Supplementary Material

Table S1. Feature inputs for models. Clinical variables which were processed into inputs for model training.

Vitals and labs	Counts	Boolean Indicators	Demographic Info
• Systolic BP • Dias BP • HR • Respiratory Rate • Temperature • Hematocrit • SpO_2 • GCS • Platelet Count • Sputum • Blood culture • WBC • Creatinine	Numbers of MV hours, sputum tests, blood cultures, urine output measurements, and antibiotic input events	Indicators of cirrhosis, congestive heart failure, fever, bacteremia, intracranial hemorrhage, renal failure, respiratory distress, respiratory failure, sepsis, subarachnoid hemorrhage, shortness of breath, and acute respiratory distress syndrome	• Age • Weight • Total urine output

Table S2. Demographic information of patients included in time windows of k = 12, 24, 36 hours. Abbreviations: VAP - Ventilator-associated pneumonia

		***k* = 12**		***k* = 24**		***k* = 36**	
		VAP +	VAP -	VAP +	VAP -	VAP +	VAP -
	Characteristic	n = 470	n = 20594	n = 470	n = 20017	n = 470	n = 19447
Age	<30	26 (5.5%)	899 (4.4%)	26 (5.5%)	848 (4.2%)	26 (5.5%)	776 (4%)
	30-49	78 (16.6%)	3024 (14.7%)	78 (16.6%)	2921 (14.6%)	78 (16.6%)	2811 (14.5%)
	50-59	98 (20.9%)	3725 (18.1%)	98 (20.9%)	3653 (18.2%)	98 (20.9%)	3569 (18.4%)
	60-69	99 (21.1%)	4734 (23%)	99 (21.1%)	4631 (23.1%)	99 (21.1%)	4541 (23.4%)
	70-79	95 (20.2%)	4641 (22.5%)	95 (20.2%)	4527 (22.6%)	95 (20.2%)	4431 (22.8%)
	80+	74 (15.7%)	3571 (17.3)	74 (15.7%)	3427 (17.2%)	74 (15.7%)	3319 (17.1%)
Gender	Male	282 (60%)	12303 (59.7%)	282 (60%)	11969 (59.8%)	282 (60%)	11641 (59.9%)
	Female	188 (40%)	8291 (40.3%)	188 (40%)	8048 (40.2%)	188 (40%)	7806 (40.1%)
