In [ ]:

```python
# coding: utf-8 (python 3)
# Author -- Ping-Yen Chung


#  import required packages
from scipy.stats import chi2_contingency
from matplotlib.lines import Line2D
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import json
import sklearn
from sklearn.preprocessing import LabelEncoder
import networkx as nx
from sklearn.model_selection import train_test_split
import xgboost as xgb
from sklearn.metrics import make_scorer, accuracy_score, f1_score, recall_score
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import LabelEncoder
from matplotlib.pylab import rcParams
from itertools import combinations
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
import re


''' This is the package for data preprocessing '''
class PredictaVie_Preprocess:
    def __init__(self, dataframe):
        self.dataframe = dataframe
        self.focus_event = None

    # This function can be used to combine the simultaneous events (those that have the same event_date) into one
    def filter_simultaneous_event(self):
        event_sequence_df = pd.DataFrame(columns=['person_id', 'gender'])
        for person_id, group in self.dataframe.groupby('person_id'):
            gender = group['gender'].iloc[0]
            events = group.sort_values(by='event_date')['event'].tolist()
            events_date = group.sort_values(by='event_date')['event_date'].tolist()
            previous_event = None
            event_list = []
            event_date_list = []
            for index, event in enumerate(events):
                if event != previous_event:
                    event_list.append(event)
                    event_date_list.append(events_date[index])
                previous_event = event

            filtered_event_list = []
            filtered_event_date_list = []

            # combine simultaneous event based on date
            for date in set(event_date_list):
```

```python
                combined_event = " + ".join([event_list[i] for i in range(len(event_date_list)) if event_date_list[i] == date])
                filtered_event_list.append(combined_event)
                filtered_event_date_list.append(date)

            event_dict = {'person_id': person_id, 'gender': gender, 'event': filtered_event_list, 'event_date': filtered_event_date_list}
            event_sequence_df = pd.concat([event_sequence_df, pd.DataFrame(event_dict)], ignore_index=True)
        self.filterdata = event_sequence_df.sort_values(by=['person_id', 'event_date'])
        return self.filterdata

    ''' This function below is visualization or so to let users know and check the dataset '''
    # this function can be called to see the distribution (counts) of each catogories within the column "columns_name" in the dataset
    # ratio_setting is the baseline of "rare", those that are below the baseline will be named as "other"
    # ratio_setting is automatically set to 0%, but can be adjusted when calling the function
    def event_pie_chart(self, column_name, ratio_setting=0.00):
        if not hasattr(self, 'filterdata'):
            raise AttributeError("The 'filterdata' has not been generated. Please call 'filter_simultaneous_event' method first.")

        counts = self.filterdata[column_name].value_counts()
        total_count = counts.sum()
        ratios = counts / total_count
        significant_events = ratios[ratios > ratio_setting].index.tolist()
        other_ratio = ratios[ratios <= ratio_setting].sum()
        plt.figure(figsize=(8, 6))
        plt.pie(ratios[ratios > ratio_setting].values.tolist() + [other_ratio], labels=significant_events + ['others'], autopct='%1.1f%%', startangle=140)
        plt.title('Distribution of ' + column_name + ' in the target dataset:')
        plt.show()

    # this function can be called to filter out the rare catogories within the column "columns_name" in the dataset
    # ratio_setting is the baseline of "rare", it is automatically set to 0%, but can be adjusted when calling the function
    def filter_data_by_significant_events(self, column_name, ratio_setting=0.00):
        if not hasattr(self, 'filterdata'):
            raise AttributeError("The 'filterdata' has not been generated. Please call 'filter_simultaneous_event' method first.")

        significant_events = self.filterdata[column_name].value_counts(normalize=True)[self.filterdata[column_name].value_counts(normalize=True) > ratio_setting].index.tolist()
        significant_events_data = self.filterdata[self.filterdata[column_name].isin(significant_events)]
        self.filterdata = significant_events_data
        return self.filterdata

    ''' This function below is visualization or so to let users know and check the dataset '''
    # this function can be called to show the condition journey of one person (person_id) in a dataframe way
    def ind_event_sequence(self, person_id):
        if not hasattr(self, 'filterdata'):
            raise AttributeError("The 'filterdata' has not been generated. Please call 'filter_simultaneous_event' method first.")

        df = self.filterdata
        df['event_date'] = pd.to_datetime(df['event_date'])
        df0 = df[df['person_id'] == person_id]
        df1 = df0.sort_values(by='event_date')
        df2 = df1[df1['event'] != df1['event'].shift(-1)]
        df2.reset_index(drop=True, inplace=True)
        return df2

    ''' This function below is visualization or so to let users know and check the dataset '''
    # this function can be called to show the condition journey of one person (person_id) in a graph way
    def plot_journey(self, person_id):
        if not hasattr(self, 'filterdata'):
            raise AttributeError("The 'filterdata' has not been generated. Please call 'filter_simultaneous_event' method first.")
```

```python
        df = self.filterdata
        df['event_date'] = pd.to_datetime(df['event_date'])
        person_data = df[df['person_id'] == person_id]
        person_data_sorted = person_data.sort_values(by='event_date')
        person_data_unique = person_data_sorted[person_data_sorted['event'] != person_data_sorted['event'].shift(-1)]
        person_data_unique.reset_index(drop=True, inplace=True)
        G = nx.DiGraph()
        for _, row in person_data_unique.iterrows():
            G.add_node(row['event'])
        for i in range(len(person_data_unique) - 1):
            current_event = person_data_unique.iloc[i]['event']
            next_event = person_data_unique.iloc[i + 1]['event']
            G.add_edge(current_event, next_event)
        pos = nx.circular_layout(G)
        nx.draw(G, pos, with_labels=True, node_size=2000, node_color='skyblue', font_size=7, font_weight='bold',
                arrows=True, arrowsize=20)
        plt.title('Event Sequence for Person ID: ' + str(person_id))
        plt.show()

    ''' From here the data preprocessing output dataframe that will be used in further steps '''
    # this function can be called to create a dataframe showing the conditions of each patient sorting by svc_date
    # condition sequence will be the columns
    # the last column, which is "last condition" shows the last condition the patient was in
    # the last condition won't be included in the condition sequence (the columns named "condition_n" will not show the last condition)


    def focus_condition(self, final_event_name, focus, occur_number=1):
        if not hasattr(self, 'filterdata'):
            raise AttributeError("The 'filterdata' data has not been generated. Please call 'filter_simultaneous_event' method first.")

