Leveraging G-Methods to Mitigate Time-Varying Confounding in Longitudinal Data Analysis

Hyemin Park, Grace Yin, Yulin Zhang, Qinhan Zhou

Appendix A

A1 Code for simulation data generation

```
import numpy as np
import pandas as pd
import sys
from sklearn.linear_model import LinearRegression

seed = int(sys.argv[1])

np.random.seed(seed)

# number of observations Y
n_obs = 2000

# u and a0 are Bernoulli with p = 0.4 and 0.5, respectively
u = (np.random.uniform(size=n_obs) < 0.4).astype(float)
a0 = (np.random.uniform(size=n_obs) < 0.5).astype(float)

# Condition on a0 and u to generate l1, Bernoulli with p = 0.25 + 0.3*a0 - 0.2*u -
0.05*a0*u
l1_prob = 0.25 + 0.3*a0 - 0.2*u - 0.05*a0*u
l1 = (np.random.uniform(size=n_obs) < l1_prob).astype(float)

# Condition on a0 and l1 to generate a1, Bernoulli with p = 0.4 + 0.5*a0 - 0.3*l -
0.4*a0*l
a1_prob = 0.4 + 0.5*a0 - 0.3*l1 - 0.4*a0*l1
a1 = (np.random.uniform(size=n_obs) < a1_prob).astype(float)

# Condition on a0, a1, and u to generate y, Normal with mean 2.5 - 0.5*a0 - 0.75*a1 -
u and variance 0.2
y = (2.5 - 0.5*a0 - 0.75*a1 - u + 0.2*a0*a1 +
    0.2*np.random.normal(size=n_obs))
```

```
# Create a DataFrame with all variables
df = pd.DataFrame({
   'u': u,
   'a0': a0,
   'l': l1,
   'a1': a1,
   'y': y
})

# Save to file
df.to_csv('sim1.csv', index=False)
```

A2 Code for applying linear regression on the simulation dataset

```
def simple_models(df):
   # Create interaction term for final regression
   df['a0xa1'] = df['a0'] * df['a1']

   # m1 weighted regression
   X_m1 = df[['a0', 'a1', 'a0xa1']]
   m1_model = LinearRegression()
   # For weighted regression in sklearn, we need to use sample_weight in fit
   m1_model.fit(X_m1, df['y'])

   # m2 weighted regression
   X_m2 = df[['a0', 'a1', 'a0xa1', 'l']]
   m2_model = LinearRegression()
   # For weighted regression in sklearn, we need to use sample_weight in fit
   m2_model.fit(X_m2, df['y'])

   # Get results
   coefficients = pd.DataFrame({
       'Feature': ['gamma_int', 'gamma_0', 'gamma_1', 'gamma_01', 'gamma_int',
'gamma_0', 'gamma_1', 'gamma_01'],
       'Model': ['m1', 'm1', 'm1', 'm1', 'm2', 'm2', 'm2', 'm2'],
       'Coefficient': [m1_model.intercept_] + m1_model.coef_.tolist() +
[m2_model.intercept_] + m2_model.coef_.tolist()[:-1]
   })
   return coefficients

def bootstrap_simple_models(df, n_bootstrap = 10000):
   results_m1 = {
       'gamma_int': [],
       'gamma_0': [],
       'gamma_1': [],
       'gamma_01': []
   }
   results_m2 = {
       'gamma_int': [],
       'gamma_0': [],
       'gamma_1': [],
       'gamma_01': []
   }
   for i in range(n_bootstrap):
       # Sample with replacement
       bootstrap_indices = np.random.choice(n_obs, size=n_obs, replace=True)
       bootstrap_sample = df.iloc[bootstrap_indices].copy()

       # Calculate statistics
       bootstrap_results = simple_models(bootstrap_sample)

       # Store results
       for key in results_m1.keys():
           results_m1[key].append(bootstrap_results.loc[(bootstrap_results['Model'] ==
'm1') &
                                                        (bootstrap_results['Feature']
== key), 'Coefficient'].values[0])
           results_m2[key].append(bootstrap_results.loc[(bootstrap_results['Model'] ==
'm2') &
                                                        (bootstrap_results['Feature']
== key), 'Coefficient'].values[0])
   return results_m1, results_m2



results_m1, results_m2 = bootstrap_simple_models(df, n_bootstrap = 1000)
# Print results
print("m1")
outdict = {"Parameter": [], "Model":[], "Mean": [], "Std. Error": [], "95% CI Lower":
[], "95% CI Upper": []}
for param, stats in results_m1.items():
   outdict["Parameter"].append(param)
   outdict["Model"].append("m1")
```

```
    outdict["Mean"].append(np.mean(stats))
    outdict["Std. Error"].append(np.std(stats, ddof=1))
    outdict["95% CI Lower"].append(np.percentile(stats, 2.5))
    outdict["95% CI Upper"].append(np.percentile(stats, 97.5))
for param, stats in results_m2.items():
    outdict["Parameter"].append(param)
    outdict["Model"].append("m2")
    outdict["Mean"].append(np.mean(stats))
    outdict["Std. Error"].append(np.std(stats, ddof=1))
    outdict["95% CI Lower"].append(np.percentile(stats, 2.5))
    outdict["95% CI Upper"].append(np.percentile(stats, 97.5))
print(pd.DataFrame(outdict))
```