Ethnicity	White	314 (66.8%)	14634 (71.1%)	314 (66.8%)	14259 (71.2%)	314 (66.8%)	13865 (71.3%)
	Black/African-American	43 (9.1%)	1500 (7.3%)	43 (9.1%)	1449 (7.2%)	43 (9.1%)	1402 (7.2%)
	Asian	18 (3.8%)	446 (2.2%)	18 (3.8%)	434 (2.2%)	18 (3.8%)	418 (2.1%)
	Hispanic/ Latino	12 (2.6%)	666 (3.2%)	12 (2.6%)	644 (3.2%)	12 (2.6%)	619 (3.2%)
	Unknown/ Other	83 (17.7%)	3348 (16.2%)	83 (17.7%)	3231 (16.1%)	83 (17.7%)	3143 (16.1%)

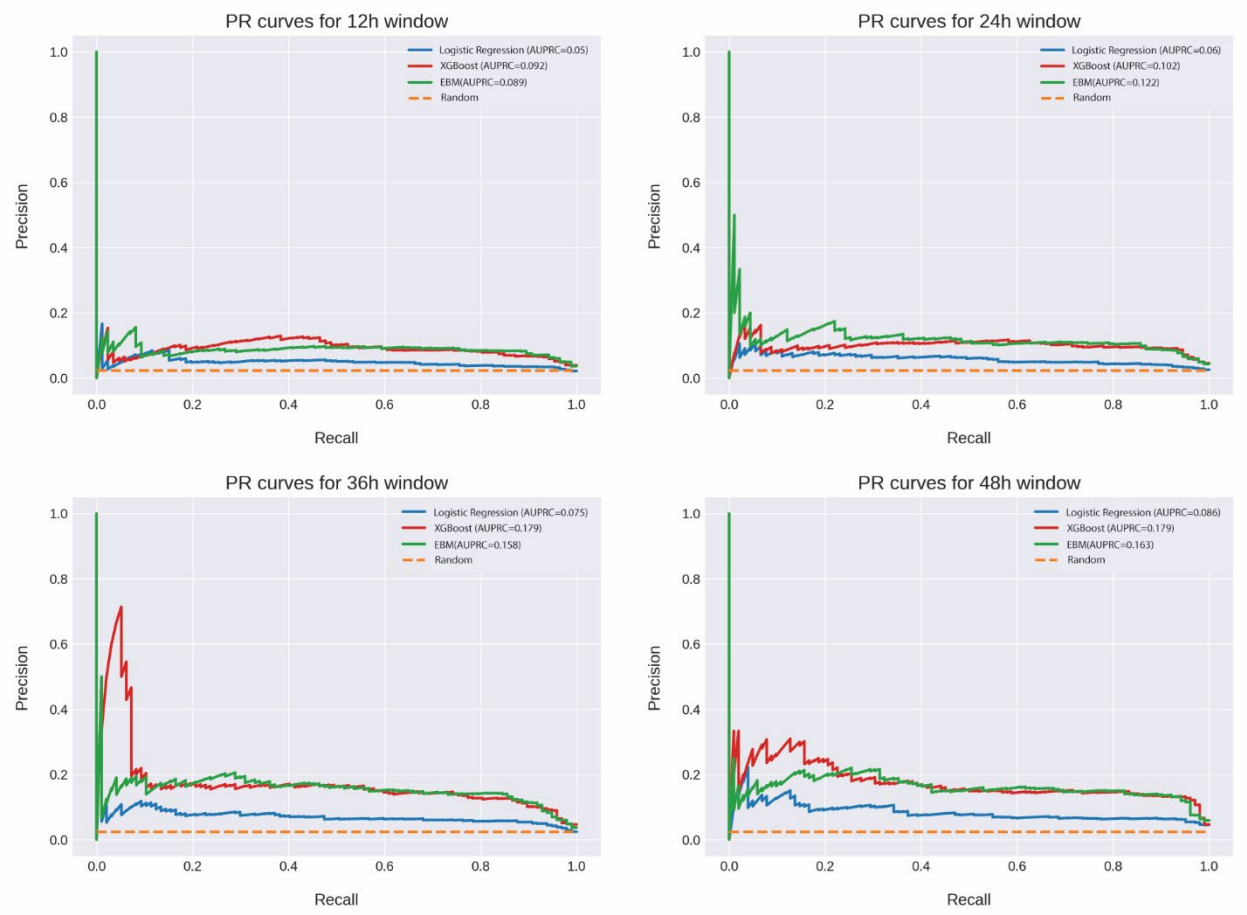

Figure S1. Precision recall (PR) curves for LR, fEBM, and XGB models at k = 12, 24, 36, 48 hours. Abbreviations: LR – Logistic Regression; fEBM – full feature Explainable Boosting Machine; XGB – XGBoost; AUPRC – Area Under the Precision Recall Curve

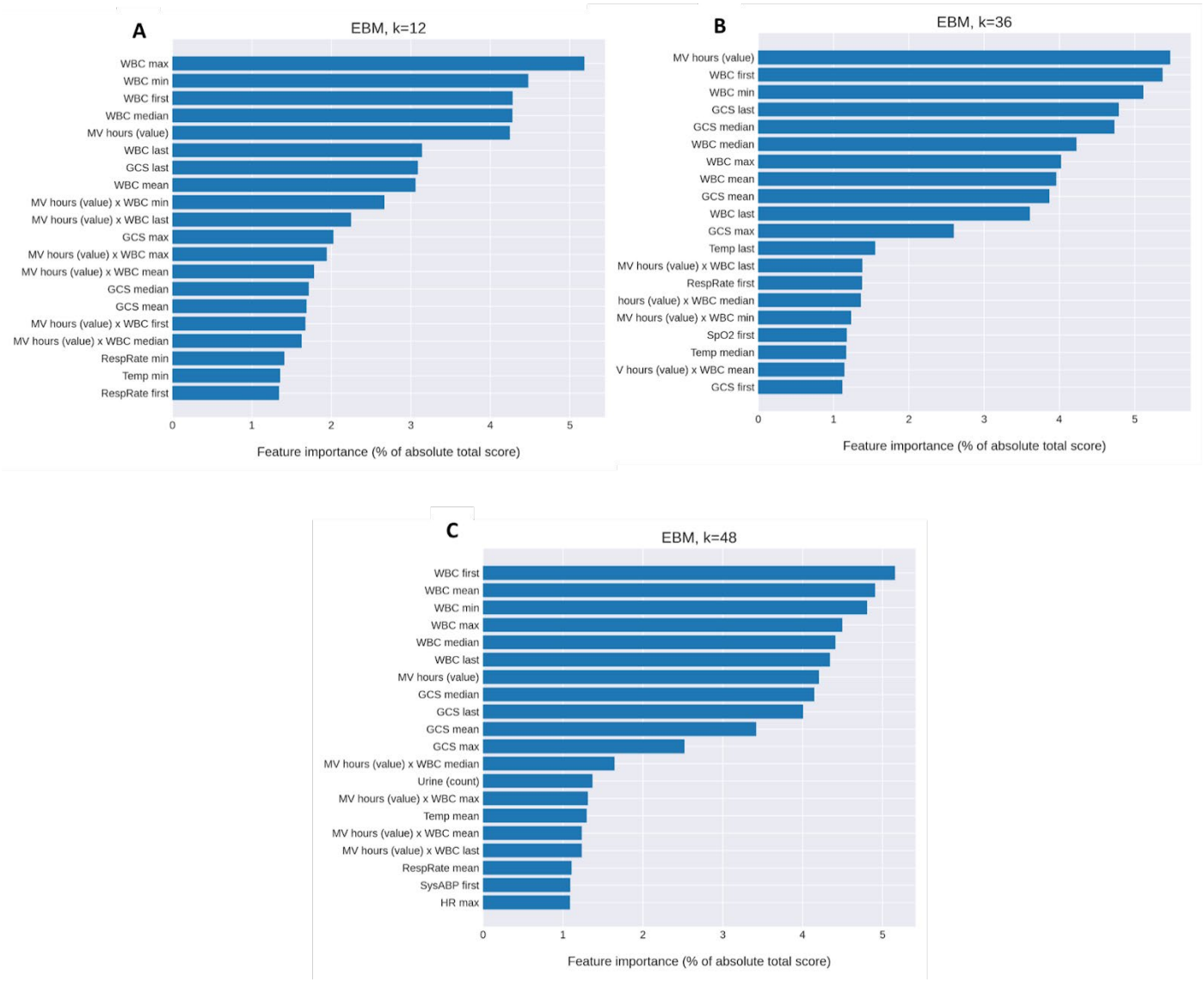

Figure S2. Feature importance plots for the fEBM model at k = 12, 36, 48 h. Abbreviations: WBC - white blood cell; MV - mechanical ventilator; GCS - Glasgow coma scale; HR - heart rate; fEBM – full feature Explainable Boosting Machine.

Figure S3. Contribution of the top features to fEBM scores. There are three panels corresponding to the top features of fEBM—MV hours, WBC first , and GCS last. Each panel consists of two subplots, the top plot shows the contribution to the fEBM risk score versus standardized feature value; the bottom plot indicates how common it is to observe a range of feature values. A higher score reflects greater risk of VAP. The score fEBM

produces is a sum of scores resulting from each feature. Abbreviations: MV- mechanical ventilation; WBC - white blood cell count; and GCS - Glasgow Coma Scale; fEBM – full feature Explainable Boosting Machine.

