creating_observations_for_2425

February 9, 2021

[1]:
```
import pandas as pd
import numpy as np

from models.panel import PanelData
from models.episode import EpisodeModel
from feature_extraction.methods import episode_dynamics_dummy
```

[2]:
```
mimic_2425 = pd.read_csv('_data/mimic_2425.csv')
```

[3]:
```
print(mimic_2425.head(3))
# raw time series with minute by minute entries
```

```
            Unnamed: 0    RR   SPO2   MAP   SBP   DBP     HR    PP       CO
0  2020-10-18 15:24:25  35.0   99.9   0.0   0.0   0.0  106.9   0.0     0.00
1  2020-10-18 15:25:25  36.4  100.0  87.0  98.9  63.1  107.3  35.8  3841.34
2  2020-10-18 15:26:25  35.2  100.0  75.2  97.9  63.0  107.5  34.9  3751.75
```

[4]:
```
mimic_2425 = mimic_2425.drop('Unnamed: 0', axis=1)
print(mimic_2425.shape)
```

```
(12877, 8)
```

[5]:
```
data_model = PanelData(prototype=mimic_2425,
                       periods=(60, 60, 30))
# class I created to handle data processing
```

[6]:
```
for col in ['MAP', 'DBP', 'SBP', 'HR']:
    mimic_2425[col] = mimic_2425[col].where(mimic_2425[col].between(10, 200))
```

[7]:
```
target_specs = \
    dict(
        hypotension=dict(above_threshold=False,
                         value_threshold=60,
                         ratio_threshold=0.9,
                         target_variable='MAP'),
        tachycardia=dict(above_threshold=True,
                         value_threshold=100,
                         ratio_threshold=0.9,
```

```
                                target_variable='HR'))
```

[8]:

```
# building the observations
X, y = [], []
for i in np.arange(data_model.episode_len, mimic_2425.shape[0] + 1):
    # a for loop that goes from the episode duration (60min+60min+30min = 150␣
↪time series points)
    # to the length of the episode

    indices = np.arange(i - data_model.episode_len, i)

    # an episode is a 150 window with the obser. period + warning period +␣
↪target period
    episode = mimic_2425.iloc[indices, :]

    # getting features from the episode
    # the method .data_points_predictors automatically retrieves the␣
↪observation period only
    X_i, episode_is_valid = data_model.data_points_predictors(episode,␣
↪predictors_fun=episode_dynamics_dummy)


    if not episode_is_valid:
        X.append(X_i)
        y.append(np.nan)
    else:
        y_i_spot = data_model.spot_threshold_target(episode,␣
↪**target_specs['hypotension'])
        X.append(X_i)
        y.append(y_i_spot)
```

[9]:

```
X = pd.concat(X, axis=1).T
```

[10]:

```
#dummy predictors, the real ones are more extensive
print(X.head())
print(X.shape)
```

```
     RR_mean   SPO2_mean   MAP_mean   SBP_mean   DBP_mean     HR_mean  \
0  14.345000  98.133333  67.820339  88.091071  56.767857  103.928333
1  13.761667  98.118333  67.825000  87.971930  56.840351  103.826667
2  13.155000  98.098333  67.506667  87.666667  56.791228  103.726667
3  12.631667  98.075000  67.348333  87.401754  56.668421  103.615000
4  12.065000  98.056667  67.183333  87.135088  56.542105  103.505000

     PP_mean      CO_mean  RR_SPO2_ccf  RR_MAP_ccf  …  SBP_DBP_ccf  \
0  29.235000  3042.300667      3465.00     2338.00  …     5179.755
1  29.575000  3076.572667      3603.60     2478.84  …     6023.010
```

```
2  29.331667  3048.343000     3477.76     2390.08  …     5903.370
3  29.196667  3030.837833     3352.40     2233.80  …     5443.200
4  29.063333  3013.634167     3451.61     2265.01  …     5308.800

    SBP_HR_ccf  SBP_PP_ccf     SBP_CO_ccf  DBP_HR_ccf  DBP_PP_ccf  \
0  8923.625536  1911.57625  193642.674125  5750.583929   1231.8625
1  9969.120000  2017.56000  203370.048000  6360.480000   1287.2400
2  9917.270000  2075.48000  210246.124000  6381.900000   1335.6000
3  9797.760000  2604.96000  262579.968000  6300.000000   1675.0000
4  9638.400000  2563.20000  257345.280000  6224.800000   1655.4000

     DBP_CO_ccf  HR_PP_ccf   HR_CO_ccf  PP_CO_ccf
0  124787.67125    2319.73  234988.649      0.000
1  129753.79200    2188.92  220643.136  73616.256
2  135296.28000    2279.00  230862.700  74949.844
3  168840.00000    2867.60  289054.080  93739.968
4  166202.16000    2856.90  286832.760  91143.120

[5 rows x 36 columns]
(12728, 36)
```

[11]:
```python
print(len(y))
print(pd.Series(y).value_counts())
```

```
12728
0.0    12580
1.0       19
dtype: int64
```

[12]:
```python
print(pd.Series(y).value_counts() / len(y))
```

```
0.0    0.988372
1.0    0.001493
dtype: float64
```

[13]:
```python
X['target'] = y
# so, we got the RAW observations without any restriction
# we will do it now
```

[14]:
```python
ep_model = EpisodeModel(target_variable='target',
                        min_ep_duration=150,
                        max_event_duration=60,
                        positive_entities_only = True)
# the minimum duration of an episode is 150 data points, which is fine in this␣
↪case
# if an AHE event takes longer than 60 min, we truncate the rest of the␣
↪observations because then predicting anything
```

```
# is useless in practice
```

[15]:
```
# first, we split the entity by activity. e.g. if an entity contains two␣
↪different event, these will be two separate episodes
# besides, we truncate the event duration to 60min (max_event_duration)
X_split = ep_model.episode_split(X,
                                 target_variable=ep_model.target_variable,
                                 min_ep_duration=ep_model.min_ep_duration,
                                 max_event_duration=ep_model.max_event_duration)

X_split_df0 = pd.concat(X_split)

# then, for each sub episode, we sample the data every 30 min␣
↪(sample_interval_size)
# this will basically reduce the dataset, as the distribution of the class␣
↪should remain the same
for k in X_split:
    X_split[k] = \
        ep_model.non_overlapping_resample(episode=X_split[k],
                                          target_variable=ep_model.target_variable,
                                          sample_interval_size=30,
                                          include_class_condition=True)

X_split_df = pd.concat(X_split)
```

[16]:
```
print(X_split_df0.shape)
print(X_split_df0['target'].value_counts() / X_split_df0.shape[0])

print(X_split_df.shape)
print(X_split_df['target'].value_counts() / X_split_df.shape[0])
```

```
(11073, 37)
0.0    0.998555
1.0    0.001355
Name: target, dtype: float64
(385, 37)
0.0    0.961039
1.0    0.038961
Name: target, dtype: float64
```

[17]:
```
print((X_split_df['target'] > 0).sum())
```

```
15
```

[18]:
```
print((X_split_df['target'].isna().sum()))
```

```
0
```

[ ]:

```

```