A3 Code for applying G-formula on the simulation dataset

```
def true_exps():
    # Calculate true terms following the formula
    # E(y_a0a1) = 2.1 - 0.5*a0 - 0.75*a1 + 0.2*a0*a1

    true_y00 = 2.1
    true_y10 = 2.1 - 0.5
    true_y01 = 2.1 - 0.75
    true_y11 = 2.1 - 0.5 - 0.75 + 0.2

    gamma_int = true_y00
    gamma_0 = true_y10 - true_y00
    gamma_1 = true_y01 - true_y00
    gamma_01 = true_y11 - true_y10 - true_y01 + true_y00

    return {
        'true_y00': true_y00,
        'true_y10': true_y10,
        'true_y01': true_y01,
        'true_y11': true_y11,
        'gamma_int': gamma_int,
        'gamma_0': gamma_0,
        'gamma_1': gamma_1,
        'gamma_01': gamma_01
    }

def gcomp_exps(df):
    # From Daniel et al. (2012), the derived estimator for E(y_a0a1) is:
```

```
    # E(y_a0a1) = sum_{l1} E(Y | A0 = a0, A1 = a1, L = l1) * P(L = l1 | A0 = a0)
    # One is interested in if model E(y_a0a1) = gamma_int + gamma_0*a0 + gamma_1*a1 +
gamma_01*a0*a1 holds
    # where gamma_int = E(y_00), gamma_0 = E(y_10) - E(y_00), gamma_1 = E(y_01) -
E(y_00), gamma_01 = E(y_11) - E(y_10) - E(y_01) + E(y_00)
    # The true values are calculated in the true_exps function

    def get_exp(df, a0, a1):
        # Get expected value of y given a0, a1, and l1
        p_l1_a0 = len(df[(df['a0'] == a0) & (df['l'] == 1)]) / len(df[df['a0'] == a0])
        p_l0_a0 = 1 - p_l1_a0
        e_a0_a1_l1 = df[(df['a0'] == a0) & (df['a1'] == a1) & (df['l'] ==
1)]['y'].mean()
        e_a0_a1_l0 = df[(df['a0'] == a0) & (df['a1'] == a1) & (df['l'] ==
0)]['y'].mean()
        return p_l1_a0 * e_a0_a1_l1 + p_l0_a0 * e_a0_a1_l0

    Gcomp_y00 = get_exp(df, 0, 0)
    Gcomp_y10 = get_exp(df, 1, 0)
    Gcomp_y01 = get_exp(df, 0, 1)
    Gcomp_y11 = get_exp(df, 1, 1)

    # Calculate gamma values
    gamma_int = Gcomp_y00
    gamma_0 = Gcomp_y10 - Gcomp_y00
    gamma_1 = Gcomp_y01 - Gcomp_y00
    gamma_01 = Gcomp_y11 - Gcomp_y10 - Gcomp_y01 + Gcomp_y00

    return {
        'gamma_int': gamma_int,
        'gamma_0': gamma_0,
        'gamma_1': gamma_1,
        'gamma_01': gamma_01
    }

def bootstrap_gcomp(df, n_bootstrap=1000):
    """
    Perform bootstrap analysis of gcomp results
    """
    n_samples = len(df)
    results = {
        'gamma_int': [],
        'gamma_0': [],
        'gamma_1': [],
        'gamma_01': []
    }

    # Perform bootstrap
    for _ in range(n_bootstrap):
        # Sample with replacement
        bootstrap_indices = np.random.choice(n_samples, size=n_samples, replace=True)
        bootstrap_sample = df.iloc[bootstrap_indices].copy()

        # Calculate statistics
        bootstrap_results = gcomp_exps(bootstrap_sample)

        # Store results
        for key in results.keys():
            results[key].append(bootstrap_results[key])

    # Calculate summary statistics
    summary = {}
    for key in results.keys():
        values = np.array(results[key])
        summary[key] = {
            'mean': np.mean(values),
            'std_err': np.std(values, ddof=1),
            'ci_lower': np.percentile(values, 2.5),
            'ci_upper': np.percentile(values, 97.5)
        }

    return summary

# Load the data
df = pd.read_csv('sim1.csv')

# Run bootstrap analysis
truth = true_exps()
bootstrap_results = bootstrap_gcomp(df, n_bootstrap=1000)

# Print results as dataframe
print("\nBootstrap Results:")
outdict = {"Parameter": [], "Truth":[], "Mean": [], "Std. Error": [], "95% CI Lower":
[], "95% CI Upper": []}
```

```
for param, stats in bootstrap_results.items():
    outdict["Parameter"].append(param)
    outdict["Truth"].append(truth[param])
    outdict["Mean"].append(stats['mean'])
    outdict["Std. Error"].append(stats['std_err'])
    outdict["95% CI Lower"].append(stats['ci_lower'])
    outdict["95% CI Upper"].append(stats['ci_upper'])

print(pd.DataFrame(outdict))
```