Code S1. Source code for data processing and model training

```
##############################
# _1_Processing
# ##############################

# In this part we extract the dataset from raw CSV files. It is then saved
as a PatientCollection (PC) as raw_mimic_full.dat
# After various filtering operations, the PC is saved as
filtered_mimic_full.dat. After labeling and processing, the PC is saved as
processed_mimic_full.dat
# These 3 files are only intermediate and are never used later again.

# # Imports

# +
import sys
import collections
import os

import numpy as np
import pandas as pd
import logging
logging.basicConfig(level = logging.INFO)

import utils
import config
import import_csv_all, save_to_file, load_from_file, process_raw_data_all
from tqdm import tqdm, trange
import time
import matplotlib.pyplot as plt
# %matplotlib inline
# -

# # 1 - Extract the CSV files into a PatientCollection

start = time.time()
# Load all CSV files from your source directory
mimic = import_csv_all()
print((time.time() - start)/60, 'minutes.')

### Save the raw PatientCollection
save_to_file(mimic, '~/MIMIC-III/raw_mimic_full.dat') #saves the raw
patient collection
```

```
# # 2 - Filter the data

# ## 2.1 - Basic filtering (age and existence of data)

#In case one wants to load the dataset from just above
mimic = load_from_file('~/MIMIC-III/raw_mimic_full.dat')
print(len(mimic))

mimic['200033']['Info']

mimic['200033']['RawData'].keys()

# ### Filter patients without data

utils.raw_data_filter(mimic)

# ### Filter patients under 18

utils.age_filter(mimic)

# ## 2.2 - VAP-related filtering (no CAP, intubation)

# ### Hardcode some info about MV

# +
### Hardcode some information in the 'Info' section of each patient
### To make the filtering easier afterwards

icustay = pd.read_csv('~/MIMIC-III/ICUSTAYS.csv')
for i in range(len(icustay)):
      ID = str(icustay['ICUSTAY_ID'][i])
      if ID in mimic:
      assert mimic[ID]['Info']['admission_time'] ==
np.datetime64(icustay['INTIME'][i])
      mimic[ID]['Info']['discharge_time'] =
np.datetime64(icustay['OUTTIME'][i])

# +
mv = pd.read_csv('~/MIMIC-III/ventilation_duration.csv') #already ordered
mv =
mv.loc[np.logical_not(np.isnan(mv['icustay_id']))].reset_index(drop=True)
```

```
for key in mimic:
      mimic[key]['Info']['mvstarts']=[]
      mimic[key]['Info']['mvends']=[]

for i in range(len(mv)):
      ID = str(int(mv['icustay_id'][i]))
      if ID in mimic:

mimic[ID]['Info']['mvstarts'].append(np.datetime64(mv['starttime'][i]))
      mimic[ID]['Info']['mvends'].append(np.datetime64(mv['endtime'][i]))


# +
### Check there is no problem in MV times ordering
def is_ordered(l):
      for i in range(len(l)-1):
      if l[i+1]<l[i]:
            return False
      return True

for icustay in mimic:
      if len(mimic[icustay]['Info']['mvstarts'])==0:
      assert len(mimic[icustay]['Info']['mvends'])==0
      else:
      assert
len(mimic[icustay]['Info']['mvstarts'])==len(mimic[icustay]['Info']['mvends
'])
      j = []
      for pair in zip(mimic[icustay]['Info']['mvstarts'],
mimic[icustay]['Info']['mvends']):
            j.extend(pair)
      assert is_ordered(j)
# -

# ### Filter patients who were never intubated

# +
print(len(mimic), 'encounters prior to MV filtering:')

for key in list(mimic.keys()):
      if len(mimic[key]['Info']['mvstarts'])==0:
      mimic.pop(key)
```

```
print(len(mimic), 'encounters remaining after MV filtering')
# -

# ### Filter patients diagnosed with CAP (community acquired pneumonia) at
their admission

icustay = pd.read_csv('~/MIMIC-III/ICUSTAYS.csv')
admissions = pd.read_csv('~/MIMIC-III/ADMISSIONS.csv').merge(icustay.loc[:,
['HADM_ID', 'ICUSTAY_ID']], on='HADM_ID',)
print(admissions.head())

# +
cap_ids = []
for i in range(len(admissions)):
      if 'pneumonia' in str(admissions['DIAGNOSIS'][i]).lower():
      cap_ids += [admissions['ICUSTAY_ID'][i]]
valid_cap_ids = [str(ID) for ID in cap_ids if str(ID) in
list(mimic.keys())]

print(len(mimic), 'encounters prior to CAP filtering')
for key in valid_cap_ids:
      mimic.pop(key)
print(len(mimic), 'encounters remaining after CAP filtering')
print(len(valid_cap_ids), 'encounters removed')
# -

# ### Save the filtered patient collection

save_to_file(mimic, '~/MIMIC-III/filtered_mimic_full.dat')

# # 3 - Label VAP cases

#In case one wants to load the dataset from just above
mimic = load_from_file('~/MIMIC-III/filtered_mimic_full.dat')

# ### Labeling

icustay = pd.read_csv('~/MIMIC-III/ICUSTAYS.csv')
diagnoses_icd =
pd.read_csv('~/MIMIC-III/DIAGNOSES_ICD.csv').merge(icustay.loc[:,
['HADM_ID', 'ICUSTAY_ID']], on='HADM_ID',)
#Not the same as the diagnosis column in the ADMISSIONS tables. This table
corresponds to the diagnoses
```

```
#made at the end of an hospital stay.There may be multiple diagnoses for a
single patient
diagnoses_icd.head()

# +
#Retrieve the ICU ID's for patients with a VAP diagnosis
vap_code = '99731' #ICD-9 code for VAP
vap_filter = diagnoses_icd.query("ICD9_CODE == @vap_code").loc[:,
['ICD9_CODE', 'ICUSTAY_ID']]
vap_ids = set(vap_filter['ICUSTAY_ID'])
valid_vap_ids = [str(ID) for ID in vap_ids if str(ID) in
list(mimic.keys())]


print('Number of stays:', len(mimic))
print('Number of VAP cases: {}'.format(len(valid_vap_ids)),
      '\nPrevalence:',
'{}%'.format(round(100*len(valid_vap_ids)/len(mimic), 2)))

# +
### Assign a pneumonia label to each patient
for patient in list(mimic.keys()):
      mimic[patient]['Outcomes'] = {'Pneumonia': {'is_pneumonia': False}}

for ID in valid_vap_ids:
      mimic[ID]['Outcomes']['Pneumonia']['is_pneumonia'] = True
# -

mimic = process_raw_data_all(mimic) #bins the raw data, dataset size stays
the same

# ### Save the processed patient collection

save_to_file(mimic, '~/MIMIC-III/processed_mimic_full.dat')

# # 4 - Demo

#In case one wants to load the dataset from just above
mimic = load_from_file('~/MIMIC-III/processed_mimic_full.dat')

# +
print(len(mimic))
```

```
for ID in list(mimic.keys()):
      if 'HR' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'DiasABP' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'SysABP' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'Temp' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'SpO2' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'WBC' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'Creatinine' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'RespRate' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'GCS' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'Platelets' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'BUN' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)
      elif 'Hematocrit' not in mimic[ID]['ProcessedData']:
      mimic.pop(ID)

print(len(mimic)) #should be 6126 with everything ?