        # Sort the DataFrame by person_id and event_date
        df = self.filterdata
        df.sort_values(by=['person_id', 'event_date'], inplace=True)
        occur = int(occur_number)
        if occur < 1:
            print('must enter number at least value 1')

        # Filter out those patients who don't have the target condition
        ids = []
        for person_id, group in df.groupby('person_id'):
            if final_event_name not in group['event'].values:
                ids.append(person_id)
        df = df[~df['person_id'].isin(ids)]

        # Filter out thos patients who don't have the focus condition
        IDS = []
        for person_id, group in df.groupby('person_id'):
            if focus not in group['event'].values:
                IDS.append(person_id)
        df = df[~df['person_id'].isin(IDS)]

        # Filter out patients whose first event is the specified event (if pick the first occurence)
        if occur == 1:
            first_events = df.groupby('person_id').head(1)
            patients_to_exclude = first_events[first_events['event'] == final_event_name]['person_id']
            # Exclude these patients from the dataframe
            df = df[~df['person_id'].isin(patients_to_exclude)]
```

```
    # Filter out patients whose chosen occurance of the specified event is the last event in their journey
    patient_id = []
    for person_id, group in df.groupby('person_id'):
        chosen_event_index = None
        last_event_index = None
            # Get the index for the first occurrence of the specified event
        if group[group['event'] == final_event_name].shape[0] >= occur:
            chosen_event_index = group[group['event'] == final_event_name].index[occur-1]
            last_event_index = group[group['event'] == final_event_name].index[-1]
            if chosen_event_index == last_event_index:
                patient_id.append(person_id)
    df = df[~df['person_id'].isin(patient_id)]

# exclude those patients whose non-continous-occurence of the target event is less than the number picked
    patient_ids = []
    for person_id, group in df.groupby('person_id'):
        previous_event = None
        event_list = []
        events = group.sort_values(by='event_date')['event'].tolist()
        for event in events:
            if event != previous_event:
                event_list.append(event)
            previous_event = event
        if event_list.count(final_event_name) < occur:
            patient_ids.append(person_id)
    df = df[~df['person_id'].isin(patient_ids)]

# Create a new dataframe to hold history until the specified event
    event_sequence_df = pd.DataFrame(columns=['person_id', 'gender'])
    last_event_df = pd.DataFrame(columns=['person_id'])
    for person_id, group in df.groupby('person_id'):
        gender = group['gender'].iloc[0]
        events = group.sort_values(by='event_date')['event'].tolist()
        previous_event = None
        event_list = []
        for event in events:
            if event != previous_event:
                event_list.append(event)
            previous_event = event
        # get the index of the target event
        num = [i for i,x in enumerate(event_list) if x == final_event_name][occur-1]
        # get the index of the focus event that is the closest after the target event
        focus_index_list_after_finalevent = [index for index in [i for i,x in enumerate(event_list) if x == focus] if index > num]
        if focus_index_list_after_finalevent:
            num1 = min(focus_index_list_after_finalevent)
        else:
            num1 = None

        event_sequence = {'person_id': person_id, 'gender': gender}
        sequence_count = 0

        if num1 != None:
            for i in event_list[:num+1]:
                sequence_count += 1
                event_sequence[f'condition_{sequence_count}'] = i
            event_sequence_df = pd.concat([event_sequence_df, pd.DataFrame(event_sequence, index=[0])],ignore_index=True)
            last_event_df = pd.concat([last_event_df, pd.DataFrame({'person_id': [person_id],'last_condition': f'{num1+1}: {[event_list[num1]]}'})],ignore_index=True)
        else:
```

```
            for i in event_list[:num+1]:
                sequence_count += 1
                event_sequence[f'condition_{sequence_count}'] = i
            event_sequence_df = pd.concat([event_sequence_df, pd.DataFrame(event_sequence, index=[0])],ignore_index=True)
            last_event_df = pd.concat([last_event_df, pd.DataFrame({'person_id': [person_id],'last_condition': 'No Focus Events Happen'})],ignore_index=True)

    '''result = []
    for _, row in event_sequence_df.iterrows():
        indices = [index for index, item in enumerate(row) if item == final_event_name]
        if len(indices) >= occur:
            target_occurence = indices[occur-1]
            result.append(list(row[:target_occurence + 1]) + [np.nan] * (len(row) - target_occurence - 1))

    resultdf = pd.DataFrame(result, columns=event_sequence_df.columns)'''

    all_together = pd.merge(event_sequence_df, last_event_df, on='person_id', how='inner')
    self.focus_event = all_together
    return self.focus_event

# this function is used to create condition sequence if the focus_condition function is not applied
def create_event_sequence(self):
    if self.focus_event is None:
        if not hasattr(self, 'filterdata'):
            raise AttributeError("The 'filterdata' data has not been generated. Please call 'filter_simultaneous_event' method first.")

        event_sequence_df = pd.DataFrame(columns=['person_id', 'gender'])
        last_event_df = pd.DataFrame(columns=['person_id'])
        for person_id, group in self.filterdata.groupby('person_id'):
            gender = group['gender'].iloc[0]
            events = group.sort_values(by='event_date')['event'].tolist()
            previous_event = None
            event_list = []
            for event in events:
                if event != previous_event:
                    event_list.append(event)
                previous_event = event
            event_sequence = {'person_id': person_id, 'gender': gender}
            sequence_count = 0
            for i in event_list[:-1]:
                sequence_count += 1
                event_sequence[f'condition_{sequence_count}'] = i
            event_sequence_df = pd.concat([event_sequence_df, pd.DataFrame(event_sequence, index=[0])],
                                          ignore_index=True)
            last_event_df = pd.concat([last_event_df, pd.DataFrame({'person_id': [person_id],
                                                                    'last_condition': [event_list[-1]]})],
                                      ignore_index=True)
        all_together = pd.merge(event_sequence_df, last_event_df, on='person_id', how='inner')
        self.event_sequence = all_together
    else:
        print("No need to use this function as the focus_condition function is applied. Below is the dataframe from that function.")
        self.event_sequence = self.focus_event
    return self.event_sequence

# this function can be used to filter out those patients who has short condition journey
# the top_numbers is automatically set to 2000 but can be adjusted when calling the function
def filter_long_journey(self, top_numbers=2000):
    if not hasattr(self, 'event_sequence'):
        raise AttributeError("The 'event_sequence' attribute has not been generated. Please call 'create_event_sequence' method first.")