A4 Code for applying IPW on the simulation dataset

```
from sklearn.linear_model import LinearRegression
import numpy as np
import pandas as pd

def true_exps():
    # Calculate true terms following the formula
    # E(y_a0a1) = 2.1 - 0.5*a0 - 0.75*a1 + 0.2*a0*a1

    true_y00 = 2.1
    true_y10 = 2.1 - 0.5
    true_y01 = 2.1 - 0.75
    true_y11 = 2.1 - 0.5 - 0.75 + 0.2

    gamma_int = true_y00
    gamma_0 = true_y10 - true_y00
    gamma_1 = true_y01 - true_y00
    gamma_01 = true_y11 - true_y10 - true_y01 + true_y00

    return {
        'true_y00': true_y00,
        'true_y10': true_y10,
        'true_y01': true_y01,
        'true_y11': true_y11,
        'gamma_int': gamma_int,
        'gamma_0': gamma_0,
        'gamma_1': gamma_1,
        'gamma_01': gamma_01
    }

def IPW_true_weight_table():
    # from simulation formula, we know that the true weights are:
    def get_pa0(a0):
        return 0.5
    def get_pa1(a0, l):
        return 0.4 + 0.5*a0 - 0.3*l - 0.4*a0*l
    outdict = {"A0":[], "L1":[], "A1":[], "W":[]}
    for a0 in [0, 1]:
        for l1 in [0, 1]:
            for a1 in [0, 1]:
                outdict["A0"].append(a0)
                outdict["L1"].append(l1)
                outdict["A1"].append(a1)
                if a1 == 0:
                    outdict["W"].append(1/(get_pa0(a0)*(1-get_pa1(a0, l1))))
                else:
                    outdict["W"].append(1/(get_pa0(a0)*get_pa1(a0, l1)))
    return pd.DataFrame(outdict)

def IPW_est(df):
    # f_a0 = P(A0 = a0)
    f_a0 = df['a0'].mean()
    df['p0'] = f_a0
    df.loc[df['a0'] == 0, 'p0'] = 1 - f_a0

    # Create interaction term
    df['a0xl'] = df['a0'] * df['l']
    # Second regression (a1)
    X_1 = df[['a0', 'l', 'a0xl']]
    model_1 = LinearRegression()
    model_1.fit(X_1, df['a1'])
    df['p1'] = model_1.predict(X_1)
    # Replace p1 where a1 is 0
    df.loc[df['a1'] == 0, 'p1'] = 1 - df.loc[df['a1'] == 0, 'p1']

    # Calculate weights
    df['p'] = df['p0'] * df['p1']
    df['w'] = 1 / df['p']

    # Create interaction term for final regression
    df['a0xa1'] = df['a0'] * df['a1']

    # Final weighted regression
    X_final = df[['a0', 'a1', 'a0xa1']]
    final_model = LinearRegression()
    # For weighted regression in sklearn, we need to use sample_weight in fit
    final_model.fit(X_final, df['y'], sample_weight=df['w'])
    # Get R-squared
    r2 = final_model.score(X_final, df['y'], sample_weight=df['w'])

    # Get results
    coefficients = pd.DataFrame({
        'Feature': ['a0', 'a1', 'a0xa1'],
        'Coefficient': final_model.coef_
    })
    weights = df[['a0', 'l', 'a1', 'w']].groupby(['a0', 'l',
'a1']).mean().reset_index()
    return {
        "gamma_int": final_model.intercept_,
        "gamma_0": coefficients.loc[coefficients['Feature'] == 'a0',
'Coefficient'].values[0],
        "gamma_1": coefficients.loc[coefficients['Feature'] == 'a1',
'Coefficient'].values[0],
        "gamma_01": coefficients.loc[coefficients['Feature'] == 'a0xa1',
'Coefficient'].values[0],
    }, weights

def bootstrap_IPW(df, n_bootstrap = 10000):
    n_samples = len(df)
    results = {
        'gamma_int': [],
        'gamma_0': [],
        'gamma_1': [],
        'gamma_01': []
    }

    # Perform bootstrap
    for _ in range(n_bootstrap):
        # Sample with replacement
        bootstrap_indices = np.random.choice(n_samples, size=n_samples, replace=True)
        bootstrap_sample = df.iloc[bootstrap_indices].copy()

        # Calculate statistics
        bootstrap_results, _ = IPW_est(bootstrap_sample)
        # Store results
        for key in results.keys():
            results[key].append(bootstrap_results[key])

    # Calculate summary statistics
    summary = {}
    for key in results.keys():
        values = np.array(results[key])
        summary[key] = {
            'mean': np.mean(values),
            'std_err': np.std(values, ddof=1),
            'ci_lower': np.percentile(values, 2.5),
            'ci_upper': np.percentile(values, 97.5)
        }

    return summary

df = pd.read_csv('sim1.csv')

# get weights terms
truth = IPW_true_weight_table()
results, weights = IPW_est(df)
weights.columns = ['A0', 'L1', 'A1', 'W_IPW']
weights = weights.merge(truth, on=['A0', 'L1', 'A1'])
print("\nWeight Table:")
print(weights)

# Run bootstrap analysis
truth = true_exps()
bootstrap_results = bootstrap_IPW(df, n_bootstrap=1000)
# Print results
print("\nBootstrap Results:")
outdict = {"Parameter": [], "Truth":[], "Mean": [], "Std. Error": [], "95% CI Lower":
[], "95% CI Upper": []}
for param, stats in bootstrap_results.items():
    outdict["Parameter"].append(param)
    outdict["Truth"].append(truth[param])
    outdict["Mean"].append(stats['mean'])
    outdict["Std. Error"].append(stats['std_err'])
    outdict["95% CI Lower"].append(stats['ci_lower'])
    outdict["95% CI Upper"].append(stats['ci_upper'])
print(pd.DataFrame(outdict))
```