# +
### Keys are patient ICUSTAY id's, between 200,001 and 299,999 (strings)
#keys = list(mimic.keys())
#keys.remove('')
#keys = list(map(int, keys))
#print(min(keys), max(keys))

patient = mimic['222222'] #a random patient

#one can recognize the keys defined earlier in the CONFIG
print('Keys of a patient\'s PatientCollection:', list(patient.keys()))
print('--> ID:', patient['ID'])
print('--> ICD9:', patient['ICD9'])
print('--> Info:', patient['Info'], '\n')
print('ProcessedData keys:', list(patient['ProcessedData'].keys()), '\n')
```

```
#processed data contains time series of the patient's measurements
print('Keys of every measurement in ProcessedData:',
list(patient['ProcessedData']['HR'].keys()), '\n')

print('For example, the heart rate (HR)')
print('Time: \n',
      patient['ProcessedData']['HR']['time'],
      '\n Value: \n',
      patient['ProcessedData']['HR']['value'])


##############################
# _2_Processing_contd.
# ##############################

# In this part we go from processed_mimic_full.dat to processed_mimic_k.dat
with k = 12, 24, 36, 48
# The resulting files are our final patient population data for each k.
However it requires a bit more work to get this data ready for learning.

# # Imports

# +
import sys
import utils
import collections
import os

import save_to_file
import numpy as np
import pandas as pd
import logging
logging.basicConfig(level = logging.INFO)

from tqdm import tqdm, trange
import time
import matplotlib.pyplot as plt
# %matplotlib inline
# -

mimic = load_from_file('~/MIMIC-III/processed_mimic_full.dat')
```

```
# # Independence

icustay = pd.read_csv('~/MIMIC-III/ICUSTAYS.csv')
mv = pd.read_csv('~/MIMIC-III/ventilation_duration.csv')
mv =
mv.loc[np.logical_not(np.isnan(mv['icustay_id']))].reset_index(drop=True)

mv.head()

# +
### Just to illustrate

ids = [int(ID) for ID in mimic] #ICUSTAY_ID's in our dataset

#corresponding hospital admission id for each ICU stay in our dataset
hadm_ids =
icustay.loc[icustay['ICUSTAY_ID'].isin(ids)]['HADM_ID'].to_numpy() #may
contain repetitions

print('Number of unique hospital admissions represented in the dataset:',
len(set(hadm_ids)), '/', len(ids), 'ICU stays')

# +
### Here we remove all the ICU stays (with MV) such that the patient was
admitted to the ICU and ventilated
### another time during the same hospital stay. Otherwise there is no way
to know which instance of ventilation led to VAP.
### (diagnoses are labeled with the hospital admission ID)

#ICU STAY id's with MV
mv_icu_ids = mv['icustay_id'].to_numpy()

#HADM ID's with ICU STAY with MV (may contain repetitions !)
mv_hadm_ids =
icustay.loc[icustay['ICUSTAY_ID'].isin(mv_icu_ids)]['HADM_ID'].to_numpy()

#HADM ID's with multiple ICU stays with MV
multiplemv_hadm_ids = []
counter = dict(collections.Counter(mv_hadm_ids))
for hadm_id in counter:
      if counter[hadm_id]>1:
      multiplemv_hadm_ids.append(hadm_id)
      else:
```

```
      pass

#ICUSTAY ID's corresponding to HADM ID's with multiple MV ICU stays
bad_icustay_ids =
icustay.loc[icustay['HADM_ID'].isin(multiplemv_hadm_ids)]['ICUSTAY_ID'].to_
numpy()

#The same but only those contained in our dataset
valid_bad_icustay_ids = [str(ID) for ID in bad_icustay_ids if str(ID) in
mimic]

print(len(mimic), 'encounters prior to multiple MV filtering')
for icustay in list(mimic.keys()):
      if icustay in valid_bad_icustay_ids:
      mimic.pop(icustay)
print(len(mimic), 'encounters remaining after multiple MV filtering')
print(len(valid_bad_icustay_ids), 'encounters removed')
# -

count = 0
for icustay in mimic:
      if mimic[icustay]['Outcomes']['Pneumonia']['is_pneumonia'] == True:
count+=1
print('Number of VAP cases: {}'.format(count),
      '\nPrevalence:', '{}%'.format(round(100*count/len(mimic), 2)))

# ### Filtering altenatives

# # Time filtering

### Check everything went as planned during processing
# All times are the same
for ID in mimic:
      t = [mimic[ID]['ProcessedData'][key]['time'] for key in
mimic[ID]['ProcessedData']]
      assert (t==t[0]).all()
# Times are hourly spaced
for ID in mimic:
      t =
mimic[ID]['ProcessedData'][list(mimic[ID]['ProcessedData'].keys())[0]]['tim
e']
      assert np.all((t[1:]-t[:-1])==np.timedelta64(1, 'h'))
```

```
### Adapts the way these are computed (with ProcessedData times instead of
RawData times)
# (For better clarity moving on)
for ID in mimic:
      mimic[ID]['Info']['first_measurement_time'] =
mimic[ID]['ProcessedData'][list(mimic[ID]['ProcessedData'].keys())[0]]['tim
e'][0]
      mimic[ID]['Info']['last_measurement_time'] =
mimic[ID]['ProcessedData'][list(mimic[ID]['ProcessedData'].keys())[0]]['tim
e'][-1]