```

```python
        data = self.event_sequence
        data['null_count'] = data.isnull().sum(axis=1)
        sort = data.sort_values(by='null_count', ascending=True)
        sort.reset_index(drop=True, inplace=True)
        result = sort.iloc[:, :-1].head(top_numbers)
        self.long_journey = result

        return result

    # this function can be used to generate condition into condition pairs and output a dataframe
    # one patient may appear more than once in the dataframe output if he/she has multiple condition pairs
    def generate_pairs(self):
        if not hasattr(self, 'long_journey'):
            raise AttributeError("The 'long_journey' attribute has not been generated. Please call 'filter_long_journey' method first.")

        df = self.long_journey
        pairs = []
        for index, row in df.iterrows():
            person_id = row['person_id']
            gender = row['gender']
            last = row['last_condition']
            conditions = [col for col in row.iloc[2:-1] if pd.notnull(col)]

            for pair in combinations(enumerate(conditions, 1), 2): #pairing two conditions into a pair, and this step consider the sequence as well
                # e.g. we want condition_1 -> condition_3 (sequence considered), not condition_3 -> condition_1 (no sequence)
                # pair[0] and pair[1] are tuples containing (index, condition)
                index1, condition1 = pair[0]
                index2, condition2 = pair[1]
                if condition1 != condition2:  # avoid having same conditions paired
                    pair_str = f"{condition1} -> {condition2}"
                    start_index = index1
                    avg_position = ((index1) + (index2)) / 2
                    pair_str = pair_str.replace('[', '').replace(']', '')
                    pairs.append({'person_id': person_id, 'gender': gender, 'pair': pair_str,
                                  'last_condition': last, 'pair_start': start_index,
                                  'avg_pair_position': avg_position})

        pairs_df = pd.DataFrame(pairs)
        self.pairs = pairs_df
        return pairs_df

    # this function can be called to make condition pairs into "starting part of pairs" columns, showing the in which parts of the medical journey
    # do pairs appear (1: <=25%, 2: >25% & <=50%, 3: >50% & <=75%, 4: >75%)
    def pairs_to_startpos_part(self):
        if not hasattr(self, 'pairs'):
            raise AttributeError("The 'pairs' attribute has not been generated. Please call 'generate_pairs' method first.")

        df = self.pairs
        pair_types = df['pair'].unique()
        pair_start_part_df = pd.DataFrame(columns=['person_id', 'IS_MALE'] + list(pair_types))

        for person_id, group in df.groupby('person_id'):
            gender = group['gender'].iloc[0]
            last = group['last_condition'].iloc[0]

            pair_indices = group.groupby('pair')['pair_start'].mean().reset_index()
```

```
            pair_start_part_dict = {'person_id': person_id, 'last_condition': last}

            if 'MALE' == gender:
                pair_start_part_dict['IS_MALE'] = 1
            else:
                pair_start_part_dict['IS_MALE'] = 0

            # get the starting point of the "latest condition pairs" to assume it as the indicator of the length of a patient's condition journey
            length = group['pair_start'].max()-group['pair_start'].min()
            start = group['pair_start'].min()

            for pair_type in pair_types:
                if pair_type in pair_indices['pair'].values:
                    # get the starting position of the condition pairs
                    position = round(pair_indices.loc[pair_indices['pair'] == pair_type, 'pair_start'].values[0], 2)
                    if position <= round(start+(length/4),2):
                        pair_start_part_dict[pair_type] = 1
                    elif round(length/4,2) < position <= round(start+2*(length/4),2):
                        pair_start_part_dict[pair_type] = 2
                    elif round(2*length/4,2) < position <= round(start+3*(length/4),2):
                        pair_start_part_dict[pair_type] = 3
                    else:
                        pair_start_part_dict[pair_type] = 4
                else:
                    pair_start_part_dict[pair_type] = 0
            pair_start_part_df = pd.concat([pair_start_part_df, pd.DataFrame([pair_start_part_dict])], ignore_index=True)
            pair_start_part_df.fillna(0, inplace=True)

        return pair_start_part_df

    # this function can be called to make condition pairs into "starting position of pairs" columns, showing the average starting point of pairs
    # those pairs that appear more than one time will be counted to show how many times the patient move back and forth between 2 conditions
    def pairs_to_startpos(self):
        if not hasattr(self, 'pairs'):
            raise AttributeError("The 'pairs' attribute has not been generated. Please call 'generate_pairs' method first.")

        df = self.pairs
        pair_types = df['pair'].unique()
        pair_start_df = pd.DataFrame(columns=['person_id', 'IS_MALE'] + list(pair_types))

        for person_id, group in df.groupby('person_id'):
            gender = group['gender'].iloc[0]
            last = group['last_condition'].iloc[0]

            pair_indices = group.groupby('pair')['pair_start'].mean().reset_index()

            pair_start_dict = {'person_id': person_id, 'last_condition': last}

            if 'MALE' == gender:
                pair_start_dict['IS_MALE'] = 1
            else:
                pair_start_dict['IS_MALE'] = 0

            for pair_type in pair_types:
                if pair_type in pair_indices['pair'].values:
                    pair_start_dict[pair_type] = round(pair_indices.loc[pair_indices['pair'] == pair_type, 'pair_start'].values[0], 2)
                else:
                    pair_start_dict[pair_type] = 0
```

```
        pair_start_df = pd.concat([pair_start_df, pd.DataFrame([pair_start_dict])], ignore_index=True)
        pair_start_df.fillna(0, inplace=True)

    return pair_start_df

# this function can be called to make condition pairs into "avg position part of pairs" columns, showing the in which parts of the medical
# journey do pairs appear (1: <=25%, 2: >25% & <=50%, 3: >50% & <=75%, 4: >75%)
def pairs_to_avgpos_part(self):
    if not hasattr(self, 'pairs'):
        raise AttributeError("The 'pairs' attribute has not been generated. Please call 'generate_pairs' method first.")