Appendix B

B1 Code for MIMIC data processing

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
import seaborn as sns
# Select necessary columns and limit to the first 45,000 rows for each
dataset
folder_path = "/Users/qinhan/Downloads/mimic-iii-clinical-database-1.4"
admissions = pd.read_csv(
   f'{folder_path}/ADMISSIONS.csv',
   usecols=['SUBJECT_ID', 'HADM_ID', 'ADMITTIME', 'DISCHTIME',
'HOSPITAL_EXPIRE_FLAG'],
   nrows=45000
)

patients = pd.read_csv(
   f'{folder_path}/PATIENTS.csv',
   usecols=['SUBJECT_ID', 'GENDER', 'DOB'],
   nrows=45000
)

icustays = pd.read_csv(
   f'{folder_path}/ICUSTAYS.csv',
   usecols=['SUBJECT_ID', 'HADM_ID', 'ICUSTAY_ID', 'INTIME', 'OUTTIME'],
   nrows=45000
)

prescriptions = pd.read_csv(
   f'{folder_path}/PRESCRIPTIONS.csv',
   usecols=['HADM_ID', 'DRUG', 'STARTDATE', 'ENDDATE'],
   nrows=45000
)

chartevents = pd.read_csv(
   f'{folder_path}/CHARTEVENTS.csv',
    usecols=['SUBJECT_ID', 'HADM_ID', 'ITEMID', 'VALUENUM', 'CHARTTIME'],
    nrows=45000
)