# +
### VAP by definition can only happen after 48h so we want to have 48h of
data at least
### (or more precisely data in 48 different hourly bins after intubation)
### starting in the first hour following intubation.

### We keep patients that have at least 12/24/36/48h of data. Set k here.
k = 48

print(len(mimic), 'encounters before time filtering')
for ID in list(mimic.keys()):
      if
mimic[ID]['Info']['first_measurement_time']-mimic[ID]['Info']['mvstarts'][0
] > np.timedelta64(1, 'h'):
      mimic.pop(ID)
      elif
mimic[ID]['Info']['last_measurement_time']-max(mimic[ID]['Info']['mvstarts'
][0], mimic[ID]['Info']['first_measurement_time'])  < np.timedelta64(k-1,
'h'):
      mimic.pop(ID)
print(len(mimic), 'encounters remaining after time filtering')

# +
count=0
for ID in mimic:
      if
mimic[ID]['Info']['discharge_time']-mimic[ID]['Info']['mvstarts'][0]<np.tim
edelta64(48, 'h'):
      count+=1
print(count)
# -
```

```
# # Prevalence check

count = 0
for icustay in mimic:
      if mimic[icustay]['Outcomes']['Pneumonia']['is_pneumonia'] == True:
count+=1
print('Number of VAP cases: {}'.format(count),
      '\nPrevalence:', '{}%'.format(round(100*count/len(mimic), 2)))
# We lost 20 positive examples no matter k: there are 20 VAP positive
patients with less than 12h of data.

# # Save

save_to_file(mimic, '~/MIMIC-III/processed_mimic_{}.dat'.format(k))


##############################
# _3_Demographics
# ##############################

import sys
import utils
import collections
import os

import numpy as np
import pandas as pd
import logging
logging.basicConfig(level = logging.INFO)

import load_from_file
from tqdm import tqdm, trange
import time
import matplotlib.pyplot as plt
# %matplotlib inline
# -

mimic = load_from_file('~/MIMIC-III/processed_mimic_36.dat')

ids = list(mimic.keys())
pos_ids = [ID for ID in ids if
mimic[ID]['Outcomes']['Pneumonia']['is_pneumonia']==True]
neg_ids = [ID for ID in ids if
```

```
mimic[ID]['Outcomes']['Pneumonia']['is_pneumonia']==False]
print('Positive cases: {}, Negative cases: {}'.format(len(pos_ids),
len(neg_ids)))

# # ARDS

ards = pd.read_csv('~/MIMIC/mpwr_ards.csv')

ards = ards.loc[ards['icustay_id'].isin(list(map(int, ids)))]
ards = ards.loc[ards['ards']==1, :].reset_index(drop=True)
ards.head(10)

# +
print(len(set(ards['icustay_id']).intersection(set(map(int, pos_ids)))),
      len(set(ards['icustay_id']).intersection(set(map(int,
pos_ids))))/len(pos_ids),
      len(set(ards['icustay_id']).intersection(set(map(int, neg_ids)))),
      len(set(ards['icustay_id']).intersection(set(map(int,
neg_ids))))/len(neg_ids))

print(len(pos_ids)-len(set(ards['icustay_id']).intersection(set(map(int,
pos_ids)))),
      1-len(set(ards['icustay_id']).intersection(set(map(int,
pos_ids))))/len(pos_ids),
      len(neg_ids)-len(set(ards['icustay_id']).intersection(set(map(int,
neg_ids)))),
      1-len(set(ards['icustay_id']).intersection(set(map(int,
neg_ids))))/len(neg_ids))
# -

# # Age

# +
ages = [mimic[ID]['Info']['age'] for ID in neg_ids]
bins = [30, 50, 60, 70, 80]

c = np.bincount(np.digitize(ages, bins = bins))
print(c)
print(np.round(c/len(neg_ids), 3))
# -

# # Gender
```

```
genders = np.array([mimic[ID]['Info']['gender'] for ID in neg_ids])
c = np.sum(genders=='F')
print(c)
print(c/len(neg_ids))

# # Ethnicity

# +
icustay = pd.read_csv('~/MIMIC/ICUSTAYS.csv')

admissions = pd.read_csv('~/MIMIC/ADMISSIONS.csv').merge(icustay.loc[:,
['HADM_ID', 'ICUSTAY_ID']], on = 'HADM_ID')
admissions = admissions.loc[admissions['ICUSTAY_ID'].isin(list(map(int,
ids)))]
assert len(admissions)==len(set(admissions['ICUSTAY_ID']))
# -

eth = admissions.loc[admissions['ICUSTAY_ID'].isin(list(map(int,
pos_ids)))]['ETHNICITY'].to_numpy()
collections.Counter(eth).most_common()

##############################
# _4_Features
# ##############################

# In this part we retrieve and save all the additional features we want
that are not present in the original patient collection
(processed_mimic.dat, which only has basic info (age, weight) and vitals
time series).
# These features include the comorbidities and various indicators and
counts, that are here split in 48 hourly bins, with the reference time
being the initiation of mechanical ventilation (MV).