    df = self.pairs
    pair_types = df['pair'].unique()
    pair_avg_part_df = pd.DataFrame(columns=['person_id', 'IS_MALE'] + list(pair_types))

    for person_id, group in df.groupby('person_id'):
        gender = group['gender'].iloc[0]
        last = group['last_condition'].iloc[0]

        pair_indices = group.groupby('pair')['avg_pair_position'].mean().reset_index()

        pair_avg_part_dict = {'person_id': person_id, 'last_condition': last}

        if 'MALE' == gender:
            pair_avg_part_dict['IS_MALE'] = 1
        else:
            pair_avg_part_dict['IS_MALE'] = 0

        # get the max average point of the "latest condition pairs" to assume it
        # as the indicator of the length of a patient's condition journey
        length = group['avg_pair_position'].max()

        for pair_type in pair_types:
            if pair_type in pair_indices['pair'].values:
                # get the average position of the condition pairs
                position = round(pair_indices.loc[pair_indices['pair'] == pair_type, 'avg_pair_position'].values[0], 2)
                if position <= round(length/4,2):
                    pair_avg_part_dict[pair_type] = 1
                elif round(length/4,2) < position <= round(2*length/4,2):
                    pair_avg_part_dict[pair_type] = 2
                elif round(2*length/4,2) < position <= round(3*length/4,2):
                    pair_avg_part_dict[pair_type] = 3
                else:
                    pair_avg_part_dict[pair_type] = 4
            else:
                pair_avg_part_dict[pair_type] = 0
        pair_avg_part_df = pd.concat([pair_avg_part_df, pd.DataFrame([pair_avg_part_dict])], ignore_index=True)
        pair_avg_part_df.fillna(0, inplace=True)

    return pair_avg_part_df

# this function can be called to make condition pairs into "average position of pairs" columns, showing the average weighted point of pairs
# those pairs that appear more than one time will be counted to show how many times the patient move back and forth between 2 conditions
def pairs_to_avgpos(self):
    if not hasattr(self, 'pairs'):
        raise AttributeError("The 'pairs' attribute has not been generated. Please call 'generate_pairs' method first.")

    df = self.pairs
```

```python
        pair_types = df['pair'].unique()
        pair_avg_df = pd.DataFrame(columns=['person_id', 'IS_MALE'] + list(pair_types))

        for person_id, group in df.groupby('person_id'):
            gender = group['gender'].iloc[0]
            last = group['last_condition'].iloc[0]

            pair_indices = group.groupby('pair')['avg_pair_position'].mean().reset_index()

            pair_avg_dict = {'person_id': person_id, 'last_condition': last}

            if 'MALE' == gender:
                pair_avg_dict['IS_MALE'] = 1
            else:
                pair_avg_dict['IS_MALE'] = 0

            for pair_type in pair_types:
                if pair_type in pair_indices['pair'].values:
                    pair_avg_dict[pair_type] = round(pair_indices.loc[pair_indices['pair'] == pair_type, 'avg_pair_position'].values[0], 2)
                else:
                    pair_avg_dict[pair_type] = 0
            pair_avg_df = pd.concat([pair_avg_df, pd.DataFrame([pair_avg_dict])], ignore_index=True)
            pair_avg_df.fillna(0, inplace=True)

        return pair_avg_df

''' This is the package for data splitting '''
class PredictaVie_SplitData:
    def __init__(self, dataframe):
        self.dataframe = dataframe

    # this function can be called to split the data into training and testing
    # test_size and random_state are automatically set to 0.2 and 42, respectively. These numbers can be adjusted when calling the function
    # in this function, the condition pairs that happen before the last condition will be the input of the models
    # the last condition will be the output (prediciton) of the models
    def c2c_split_data(self, test_size=0.2, random_state=42):
        df = self.dataframe
        exclude_column = ['person_id', 'last_condition']
        filtered_columns = [col for col in df.columns if col not in exclude_column]
        y = df['last_condition']
        x = df.loc[:, filtered_columns].astype(int)
        x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size, random_state=random_state)
        return x_train, x_test, y_train, y_test

''' This is the package for model building and evaluating '''
class PredictaVie_Model:
    def __init__(self, x_train, y_train, x_test, y_test):
        self.x_train = x_train
        self.y_train = y_train
        self.x_test = x_test
        self.y_test = y_test

    # this function can be called to train the best XGBoost model by using grid search
    # cross validation setting is set to 5 but can be adjusted when calling the function
    # the parameters of grid search are automatically set as below:
    # param_grid = {
    #             'n_estimators': [100, 150, 200],
    #             'max_depth': [3, 4, 5],
```

```python
    #             'learning_rate': [0.1, 0.01, 0.001]
    #         }
    # and this can be adjusted when calling the function as well
    # refit_setting is automatically set to 'accuracy' to allow grid_search use average accuracy score as the standard of choosing the best
    def c2c_train_xgb(self, param_grid=None, cv=5, refit_setting='accuracy'):
        label_encoder = LabelEncoder()
        y_train_encoded = label_encoder.fit_transform(self.y_train)

        if param_grid is None:
            param_grid = {
                'n_estimators': [100, 150, 200],
                'max_depth': [3, 4, 5],
                'learning_rate': [0.1, 0.01, 0.001]
            }

        model = xgb.XGBClassifier()
        grid_search = GridSearchCV(model, param_grid, cv=cv, refit=refit_setting)
        grid_search.fit(self.x_train, y_train_encoded)

        best_model = grid_search.best_estimator_
        best_params = grid_search.best_params_
        best_score = grid_search.best_score_

        self.best_xgbmodel = best_model
        self.train_label_encoder = label_encoder

        return best_model, label_encoder

    # this function can be called to evaluate the XGBoost model get from function "c2c_train_xgb"
    # this will show the confusion matrix of the result using the model on testing data
    def c2c_evaluate_xgb(self):
        if not hasattr(self, 'best_xgbmodel'):
            raise AttributeError("The 'best_xgbmodel' has not been generated. Please call 'c2c_train_xgb' method first.")