labevents = pd.read_csv(
    f'{folder_path}/LABEVENTS.csv',
    usecols=['SUBJECT_ID', 'HADM_ID', 'ITEMID', 'VALUENUM', 'CHARTTIME'],
    nrows=45000
)

outputevents = pd.read_csv(
    f'{folder_path}/OUTPUTEVENTS.csv',
    usecols=['SUBJECT_ID', 'HADM_ID', 'ITEMID', 'VALUE', 'CHARTTIME'],
    nrows=45000
)

print("ADMISSIONS:", admissions.shape)
print("PATIENTS:", patients.shape)
print("ICUSTAYS:", icustays.shape)
print("PRESCRIPTIONS:", prescriptions.shape)
print("CHARTEVENTS:", chartevents.shape)
print("LABEVENTS:", labevents.shape)
print("OUTPUTEVENTS:", outputevents.shape)

# Keywords to search
search_keyword = "blood pressure"
d_items =
pd.read_csv('/Users/qinhan/Downloads/mimic-iii-clinical-database-1.4/D_ITE
MS.csv', usecols=['ITEMID', 'LABEL'])
filtered_items = d_items[d_items['LABEL'].str.contains(search_keyword,
case=False, na=False)]
print(filtered_items)
# Combine datasets
# Step 1: Convert datetimes
admissions['ADMITTIME'] = pd.to_datetime(admissions['ADMITTIME'])
admissions['DISCHTIME'] = pd.to_datetime(admissions['DISCHTIME'])
icustays['INTIME'] = pd.to_datetime(icustays['INTIME'])
icustays['OUTTIME'] = pd.to_datetime(icustays['OUTTIME'])
prescriptions['STARTDATE'] = pd.to_datetime(prescriptions['STARTDATE'])
prescriptions['ENDDATE'] = pd.to_datetime(prescriptions['ENDDATE'])
```

```
chartevents['CHARTTIME'] = pd.to_datetime(chartevents['CHARTTIME'])

# # Step 2: Filter ICU patients
# icu_admissions =
admissions[admissions['HADM_ID'].isin(icustays['HADM_ID'])]

# # Step 3: Define treatment (Vasopress)
# treatment_drug = 'Vasopress'  # Replace with the actual drug name
# treated_patients =
prescriptions[prescriptions['DRUG'].str.contains(treatment_drug, na=False,
case=False)]

# # Step 4: Define outcome (Mortality)
# icu_admissions = icu_admissions.merge(icustays[['HADM_ID', 'ICUSTAY_ID',
'INTIME', 'OUTTIME']], on='HADM_ID', how='inner')

# # Step 5: Define confounder (Blood Pressure)
# confounder_itemid = 220050
# confounders = chartevents[chartevents['ITEMID'] == confounder_itemid]

# # Step 6: Merge treatment and confounders with ICU stays
# # Merge treatment
# merged_data = icu_admissions.merge(treated_patients[['HADM_ID',
'STARTDATE', 'ENDDATE']], on='HADM_ID', how='left')
# # Merge confounder
# merged_data = merged_data.merge(confounders[['HADM_ID', 'CHARTTIME',
'VALUENUM']], on='HADM_ID', how='left')

# # Step 7: Restrict to ICU period
# merged_data = merged_data[
#     (merged_data['CHARTTIME'] >= merged_data['INTIME']) &
#     (merged_data['CHARTTIME'] <= merged_data['OUTTIME'])
# ]

# # Rename columns
# merged_data.rename(columns={
#     'VALUENUM': 'Systolic_Blood_Pressure',
#     'HOSPITAL_EXPIRE_FLAG': 'Mortality',
#     'STARTDATE': 'Treatment_Start',
#     'ENDDATE': 'Treatment_End'
# }, inplace=True)

# # Display the first few rows
# print(merged_data.head())

# # Save the processed data for later analysis
# merged_data.to_csv('processed_mimic_data.csv', index=False)
# Initialize an empty DataFrame to store results
filtered_chartevents = pd.DataFrame()

# Use chunks to process CHARTEVENTS
chunk_size = 50000
for chunk in
pd.read_csv('/Users/qinhan/Downloads/mimic-iii-clinical-database-1.4/CHART
EVENTS.csv', usecols=['SUBJECT_ID', 'HADM_ID', 'ITEMID', 'VALUENUM',
'CHARTTIME'], chunksize=chunk_size):
   # Filter for systolic blood pressure (220050)
   chunk_filtered = chunk[chunk['ITEMID'] == 220050]
   filtered_chartevents = pd.concat([filtered_chartevents,
chunk_filtered], ignore_index=True)