# # Imports

# +
import sys
import utils
import collections
import os

import load_from_file, save_to_file
import numpy as np
```

```
import pandas as pd
import logging
logging.basicConfig(level = logging.INFO)

from tqdm import tqdm, trange
import time
import matplotlib.pyplot as plt
# %matplotlib inline
# -

# We load the processed_data
#it's just to have the id's
#for thinning experiments, load for k=24 as we want the experiments on this
window.
mimic = load_from_file('~/MIMIC-III/processed_mimic_24.dat')

ids = list(mimic.keys())
pos_ids = [ID for ID in ids if
mimic[ID]['Outcomes']['Pneumonia']['is_pneumonia']==True]
neg_ids = [ID for ID in ids if
mimic[ID]['Outcomes']['Pneumonia']['is_pneumonia']==False]

# +
### Ignore

for key in mimic:
      if not
mimic[key]['Info']['intime'].astype('datetime64[h]')-np.timedelta64(1,
'h')<=mimic[key]['Info']['mvstarts'][0]:
      print(key,
            mimic[key]['Info']['intime'].astype('datetime64[h]'),
            mimic[key]['Info']['mvstarts'][0],
            min(mimic[key]['ProcessedData']['HR']['time']))

# -

# # Comorbidities

# +
icustay = pd.read_csv('~/MIMIC-III/ICUSTAYS.csv')

admissions = pd.read_csv('~/MIMIC-III/ADMISSIONS.csv').merge(icustay.loc[:,
['HADM_ID', 'ICUSTAY_ID']], on='HADM_ID',)
```

```
admissions = admissions.loc[admissions['ICUSTAY_ID'].isin(list(map(int,
ids)))]

comorbidities = ['INTRACRANIAL HEMORRHAGE', 'SUBARACHNOID HEMORRHAGE',
                 'SEPSIS', 'RESPIRATORY FAILURE', 'CONGESTIVE HEART
FAILURE',
                 'FEVER', 'RESPIRATORY DISTRESS', 'SHORTNESS OF BREATH',
                 'CIRRHOSIS', 'RENAL FAILURE', 'CHRONIC RENAL FAILURE',
'BACTEREMIA']

# +
comor_dict = {ID:{comor:False for comor in comorbidities} for ID in ids}

diag_dict = pd.Series(admissions.DIAGNOSIS.values,
index=admissions.ICUSTAY_ID.astype(str)).to_dict()

for icustay in diag_dict:
      for comorbidity in comorbidities:
      if comorbidity in diag_dict[icustay]:
                  comor_dict[icustay][comorbidity]=True
# -

save_to_file(comor_dict, '~/MIMIC-III/_comor.dat')

comor_dict['203909']

# +
pos_count = {comor:0 for comor in comorbidities}
neg_count = {comor:0 for comor in comorbidities}

for ID in pos_ids:
      for comor in comorbidities:
      if comor_dict[ID][comor]==True: pos_count[comor]+=1
for ID in neg_ids:
      for comor in comorbidities:
      if comor_dict[ID][comor]==True: neg_count[comor]+=1

u=max([len(comor) for comor in comorbidities])
print('Comorbidity', (u-11)*' ', '  +  ', '  -  ')
for comor in comorbidities:
      print(comor, (u-len(comor))*' ', round(pos_count[comor]/len(pos_ids),
3), round(neg_count[comor]/len(neg_ids), 3))
# -
```

```
# # Sputum (count)

# +
icustay = pd.read_csv('~/MIMIC-III/ICUSTAYS.csv')
microbio =
pd.read_csv('~/MIMIC-III/MICROBIOLOGYEVENTS.csv').merge(icustay.loc[:,
['HADM_ID', 'ICUSTAY_ID', 'INTIME', 'OUTTIME']], on='HADM_ID',)

sputum = microbio.loc[microbio['SPEC_TYPE_DESC'] == 'SPUTUM', :]
sputum = sputum.loc[sputum['ICUSTAY_ID'].isin(list(map(int, ids)))]
print(len(sputum))
sputum = sputum.loc[sputum['CHARTDATE']!=sputum['CHARTTIME'], :]
sputum =
sputum.loc[np.logical_not(np.isnan(pd.to_datetime(sputum['CHARTTIME']).to_n
umpy()))].reset_index(drop=True) #we need the true times !
print(len(sputum))
sputum['CHARTTIME'] = sputum['CHARTTIME'].astype(np.datetime64)
sputum.head(20)
# -

#drop some of the entries for thinning experiments
sputum = sputum.drop(np.random.choice(np.arange(0, len(sputum)),
3*len(sputum)//4, replace=False)).reset_index(drop=True)
print(len(sputum))

sputum_count = {ID:np.zeros(48, int) for ID in ids}
for i in range(len(sputum)):
      ID = str(sputum['ICUSTAY_ID'][i])
      time = sputum['CHARTTIME'][i]
      start = mimic[ID]['Info']['mvstarts'][0]
      index = int((time - start) / np.timedelta64(1, 'h'))
      if index <= 0: index = 0 #VERY IMPORTANT: we choose to consider
things that happened before intubation
      if 0 <= index <= 47:
      sputum_count[ID][index] +=1

save_to_file(sputum_count, '~/MIMIC-III/_sputum_75.dat')

plt.hist([np.sum(sputum_count[ID]) for ID in neg_ids], bins=40,
density=False, alpha=0.5, color='blue', label='Negative')
plt.hist([np.sum(sputum_count[ID]) for ID in pos_ids], bins=40,
density=False, alpha=0.5, color='red', label='Positive')
```

```
plt.legend()
plt.xlim(0, 47)
plt.yscale('log')
plt.show()
print(np.sum([np.sum(sputum_count[ID])>5 for ID in pos_ids]))
print(np.sum([np.sum(sputum_count[ID])>5 for ID in neg_ids]))

# # Blood (count)

blood = microbio.loc[microbio['SPEC_TYPE_DESC'] == 'BLOOD CULTURE', :]
#microbio loaded from above cell and jointed on ICUSTAYS
blood = blood.loc[blood['ICUSTAY_ID'].isin(list(map(int, ids)))]
print(len(blood))
blood = blood.loc[blood['CHARTDATE']!=blood['CHARTTIME'], :]
blood =
blood.loc[np.logical_not(np.isnan(pd.to_datetime(blood['CHARTTIME']).to_num
py()))].reset_index(drop=True) #we need the true times !