        best = self.best_xgbmodel

        label_encoder = LabelEncoder()
        y_test_encoded = label_encoder.fit_transform(self.y_test)

        y_pred = best.predict(self.x_test)

        accuracy = accuracy_score(y_test_encoded, y_pred)
        print("Test Accuracy:", accuracy)

        original_labels = label_encoder.inverse_transform(y_test_encoded)
        predicted_labels = label_encoder.inverse_transform(y_pred)
        cm = confusion_matrix(original_labels, predicted_labels)

        cm_df = pd.DataFrame(cm, index=label_encoder.classes_, columns=label_encoder.classes_)

        plt.figure(figsize=(10, 8))
        sns.heatmap(cm_df, annot=True, cmap="YlGnBu", fmt="d")
        plt.xlabel('Predicted')
        plt.ylabel('Actual')
        plt.title('Confusion Matrix')
        plt.show()

        f1 = f1_score(y_test_encoded, y_pred, average='weighted')
```

```python
        recall = recall_score(y_test_encoded, y_pred, average='weighted')

        print("F1 Score:", f1)
        print("Recall Score:", recall)

    # this function can be called to plot the feature importance of the top n condition pairs and output (a list) of them
    # n (number) is automatically set to 20 and can be adjusted when calling the function
    def c2c_xgb_feature_importance(self, number=20):
        if not hasattr(self, 'best_xgbmodel'):
            raise AttributeError("The 'best_xgbmodel' has not been generated. Please call 'c2c_train_xgb' method first.")

        xgb_model = self.best_xgbmodel
        feature_importance = xgb_model.feature_importances_
        # the .feature_importances_ get the score that can rate the importance of features based on "how many times this feature is used--
        # --to split the tree, and how many targets this feature exclude when being a split node"
        # the more a feature is used as splitting tree and the more targets it can exclude when being a split node, the higher the score
        feature_names = xgb_model.get_booster().feature_names
        feature_importance_dict = dict(zip(feature_names, feature_importance))
        sorted_feature_importance = sorted(feature_importance_dict.items(), key=lambda x: x[1], reverse=True)

        features = [x[0] for x in sorted_feature_importance[:number]]
        importance = [x[1] for x in sorted_feature_importance[:number]]

        plt.figure(figsize=(10, number/2))
        plt.barh(features, importance, color='skyblue')
        plt.xlabel('Feature Importance')
        plt.ylabel('Features')
        plt.title('XGBoost Feature Importance')
        plt.gca().invert_yaxis()
        plt.show()
        top_feature_indices = [item[0] for item in sorted_feature_importance[:number]]
        return top_feature_indices

    # this function can be called to train the best logistic regression model by grid search (cross validation set to 5 (adjustable))
    # also this function can output (a list) and plot the top n important condition pairs
    # the parameters of the grid serach are automatically set to:
    # param_grid = {
    #             'penalty': ['l1', 'l2', 'elasticnet', 'none'],
    #             'C': [0.001, 0.01, 0.1, 1, 10, 100],
    #             'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
    #         }
    # and this can be adjusted when calling the function as well
    # refit_setting is automatically set to 'accuracy' to allow grid_search use average accuracy score as the standard of choosing the best
    # n (number) is automatically set to 20 and can be adjusted when calling the function
    def c2c_train_logreg(self, param_grid=None, cross_validation=5, refit_setting='accuracy', number=20):
        logistic_reg = LogisticRegression()

        if param_grid is None:
            param_grid = {
                'penalty': ['l1', 'l2', 'elasticnet'],
                'C': [0.001, 0.01, 0.1, 1, 10, 100],
                'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
            }

        grid_search = GridSearchCV(estimator=logistic_reg, param_grid=param_grid, cv=cross_validation, refit=refit_setting)
        grid_search.fit(self.x_train, self.y_train)

        print("Best Parameters:", grid_search.best_params_)
```

```
        best_model = grid_search.best_estimator_

        feature_names = list(self.x_train.columns)
        feature_weights = best_model.coef_[0]
        abs_feature_weights = np.abs(feature_weights)
        top_indices = np.argsort(abs_feature_weights)[::-1][:number]
        top_features = [feature_names[i] for i in top_indices]
        top_weights = [feature_weights[i] for i in top_indices]

        # Plotting the top features
        plt.figure(figsize=(10, number/2))
        plt.barh(top_features, top_weights, color='skyblue')
        plt.xlabel('Feature Weight')
        plt.ylabel('Features')
        plt.title('Logistic Regression Top Feature Weights')
        plt.gca().invert_yaxis()
        plt.show()

        self.best_logregmodel = best_model

        return best_model, top_features

    # this function can be called to evaluate the result of the logistic regression model get from function "c2c_train_logreg"
    # this will show the confusion matrix of the result using the model on testing data
    def c2c_evaluate_logreg(self):
        if not hasattr(self, 'best_logregmodel'):
            raise AttributeError("The 'best_logregmodel' has not been generated. Please call 'c2c_train_logreg' method first.")

        best = self.best_logregmodel

        y_pred = best.predict(self.x_test)

        accuracy = accuracy_score(self.y_test, y_pred)
        print("Test Accuracy:", accuracy)

        cm = confusion_matrix(self.y_test, y_pred, labels=best.classes_)
        cm_df = pd.DataFrame(cm, index=best.classes_, columns=best.classes_)

        plt.figure(figsize=(10, 8))
        sns.heatmap(cm_df, annot=True, cmap="Blues", fmt='g')
        plt.xlabel('Predicted')
        plt.ylabel('Actual')
        plt.title('Confusion Matrix Heatmap')
        plt.show()

        f1 = f1_score(self.y_test, y_pred, average='weighted')
        recall = recall_score(self.y_test, y_pred, average='weighted')

        print("F1 Score:", f1)
        print("Recall Score:", recall)
```

```python
# package for future user to make predictions
class PredictaVie_Predict:
    def __init__(self, df, xgbmodel_start, xgbmodel_avg, label_encoder_start, label_encoder_avg, logregmodel_start, logregmodel_avg):
        self.dataframe = df
        self.xgbmodel_start = xgbmodel_start
        self.xgbmodel_avg = xgbmodel_avg
        self.label_encoder_start = label_encoder_start
        self.label_encoder_avg = label_encoder_avg
        self.logregmodel_start = logregmodel_start
        self.logregmodel_avg = logregmodel_avg
        self.focus_event = None

    # this function can be called to combine all simultaneous events into one using "+"
    def filter_simultaneous_event(self):
        event_sequence_df = pd.DataFrame(columns=['person_id', 'gender'])
        for person_id, group in self.dataframe.groupby('person_id'):
            gender = group['gender'].iloc[0]
            events = group.sort_values(by='event_date')['event'].tolist()
            events_date = group.sort_values(by='event_date')['event_date'].tolist()
            previous_event = None
            event_list = []
            event_date_list = []
            for index, event in enumerate(events):
                if event != previous_event:
                    event_list.append(event)
                    event_date_list.append(events_date[index])
                previous_event = event

            filtered_event_list = []
            filtered_event_date_list = []

            # combine simultaneous event based on date
            for date in set(event_date_list):
                combined_event = " + ".join([event_list[i] for i in range(len(event_date_list)) if event_date_list[i] == date])
                filtered_event_list.append(combined_event)
                filtered_event_date_list.append(date)

            event_dict = {'person_id': person_id, 'gender': gender, 'event': filtered_event_list, 'event_date': filtered_event_date_list}
            event_sequence_df = pd.concat([event_sequence_df, pd.DataFrame(event_dict)], ignore_index=True)
            self.filterdata = event_sequence_df.sort_values(by=['person_id', 'event_date'])
        return self.filterdata

    def focus_condition(self, final_event_name, occur_number=1):
        if not hasattr(self, 'filterdata'):
            raise AttributeError("The 'filterdata' data has not been generated. Please call 'filter_simultaneous_event' method first.")