# Convert CHARTTIME to datetime
filtered_chartevents['CHARTTIME'] =
pd.to_datetime(filtered_chartevents['CHARTTIME'])

# Save the filtered data for later use
filtered_chartevents.to_csv('systolic_blood_pressure.csv', index=False)

# Display the first few rows
print(filtered_chartevents.head())
systolic_bp = pd.read_csv('systolic_blood_pressure.csv',
parse_dates=['CHARTTIME'])

# Ensure ICU stay times are in datetime format
icustays['INTIME'] = pd.to_datetime(icustays['INTIME'])
icustays['OUTTIME'] = pd.to_datetime(icustays['OUTTIME'])

# Step 1: Merge systolic blood pressure with ICU stays based on HADM_ID
bp_merged = systolic_bp.merge(icustays[['HADM_ID', 'ICUSTAY_ID', 'INTIME',
'OUTTIME']], on='HADM_ID', how='inner')
```

```python
# Step 2: Filter systolic blood pressure during ICU stays
bp_merged = bp_merged[(bp_merged['CHARTTIME'] >= bp_merged['INTIME']) &
(bp_merged['CHARTTIME'] <= bp_merged['OUTTIME'])]

# Step 3: Merge with treatment data (vasopress)
final_data = bp_merged.merge(prescriptions[['HADM_ID', 'STARTDATE',
'ENDDATE', 'DRUG']], on='HADM_ID', how='left')

# Step 4: Merge with outcome data
final_data = final_data.merge(admissions[['HADM_ID',
'HOSPITAL_EXPIRE_FLAG']], on='HADM_ID', how='left')

# Rename columns
final_data.rename(columns={
'VALUENUM': 'Systolic_BP',
   'HOSPITAL_EXPIRE_FLAG': 'Mortality',
   'STARTDATE': 'Treatment_Start',
   'ENDDATE': 'Treatment_End'
}, inplace=True)

# Step 5: Handle missing values
final_data.dropna(subset=['Systolic_BP'], inplace=True)

# Fill missing Treatment_Start and Treatment_End with placeholder values
# Placeholder: ICU admission or discharge time
final_data['Treatment_Start'] =
final_data['Treatment_Start'].fillna(final_data['INTIME'])
final_data['Treatment_End'] =
final_data['Treatment_End'].fillna(final_data['OUTTIME'])

# Fill missing DRUG with "No Treatment"
final_data['DRUG'] = final_data['DRUG'].fillna("No Treatment")

# Create a binary treatment indicator
final_data['Treatment'] = np.where(final_data['DRUG'] == "No Treatment",
0, 1)

treated = final_data[final_data['Treatment'] == 1]
untreated = final_data[final_data['Treatment'] == 0]
```

```
# Define biologically plausible range for systolic BP
final_data = final_data[
    (final_data['Systolic_BP'] >= 30) &
    (final_data['Systolic_BP'] <= 300)
]

# Convert CHARTTIME to hourly intervals
final_data['Hour'] = final_data['CHARTTIME'].dt.floor('H')

# final_data.to_csv('final_dataset.csv', index=False)

# Display the first few rows of the final dataset
print(final_data.head())
# Count the number of treatments started within the ICU period
treated_within_icu = final_data[
    (final_data['Treatment_Start'] >= final_data['INTIME']) &
    (final_data['Treatment_Start'] <= final_data['OUTTIME'])
]
print(len(treated_within_icu))

# Ensure CHARTTIME is in datetime format
final_data['CHARTTIME'] = pd.to_datetime(final_data['CHARTTIME'])

# Step 1: Aggregate systolic BP by hour for each patient
final_data['Hour'] = final_data['CHARTTIME'].dt.floor('H')  # Round to
hourly intervals
time_agg = final_data.groupby(['SUBJECT_ID', 'ICUSTAY_ID', 'Hour']).agg({
    'Systolic_BP': 'mean',        # Average systolic BP for the hour
    'Treatment': 'max',           # Binary treatment indicator
    'Mortality': 'max'            # Mortality remains constant
}).reset_index()

# Rename columns for clarity
time_agg.rename(columns={'Systolic_BP': 'Avg_Systolic_BP'}, inplace=True)

# time_agg.to_csv('time_agg.csv', index=False)

# Check the structure of the aggregated dataset
print(time_agg.head())
```

B2 Code for MIMIC data analysis

```
# Regression Approach

# Import Dataset
file_path = '/Users/qinhan/Desktop/UC Berkeley/Fall 2024/STAT 256/Final 
Project/balanced_time_agg.csv'
balanced_data = pd.read_csv(file_path)