print(len(blood))
blood['CHARTTIME'] = blood['CHARTTIME'].astype(np.datetime64)
blood.head(20)

#drop some of the entries for thinning experiments
blood = blood.drop(np.random.choice(np.arange(0, len(blood)),
3*len(blood)//4, replace=False)).reset_index(drop=True)
print(len(blood))

blood_count = {ID:np.zeros(48, int) for ID in ids}
for i in range(len(blood)):
      ID = str(blood['ICUSTAY_ID'][i])
      time = blood['CHARTTIME'][i]
      start = mimic[ID]['Info']['mvstarts'][0]
      index = int((time - start) / np.timedelta64(1, 'h'))
      if index <= 0: index = 0 #VERY IMPORTANT: we choose to consider
things that happened before intubation
      if 0 <= index <= 47:
      blood_count[ID][index] +=1

save_to_file(blood_count, '~/MIMIC-III/_blood_75.dat')

plt.hist([np.sum(blood_count[ID]) for ID in neg_ids], bins=40,
density=False, alpha=0.5, color='blue', label='Negative')
plt.hist([np.sum(blood_count[ID]) for ID in pos_ids], bins=40,
density=False, alpha=0.5, color='red', label='Positive')
```

```
plt.legend()
plt.xlim(0, 55)
plt.yscale('log')
plt.show()
print(np.sum([np.sum(blood_count[ID])<5 for ID in pos_ids]))
print(np.sum([np.sum(blood_count[ID])<5 for ID in neg_ids]))

# # Antibiotics (count)

antibio =
microbio.loc[np.logical_not(np.isnan(microbio['AB_ITEMID'].to_numpy()))]
antibio = antibio.loc[antibio['ICUSTAY_ID'].isin(list(map(int, ids)))]
print(len(antibio))
antibio = antibio.loc[antibio['CHARTDATE']!=antibio['CHARTTIME'], :]
antibio =
antibio.loc[np.logical_not(np.isnan(pd.to_datetime(antibio['CHARTTIME']).to
_numpy()))].reset_index(drop=True) #we need the true times !
print(len(antibio))
antibio['CHARTTIME'] = antibio['CHARTTIME'].astype(np.datetime64)
antibio.head(20)

#drop some of the entries for thinning experiments
antibio = antibio.drop(np.random.choice(np.arange(0, len(antibio)),
3*len(antibio)//4, replace=False)).reset_index(drop=True)
print(len(antibio))

ab_count = {ID:np.zeros(48, int) for ID in ids}
for i in range(len(antibio)):
      ID = str(antibio['ICUSTAY_ID'][i])
      time = antibio['CHARTTIME'][i]
      start = mimic[ID]['Info']['mvstarts'][0]
      index = int((time - start) / np.timedelta64(1, 'h'))
      if index <= 0: index = 0 #VERY IMPORTANT: we choose to consider
things that happened before intubation
      if 0 <= index <= 47:
      ab_count[ID][index] +=1

save_to_file(ab_count, '~/MIMIC-III/_antibio_75.dat')

plt.hist([np.sum(ab_count[ID]) for ID in neg_ids], bins=40, density=False,
alpha=0.5, color='blue', label='Negative')
plt.hist([np.sum(ab_count[ID]) for ID in pos_ids], bins=40, density=False,
alpha=0.5, color='red', label='Positive')
```

```
plt.legend()
plt.xlim(0, 50)
plt.yscale('log')
plt.show()
# print(np.sum([np.sum(ab_count[ID])<5 for ID in pos_ids]))
# print(np.sum([np.sum(ab_count[ID])<5 for ID in neg_ids]))

# # Urine (count, value)

# +
urine_codes = ['40055', '43175', '40069',
            '40094', '40715', '40473',
            '40085', '40057', '40056',
            '40405', '40428', '40086',
            '40096', '40651',
            '226559', '226560', '226561',
            '226584', '226563', '226564',
            '226565', '226567', '226557',
            '226558', '227488', '227489']

items = pd.read_csv('~/MIMIC-III/D_ITEMS.csv')
items.LABEL = items.LABEL.fillna('')
#items = items.loc[items['LABEL'].str.contains('Urine')]
items = items.loc[items['ITEMID'].isin(list(map(int,urine_codes)))]
items.head(40)
# -

output = pd.read_csv('~/MIMIC-III/OUTPUTEVENTS.csv')

urine = output.loc[output['ITEMID'].isin(map(int, urine_codes))]
urine = urine.loc[urine['ICUSTAY_ID'].isin(list(map(int, ids)))]
print(len(urine))
urine = urine.loc[urine['VALUE']>=0, :]
urine = urine.loc[urine['VALUE']<=10000, :]
urine = urine.loc[urine['VALUEUOM'].isin(['ml', 'mL']), :]
urine =
urine.loc[np.logical_not(np.isnan(pd.to_datetime(urine['CHARTTIME']).to_num
py()))] #we need the true times !