        # Sort the DataFrame by person_id and event_date
        df = self.filterdata
        df.sort_values(by=['person_id', 'event_date'], inplace=True)
        occur = int(occur_number)
        if occur < 1:
            print('must enter number at least value 1')

        # Filter out those patients who don't have the target condition
        ids = []
```

```python
        for person_id, group in df.groupby('person_id'):
            if final_event_name not in group['event'].values:
                ids.append(person_id)
        df = df[~df['person_id'].isin(ids)]

        # Filter out patients whose first event is the specified event (if pick the first occurence)
        if occur == 1:
            first_events = df.groupby('person_id').head(1)
            patients_to_exclude = first_events[first_events['event'] == final_event_name]['person_id']
            # Exclude these patients from the dataframe
            df = df[~df['person_id'].isin(patients_to_exclude)]


    # exclude those patients whose non-continous-occurence of the target event is less than the number picked
        patient_ids = []
        for person_id, group in df.groupby('person_id'):
            previous_event = None
            event_list = []
            events = group.sort_values(by='event_date')['event'].tolist()
            for event in events:
                if event != previous_event:
                    event_list.append(event)
                previous_event = event
            if event_list.count(final_event_name) < occur:
                patient_ids.append(person_id)
        df = df[~df['person_id'].isin(patient_ids)]

    # Create a new dataframe to hold history until the specified event
        event_sequence_df = pd.DataFrame(columns=['person_id', 'gender'])
        last_event_df = pd.DataFrame(columns=['person_id'])
        for person_id, group in df.groupby('person_id'):
            gender = group['gender'].iloc[0]
            events = group.sort_values(by='event_date')['event'].tolist()
            previous_event = None
            event_list = []
            for event in events:
                if event != previous_event:
                    event_list.append(event)
                previous_event = event
            # get the index of the target event
            num = [i for i,x in enumerate(event_list) if x == final_event_name][occur-1]
            for i in event_list[:num+1]:
                sequence_count += 1
                event_sequence[f'condition_{sequence_count}'] = i
            event_sequence_df = pd.concat([event_sequence_df, pd.DataFrame(event_sequence, index=[0])],ignore_index=True)

        self.focus_event = event_sequence_df
        return self.focus_event

    # this function is used to create condition sequence if the focus_condition function is not applied
    def create_event_sequence(self):
        if self.focus_event is None:
            if not hasattr(self, 'filterdata'):
                raise AttributeError("The 'filterdata' data has not been generated. Please call 'filter_simultaneous_event' method first.")

            event_sequence_df = pd.DataFrame(columns=['person_id', 'gender'])
            last_event_df = pd.DataFrame(columns=['person_id'])
            for person_id, group in self.filterdata.groupby('person_id'):
```

```python
                gender = group['gender'].iloc[0]
                events = group.sort_values(by='event_date')['event'].tolist()
                previous_event = None
                event_list = []
                for event in events:
                    if event != previous_event:
                        event_list.append(event)
                    previous_event = event
                event_sequence = {'person_id': person_id, 'gender': gender}
                sequence_count = 0
                for i in event_list[:-1]:
                    sequence_count += 1
                    event_sequence[f'condition_{sequence_count}'] = i
                event_sequence_df = pd.concat([event_sequence_df, pd.DataFrame(event_sequence, index=[0])],
                                              ignore_index=True)
                last_event_df = pd.concat([last_event_df, pd.DataFrame({'person_id': [person_id],
                                                                         'last_condition': [event_list[-1]]})],
                                          ignore_index=True)
            all_together = pd.merge(event_sequence_df, last_event_df, on='person_id', how='inner')
            self.event_sequence = all_together
        else:
            print("No need to use this function as the focus_condition function is applied. Below is the dataframe from that function.")
            self.event_sequence = self.focus_event
        return self.event_sequence

    # this function can be used to generate condition into condition pairs and output a dataframe
    # one patient may appear more than once in the dataframe output if he/she has multiple condition pairs
    def generate_pairs(self):
        if not hasattr(self, 'event_sequence'):
            raise AttributeError("The 'event_sequence' attribute has not been generated. Please call 'create_event_sequence' method first.")

        df = self.event_sequence
        pairs = []
        for index, row in df.iterrows():
            person_id = row['person_id']
            gender = row['gender']
            conditions = [col for col in row.iloc[2:] if pd.notnull(col)]

            for pair in combinations(enumerate(conditions, 1), 2): #pairing two conditions into a pair, and this step consider the sequence as well
                # e.g. we want condition_1 -> condition_3 (sequence considered), not condition_3 -> condition_1 (no sequence)
                index1, condition1 = pair[0]
                index2, condition2 = pair[1]
                if condition1 != condition2:  # avoid having same conditions paired
                    pair_str = f"{condition1} -> {condition2}"
                    start_index = index1
                    avg_position = ((index1) + (index2)) / 2
                    pair_str = pair_str.replace('[', '').replace(']', '')
                    pairs.append({'person_id': person_id, 'gender': gender, 'pair': pair_str, 'pair_start': start_index,
                                  'avg_pair_position': avg_position})

        pairs_df = pd.DataFrame(pairs)
        self.pairs = pairs_df

        return pairs_df

    # this function can be called to make condition pairs into "starting position part of pairs" columns, showing the in which parts of the medical
    # journey do pairs appear (1: <=25%, 2: >25% & <=50%, 3: >50% & <=75%, 4: >75%)
    def pairs_to_startpos_part(self):
```

```python
        if not hasattr(self, 'pairs'):
            raise AttributeError("The 'pairs' attribute has not been generated. Please call 'generate_pairs' method first.")