# Define the outcome and treatment variables
outcome = 'Mortality'
treatment = 'Treatment'

# Check unique values in the outcome variable
print(f"Unique values in '{outcome}': {balanced_data[outcome].unique()}")

# Clean the outcome variable to ensure it is binary
valid_values = [0, 1]
balanced_data = balanced_data[balanced_data[outcome].isin(valid_values)]

# Prepare the data for logistic regression
X = balanced_data[[treatment]]  # Predictor: binary treatment
y = balanced_data[outcome]  # Response: binary mortality

# Add an intercept to the logistic regression model
X = sm.add_constant(X)

# Fit the logistic regression model
model = sm.Logit(y, X)
result = model.fit()

# Print the summary of the model
print(result.summary())

# Calculate the predicted probability of mortality for treated and 
untreated groups
balanced_data['Predicted_Prob'] = result.predict(X)

# Calculate the average causal effect (ACE)
# E[Y(1)] - E[Y(0)]
```

```
treated_prob = balanced_data.loc[balanced_data[treatment] == 1,
'Predicted_Prob'].mean()
untreated_prob = balanced_data.loc[balanced_data[treatment] == 0,
'Predicted_Prob'].mean()
ace = treated_prob - untreated_prob

# Output the results
print(f"\nAverage Predicted Probability for Treated (E[Y(1)]):
{treated_prob:.4f}")
print(f"Average Predicted Probability for Untreated (E[Y(0)]):
{untreated_prob:.4f}")
print(f"Average Causal Effect (ACE): {ace:.4f}")
# G formula
# Load the dataset
file_path = '/Users/qinhan/Desktop/UC Berkeley/Fall 2024/STAT 256/Final
Project/balanced_time_agg.csv'
data = pd.read_csv(file_path)

# Define variables
treatment = 'Treatment'
outcome = 'Mortality'
confounder = 'Avg_Systolic_BP'

# Split the data into training and testing sets
train_data, test_data = train_test_split(data, test_size=0.3,
random_state=42)

# Step 1: Model E[Y | Z2, Z1, X1, X0]
outcome_model = LogisticRegression(max_iter=1000, solver='lbfgs')
outcome_model.fit(train_data[[treatment, confounder]],
train_data[outcome])

# Step 2: Model P(X1 | Z1, X0) using the confounder model
confounder_model = GradientBoostingClassifier()
confounder_model.fit(train_data[[treatment]],
train_data[confounder].astype('int'))

# Step 3: Simulate potential outcomes under treatment scenarios
def simulate_potential_outcomes(data, treatment_value):
   simulated_data = data.copy()
```

```
    simulated_data[treatment] = treatment_value
    # Predict confounder X1 (Avg_Systolic_BP) for the given treatment
    simulated_data[confounder] =
confounder_model.predict(data[[treatment]])
    # Predict the outcome based on the treatment and simulated confounder
    simulated_data['Predicted_Outcome'] = outcome_model.predict_proba(
        simulated_data[[treatment, confounder]]
    )[:, 1]
    return simulated_data['Predicted_Outcome'].mean()

# Compute expected outcomes under treatment and no treatment
expected_outcome_treated = simulate_potential_outcomes(test_data,
treatment_value=1)
expected_outcome_untreated = simulate_potential_outcomes(test_data,
treatment_value=0)

# Calculate the Average Causal Effect (ACE)
ace = expected_outcome_treated - expected_outcome_untreated

# Output results
print(f"Expected Outcome (Treated): {expected_outcome_treated:.4f}")
print(f"Expected Outcome (Untreated): {expected_outcome_untreated:.4f}")
print(f"Average Causal Effect (ACE): {ace:.4f}")
# Marginal Structural Model
# Load the dataset
file_path = '/Users/qinhan/Desktop/UC Berkeley/Fall 2024/STAT 256/Final
Project/balanced_time_agg.csv'
data = pd.read_csv(file_path)

# Define variables
treatment = 'Treatment'
outcome = 'Mortality'
confounder = 'Avg_Systolic_BP'

# Split the data into training and testing sets
train_data, test_data = train_test_split(data, test_size=0.3,
random_state=42)

