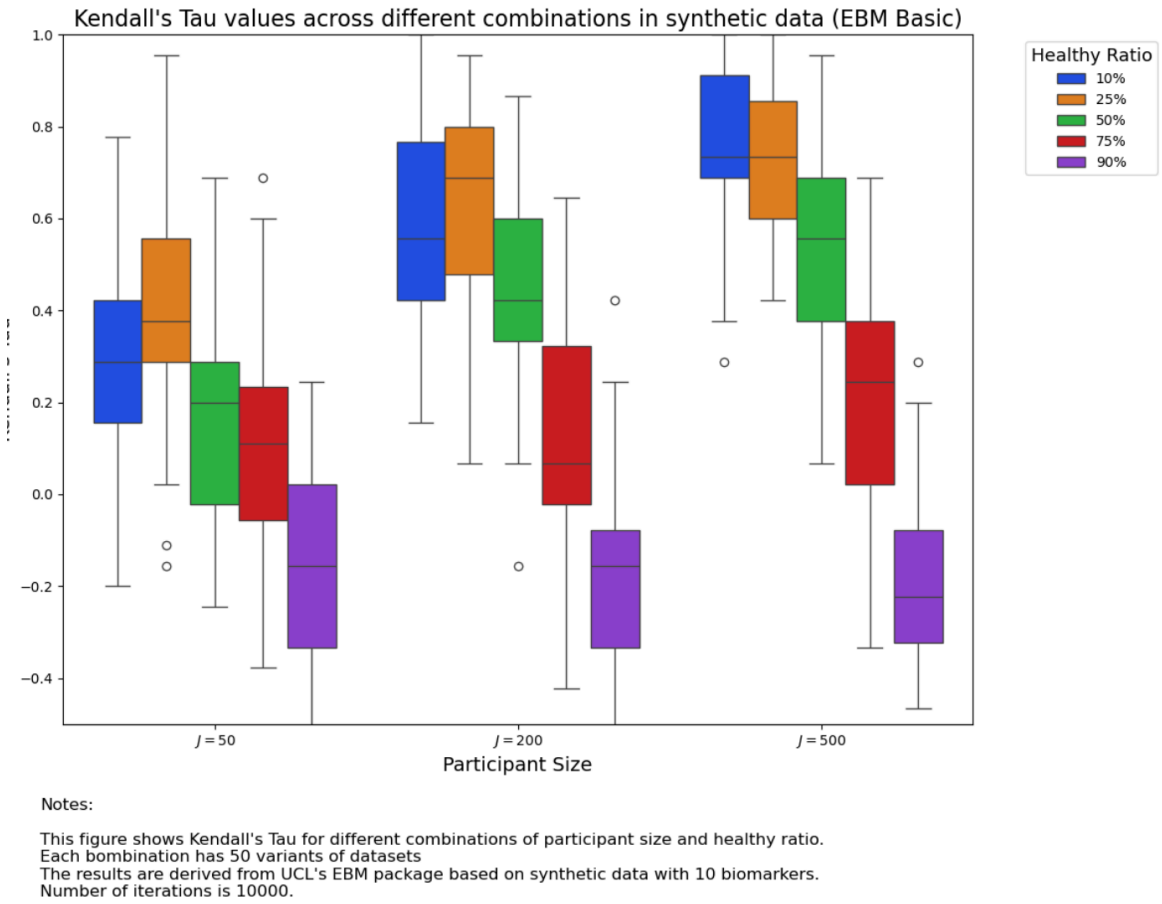


Figure 10.3: Results of EBM

The parameters we used in the package of ebm are:

- `n_iter = 10000`
- `greedy_n_iter=10`
- `greedy_n_init=5`

More specific configurations can be found at https://github.com/hongtaoh/ucl_ebm/blob/master/implement/ca