        df = self.pairs
        train_pair_type = list(self.xgbmodel_start.get_booster().feature_names)
        pair_start_part_df = pd.DataFrame(columns=['person_id'] + train_pair_type)

        for person_id, group in df.groupby('person_id'):
            gender = group['gender'].iloc[0]
            pair_indices = group.groupby('pair')['pair_start'].mean().reset_index()
            pair_start_part_dict = {'person_id': person_id}

            if 'MALE' == gender:
                pair_start_part_dict['IS_MALE'] = 1
            else:
                pair_start_part_dict['IS_MALE'] = 0

            # get the starting point of the "latest condition pairs" to assume it as the indicator of the length of a patient's condition journey
            length = group['pair_start'].max()-group['pair_start'].min()
            start = group['pair_start'].min()

            for pair_type in train_pair_type:
                if pair_type in pair_indices['pair'].values:
                    # get the starting position of the condition pairs
                    position = round(pair_indices.loc[pair_indices['pair'] == pair_type, 'pair_start'].values[0], 2)
                    if position <= round(start+(length/4),2):
                        pair_start_part_dict[pair_type] = 1
                    elif round(length/4,2) < position <= round(start+2*(length/4),2):
                        pair_start_part_dict[pair_type] = 2
                    elif round(2*length/4,2) < position <= round(start+3*(length/4),2):
                        pair_start_part_dict[pair_type] = 3
                    else:
                        pair_start_part_dict[pair_type] = 4
                else:
                    pair_start_part_dict[pair_type] = 0

            pair_start_part_df = pd.concat([pair_start_part_df, pd.DataFrame([pair_start_part_dict])], ignore_index=True)
            pair_start_part_df.fillna(0, inplace=True)

        self.pair_start_part_df = pair_start_part_df
        return pair_start_part_df

    # this function can be called to make condition pairs into "starting position of pairs" columns, showing the average starting point of pairs
    # those pairs that appear more than one time will be counted to show how many times the patient move back and forth between 2 conditions
    def pairs_to_startpos(self):
        if not hasattr(self, 'pairs'):
            raise AttributeError("The 'pairs' attribute has not been generated. Please call 'generate_pairs' method first.")

        df = self.pairs
        train_pair_type = list(self.xgbmodel_start.get_booster().feature_names)
        pair_start_df = pd.DataFrame(columns=['person_id'] + train_pair_type)

        for person_id, group in df.groupby('person_id'):
            gender = group['gender'].iloc[0]
            pair_indices = group.groupby('pair')['pair_start'].mean().reset_index()

            pair_start_dict = {'person_id': person_id}
```

```python
            if 'MALE' == gender:
                pair_start_dict['IS_MALE'] = 1
            else:
                pair_start_dict['IS_MALE'] = 0

            for pair_type in train_pair_type:
                if pair_type in pair_indices['pair'].values:
                    pair_start_dict[pair_type] = round(pair_indices.loc[pair_indices['pair'] == pair_type, 'pair_start'].values[0], 2)
                else:
                    pair_start_dict[pair_type] = 0

            pair_start_df = pd.concat([pair_start_df, pd.DataFrame([pair_start_dict])], ignore_index=True)
            pair_start_df.fillna(0, inplace=True)
        self.pair_start_df = pair_start_df
        return pair_start_df

    # this function can be called to make condition pairs into "avg position part of pairs" columns, showing the in which parts of the medical
    # journey do pairs appear (1: <=25%, 2: >25% & <=50%, 3: >50% & <=75%, 4: >75%)
    def pairs_to_avgpos_part(self):
        if not hasattr(self, 'pairs'):
            raise AttributeError("The 'pairs' attribute has not been generated. Please call 'generate_pairs' method first.")

        df = self.pairs
        train_pair_type = list(self.xgbmodel_avg.get_booster().feature_names)
        pair_avg_part_df = pd.DataFrame(columns=['person_id'] + train_pair_type)

        for person_id, group in df.groupby('person_id'):
            gender = group['gender'].iloc[0]
            pair_indices = group.groupby('pair')['avg_pair_position'].mean().reset_index()
            pair_avg_part_dict = {'person_id': person_id}

            if 'MALE' == gender:
                pair_avg_part_dict['IS_MALE'] = 1
            else:
                pair_avg_part_dict['IS_MALE'] = 0

            # get the starting point of the "latest condition pairs" to assume it as the indicator of the length of a patient's condition journey
            length = group['pair_start'].max()-group['pair_start'].min()
            start = group['pair_start'].min()

            for pair_type in train_pair_type:
                if pair_type in pair_indices['pair'].values:
                    # get the starting position of the condition pairs
                    position = round(pair_indices.loc[pair_indices['pair'] == pair_type, 'avg_pair_position'].values[0], 2)
                    if position <= round(start+(length/4),2):
                        pair_avg_part_dict[pair_type] = 1
                    elif round(length/4,2) < position <= round(start+2*(length/4),2):
                        pair_avg_part_dict[pair_type] = 2
                    elif round(2*length/4,2) < position <= round(start+3*(length/4),2):
                        pair_avg_part_dict[pair_type] = 3
                    else:
                        pair_avg_part_dict[pair_type] = 4
                else:
                    pair_avg_part_dict[pair_type] = 0

            pair_avg_part_df = pd.concat([pair_avg_part_df, pd.DataFrame([pair_avg_part_dict])], ignore_index=True)
            pair_avg_part_df.fillna(0, inplace=True)
```

```python
        self.pair_avg_part_df = pair_avg_part_df
        return pair_avg_part_df

    # this function can be called to make condition pairs into "average position of pairs" columns, showing the average weighted point of pairs
    # those pairs that appear more than one time will be counted to show how many times the patient move back and forth between 2 conditions
    def pairs_to_avgpos(self):
        if not hasattr(self, 'pairs'):
            raise AttributeError("The 'pairs' attribute has not been generated. Please call 'generate_pairs' method first.")