# Step 1: Estimate propensity scores for treatment (e(z | X))
propensity_model = LogisticRegression(max_iter=1000, solver='lbfgs')
```

```python
propensity_model.fit(train_data[[confounder]], train_data[treatment])

# Predict propensity scores
train_data['Propensity_Score'] =
propensity_model.predict_proba(train_data[[confounder]])[:, 1]
test_data['Propensity_Score'] =
propensity_model.predict_proba(test_data[[confounder]])[:, 1]

# Step 2: Compute stabilized weights (IPWs)
train_data['Stabilized_Weight'] = np.where(
   train_data[treatment] == 1,
   1 / train_data['Propensity_Score'],
   1 / (1 - train_data['Propensity_Score'])
)

# Step 3: Fit a weighted logistic regression model (MSM)
# E[Y(Z)] = β0 + β1 * Z
msm_model = LogisticRegression(max_iter=1000, solver='lbfgs')
msm_model.fit(
   train_data[[treatment]],
   train_data[outcome],
   sample_weight=train_data['Stabilized_Weight']
)

# Extract coefficients
intercept = msm_model.intercept_[0]
treatment_effect = msm_model.coef_[0][0]

# Step 4: Compute causal effect
expected_outcome_treated = msm_model.predict_proba([[1]])[0][1]  #
Predicted probability for treated
expected_outcome_untreated = msm_model.predict_proba([[0]])[0][1]  #
Predicted probability for untreated
ace = expected_outcome_treated - expected_outcome_untreated

# Output results
print(f"Intercept (Baseline Mortality): {intercept:.4f}")
print(f"Treatment Effect (β1): {treatment_effect:.4f}")
print(f"Expected Outcome (Treated): {expected_outcome_treated:.4f}")
print(f"Expected Outcome (Untreated): {expected_outcome_untreated:.4f}")
print(f"Average Causal Effect (ACE): {ace:.4f}")
# Structural Nested Model
# Load the dataset
file_path = '/Users/qinhan/Desktop/UC Berkeley/Fall 2024/STAT 256/Final
Project/balanced_time_agg.csv'
data = pd.read_csv(file_path)

# Define variables
treatment = 'Treatment'
outcome = 'Mortality'
confounder = 'Avg_Systolic_BP'

# Split the data into training and testing sets
train_data, test_data = train_test_split(data, test_size=0.3,
random_state=42)

# Step 1: Estimate propensity scores (P(Z_t | H_t))
propensity_model = LogisticRegression(max_iter=1000, solver='lbfgs')
propensity_model.fit(train_data[[confounder]], train_data[treatment])

# Add propensity scores to the dataset
train_data['Propensity_Score'] =
propensity_model.predict_proba(train_data[[confounder]])[:, 1]

# Step 2: Calculate residualized treatment effect
# Compute residualized treatment (Z_t - P(Z_t | H_t))
train_data['Residual_Treatment'] = train_data[treatment] -
train_data['Propensity_Score']

# Step 3: Regress outcome on residualized treatment (G-Estimation)
# Y = β0 + ψ_t * (Z_t - P(Z_t | H_t)) + ε
g_estimation_model = OLS(
    train_data[outcome],
    sm.add_constant(train_data['Residual_Treatment'])
)
g_estimation_result = g_estimation_model.fit()

# Extract coefficient (ψ_t)
psi_t = g_estimation_result.params['Residual_Treatment']
intercept = g_estimation_result.params['const']
```

```
# Step 4: Compute causal effect for each treatment scenario
# Simulate counterfactual outcomes
train_data['Counterfactual_Untreated'] = train_data[outcome] - psi_t *
train_data['Residual_Treatment']
train_data['Counterfactual_Treated'] =
train_data['Counterfactual_Untreated'] + psi_t

# Compute average potential outcomes
expected_outcome_treated = train_data['Counterfactual_Treated'].mean()
expected_outcome_untreated = train_data['Counterfactual_Untreated'].mean()

# Compute Average Causal Effect (ACE)
ace = expected_outcome_treated - expected_outcome_untreated

# Output results
print(f"Coefficient of Residualized Treatment (ψ_t): {psi_t:.4f}")
print(f"Intercept (Baseline Outcome): {intercept:.4f}")
print(f"Expected Outcome (Treated): {expected_outcome_treated:.4f}")
print(f"Expected Outcome (Untreated): {expected_outcome_untreated:.4f}")
print(f"Average Causal Effect (ACE): {ace:.4f}")
```