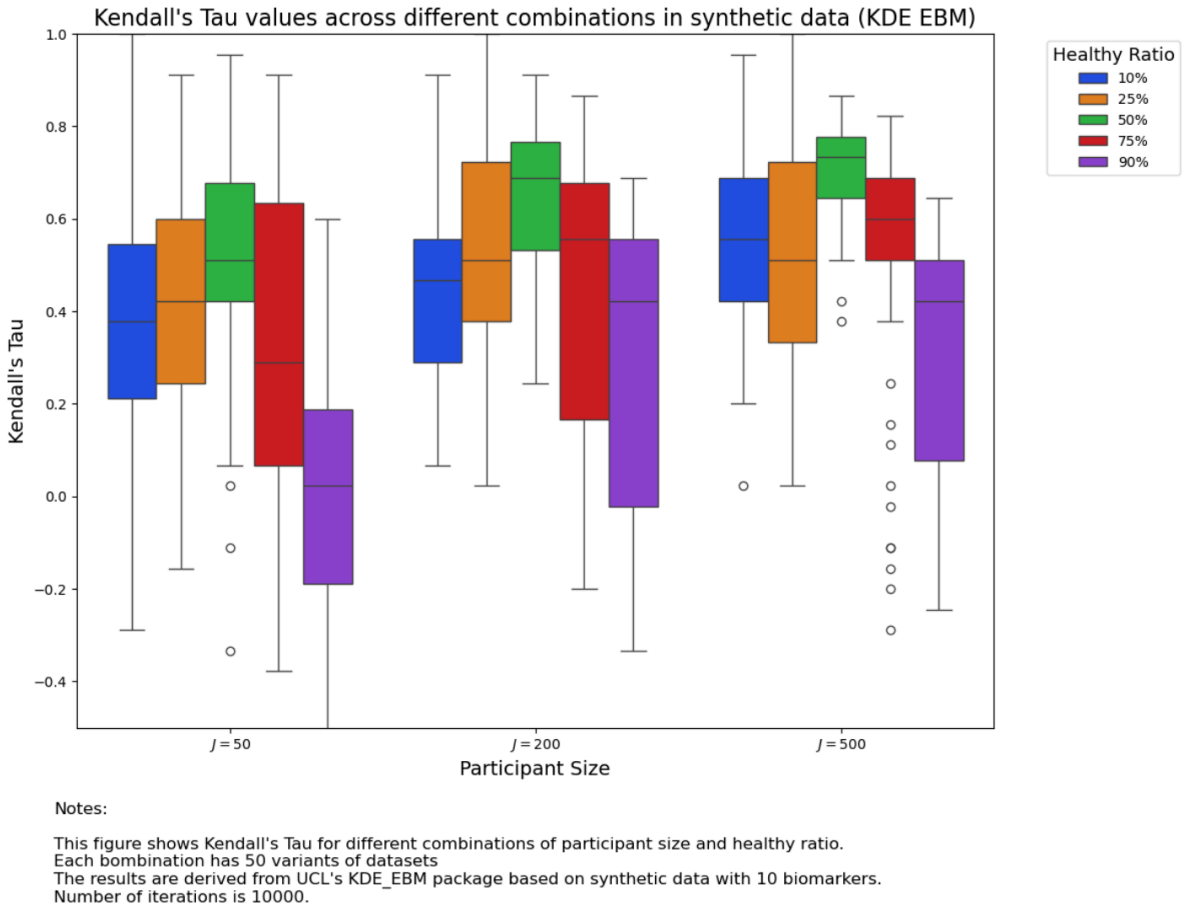


Figure 10.4: Results of KDE EBM

The parameters we used in the package of kde-ebm are:

- `n_iter = 10000`
- `greedy_n_iter=10`
- `greedy_n_init=5`

More specific configurations can be found at https://github.com/hongtaoh/ucl_kde_ebm/blob/master/impleme

We can see that the performance of KDE EBM is more stable but EBM basic performs well when the healthy ratio is below 50%.

Neither of these methods had results as good as conjugate priors. We have to point out that, even though their accuracy is not as high, their speed is really high. Both of these two algorithms can generate results with a single laptop GPU in just one hour; however, it might take days for our algorithms.

References

Babaki, Behrouz. 2017. "COP-Kmeans Version 1.5." https://doi.org/10.5281/zenodo.831850.

Chen, Guangyu, Hao Shu, Gang Chen, B Douglas Ward, Piero G Antuono, Zhijun Zhang, Shi-Jiang Li, Alzheimer's Disease Neuroimaging Initiative, et al. 2016. "Staging Alzheimer's Disease Risk by Sequencing Brain Function and Structure, Cerebrospinal Fluid, and Cognition Biomarkers." *Journal of Alzheimer's Disease* 54 (3): 983–93.

Clyde, M, M Cetinkaya-Rundel, C Rundel, D Banks, C Chai, and L Huang. 2022. "An Introduction to Bayesian Thinking: A Companion to the Statistics with r Course. 2020." https://statswithr.github.io/book/.

Fonteijn, Hubert M, Marc Modat, Matthew J Clarkson, Josephine Barnes, Manja Lehmann, Nicola Z Hobbs, Rachael I Scahill, et al. 2012. "An Event-Based Model for Disease Progression and Its Application in Familial Alzheimer's Disease and Huntington's Disease." *NeuroImage* 60 (3): 1880–89.