        df = self.pairs
        train_pair_type = list(self.xgbmodel_avg.get_booster().feature_names)
        pair_avg_df = pd.DataFrame(columns=['person_id'] + train_pair_type)

        for person_id, group in df.groupby('person_id'):
            gender = group['gender'].iloc[0]
            pair_indices = group.groupby('pair')['avg_pair_position'].mean().reset_index()

            pair_avg_dict = {'person_id': person_id}

            if 'MALE' == gender:
                pair_avg_dict['IS_MALE'] = 1
            else:
                pair_avg_dict['IS_MALE'] = 0

            for pair_type in train_pair_type:
                if pair_type in pair_indices['pair'].values:
                    pair_avg_dict[pair_type] = round(pair_indices.loc[pair_indices['pair'] == pair_type, 'avg_pair_position'].values[0], 2)
                else:
                    pair_avg_dict[pair_type] = 0

            pair_avg_df = pd.concat([pair_avg_df, pd.DataFrame([pair_avg_dict])], ignore_index=True)
            pair_avg_df.fillna(0, inplace=True)
        self.pair_avg_df = pair_avg_df
        return pair_avg_df

    # this function can be called to make predictions on the given dataframe using the XGBoost model provided
    def c2c_xgb_predict_start(self):

        best = self.xgbmodel_start

        pred = best.predict(self.pair_start_part_df.iloc[:,1:].astype(int))
        predictions = self.label_encoder_start.inverse_transform(pred)

        self.pair_start_part_df['predictions'] = predictions
        self.xgb_start_result_df = self.pair_start_part_df

        return self.pair_start_part_df[['person_id', 'predictions']]

    def c2c_xgb_predict_avg(self):

        best = self.xgbmodel_avg

        pred = best.predict(self.pair_avg_part_df.iloc[:,1:].astype(int))
        predictions = self.label_encoder_avg.inverse_transform(pred)

        self.pair_avg_part_df['predictions'] = predictions
        self.xgb_avg_result_df = self.pair_avg_part_df
```

```
        return self.pair_avg_part_df[['person_id', 'predictions']]


    # this function can be called to make predictions on the given dataframe using the Logistic Regression model provided
    def c2c_logreg_predict_start(self):

        best = self.logregmodel_start

        pred = best.predict(self.pair_start_df.iloc[:,1:].astype(float))

        self.pair_start_df['predictions'] = pred
        self.logreg_start_result_df = self.pair_start_df

        return self.pair_start_df[['person_id','predictions']]

    def c2c_logreg_predict_avg(self):

        best = self.logregmodel_avg

        pred = best.predict(self.pair_avg_df.iloc[:,1:].astype(float))

        self.pair_avg_df['predictions'] = pred
        self.logreg_avg_result_df = self.pair_avg_df

        return self.pair_avg_df[['person_id','predictions']]

    def xgb_prediction_pie_chart(self, ratio_setting=0.01):
        if not hasattr(self, 'xgb_start_result_df') or not hasattr(self, 'xgb_avg_result_df'):
            raise AttributeError("At least one of the 'xgb_start_result_df' and 'xgb_avg_result_df' has not been generated. Please call 'c2c_xgb_predict_start' and 'c2c_xgb_predic

        counts1 = self.xgb_start_result_df['predictions'].value_counts()
        counts2 = self.xgb_avg_result_df['predictions'].value_counts()
        total_count1 = counts1.sum()
        total_count2 = counts2.sum()
        ratios1 = counts1 / total_count1
        ratios2 = counts2 / total_count2
        significant_events1 = ratios1[ratios1 > ratio_setting].index.tolist()
        other_ratio1 = ratios1[ratios1 <= ratio_setting].sum()
        significant_events2 = ratios2[ratios2 > ratio_setting].index.tolist()
        other_ratio2 = ratios2[ratios2 <= ratio_setting].sum()

        plt.pie(ratios1[ratios1 > ratio_setting].values.tolist() + [other_ratio1], labels=significant_events1 + ['others'], autopct='%1.1f%%', startangle=140)
        plt.title('Distribution of XGBoost predictions in the start dataset:')
        plt.show()
        plt.pie(ratios2[ratios2 > ratio_setting].values.tolist() + [other_ratio2], labels=significant_events2 + ['others'], autopct='%1.1f%%', startangle=140)
        plt.title('Distribution of XGBoost predictions in the weighted dataset:')
        plt.show()

    def logreg_prediction_pie_chart(self, ratio_setting=0.01):
        if not hasattr(self, 'logreg_start_result_df') or not hasattr(self, 'logreg_avg_result_df'):
            raise AttributeError("At least one of the 'self.pair_start_df' and 'self.pair_avg_df' has not been generated. Please call 'c2c_logreg_predict_start' and 'c2c_logreg_pr

        counts1 = self.logreg_start_result_df['predictions'].value_counts()
        counts2 = self.logreg_avg_result_df['predictions'].value_counts()
        total_count1 = counts1.sum()
        total_count2 = counts2.sum()
        ratios1 = counts1 / total_count1
        ratios2 = counts2 / total_count2
```

```
    significant_events1 = ratios1[ratios1 > ratio_setting].index.tolist()
    other_ratio1 = ratios1[ratios1 <= ratio_setting].sum()
    significant_events2 = ratios2[ratios2 > ratio_setting].index.tolist()
    other_ratio2 = ratios2[ratios2 <= ratio_setting].sum()

    plt.pie(ratios1[ratios1 > ratio_setting].values.tolist() + [other_ratio1], labels=significant_events1 + ['others'], autopct='%1.1f%%', startangle=140)
    plt.title('Distribution of Logistic Regression predictions in the start dataset:')
    plt.show()
    plt.pie(ratios2[ratios2 > ratio_setting].values.tolist() + [other_ratio2], labels=significant_events2 + ['others'], autopct='%1.1f%%', startangle=140)
    plt.title('Distribution of Logistic Regression predictions in the weighted dataset:')
    plt.show()
```