urine =
urine.loc[np.logical_not(np.isnan(urine['VALUE'].to_numpy()))].reset_index(
drop=True)
print(len(urine))
urine['CHARTTIME'] = urine['CHARTTIME'].astype(np.datetime64)
```

```
urine.head(20)

#drop half the entries for thinning experiments
urine = urine.drop(np.random.choice(np.arange(0, len(urine)),
3*len(urine)//4, replace=False)).reset_index(drop=True)
print(len(urine))

urine_value = {ID:np.zeros(48, int) for ID in ids}
urine_count = {ID:np.zeros(48, int) for ID in ids}
for i in range(len(urine)):
      ID = str(int(urine['ICUSTAY_ID'][i]))
      time = urine['CHARTTIME'][i]
      start = mimic[ID]['Info']['mvstarts'][0]
      index = int((time - start) / np.timedelta64(1, 'h'))
      if index <= 0: index = 0 #VERY IMPORTANT: we choose to consider
things that happened before intubation
      if 0 <= index <= 47:
      urine_value[ID][index] += urine['VALUE'][i]
      urine_count[ID][index] += 1

save_to_file(urine_value, '~/MIMIC-III/_urine_value_75.dat')
save_to_file(urine_count, '~/MIMIC-III/_urine_count_75.dat')

plt.hist([np.sum(urine_value[ID]) for ID in neg_ids], bins=40,
density=False, alpha=0.5, color='blue', label='Negative')
plt.hist([np.sum(urine_value[ID]) for ID in pos_ids], bins=40,
density=False, alpha=0.5, color='red', label='Positive')
plt.legend()
plt.xlim(0, 40000)
plt.yscale('log')
plt.show()

plt.hist([np.sum(urine_count[ID]) for ID in neg_ids], bins=40,
density=False, alpha=0.5, color='blue', label='Negative')
plt.hist([np.sum(urine_count[ID]) for ID in pos_ids], bins=40,
density=False, alpha=0.5, color='red', label='Positive')
plt.legend()
plt.xlim(0, 200)
plt.yscale('log')
plt.show()


# # MV Hours (value)
```

```
def MV(mvstarts, mvends):
      '''
      From a two ordered lists of np.datetime64 representing the start and
end of consecutive events with no overlap,
      extracts the total duration of events for each hour in a 48h window
starting at the first event start,
      '''
      mv_hours = np.zeros(48)
      for i in range(len(mvstarts)):
      start = (mvstarts[i]-mvstarts[0]) / np.timedelta64(1, 'h')
      end = (mvends[i]-mvstarts[0]) / np.timedelta64(1, 'h')

      if start >= 48:
            break

      if end > 48:
            end = 48

      if int(end)-int(start)==0:
            mv_hours[int(start)] += end-start

      else:
            mv_hours[int(start)] += int(start)+1-start
            for h in range(int(start)+1, int(end)):
                  mv_hours[h] += 1
            if int(end) != 48:
                  mv_hours[int(end)] += end-int(end)
      return mv_hours


mv_hours = {}
for ID in ids:
      mv_hours[ID] = MV(mimic[ID]['Info']['mvstarts'],
mimic[ID]['Info']['mvends'])

save_to_file(mv_hours, '~/MIMIC-III/_MV.dat')

print(len(ids), len(pos_ids))
print(np.sum([np.sum(mv_hours[ID])==48 for ID in pos_ids]))

plt.hist([np.sum(mv_hours[ID]) for ID in neg_ids], bins=48, density=False,
alpha=0.5, color='blue', label='Negative')
```

```
plt.hist([np.sum(mv_hours[ID]) for ID in pos_ids], bins=48, density=False,
alpha=0.5, color='red', label='Positive')
plt.legend()
plt.xlim(0, 48)
plt.yscale('log')
plt.show()
#We can already see that this will be of upmost importance to differentiate
between positive and negative patients.

# # ARDS (indicator)

ards = pd.read_csv('~/MIMIC-III/mpwr_ards.csv')

ards = ards.loc[ards['icustay_id'].isin(list(map(int, ids)))]
ards =
ards.loc[np.logical_not(np.isnan(pd.to_datetime(ards['charttime']).to_numpy
()))]
ards = ards.loc[ards['ards']==1, :].reset_index(drop=True)
ards['charttime'] = ards['charttime'].astype(np.datetime64)
ards.head(10)

#drop half the entries for thinning experiments
ards = ards.drop(np.random.choice(np.arange(0, len(ards)), 3*len(ards)//4,
replace=False)).reset_index(drop=True)
print(len(ards))

# +
ards_indic = {ID: np.zeros(48, int) for ID in ids}

for i in range(len(ards)):
      ID = str(ards['icustay_id'][i])
      start = mimic[ID]['Info']['mvstarts'][0]
      time = ards['charttime'][i]
      index = int((time - start) / np.timedelta64(1, 'h'))
      if index < 0: index=0
      if index <= 47:
      ards_indic[ID][index:] = 1 #this means that once ards is diagnosed,
necessarily the indicator stays at 1 in the future
# -

save_to_file(ards_indic, '~/MIMIC-III/_ards_75.dat')

print(np.mean([np.any(ards_indic[ID]) for ID in neg_ids]),
```

```
      np.mean([np.any(ards_indic[ID]) for ID in pos_ids]))

#the biggest the sum, the earliest it was diagnosed
plt.hist([np.sum(ards_indic[ID]) for ID in neg_ids], bins=48,
density=False, alpha=0.5, color='blue', label='Negative')
plt.hist([np.sum(ards_indic[ID]) for ID in pos_ids], bins=48,
density=False, alpha=0.5, color='red', label='Positive')
plt.legend()
plt.xlim(0, 48)
plt.yscale('log')
plt.show()
#ARDS is not very indicative of VAP, except maybe when developped early

# # LOS (not a feature!!!)

LOS = {ID: mimic[ID]['Info']['length_of_stay'] for ID in ids}

save_to_file(LOS, '~/MIMIC-III/_LOS.dat')

plt.hist([LOS[ID] for ID in neg_ids], bins=20, density=False, alpha=0.5,
color='blue', label='Negative')
plt.hist([LOS[ID] for ID in pos_ids], bins=20, density=False, alpha=0.5,
color='red', label='Positive')
plt.legend()
plt.xlim(0, 100)
plt.yscale('log')
plt.show()


##############################
# _5_Learning
# ##############################

# In this part we train all the models necessary for the final results,
using the final datasets.

# +
import sys
import utils
import collections
import os

import numpy as np
```

```
import pandas as pd
import logging
logging.basicConfig(level = logging.WARNING)

from tqdm import tqdm, trange
from time import time
import matplotlib.pyplot as plt
# %matplotlib inline
plt.style.use('seaborn-darkgrid')

import load_from_file

import sklearn
from sklearn.linear_model import LogisticRegression, SGDClassifier

from sklearn.model_selection import train_test_split, GridSearchCV,
cross_val_score, ParameterGrid
from sklearn.impute import SimpleImputer
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler# , Imputer
from sklearn.calibration import calibration_curve, CalibratedClassifierCV
from sklearn import svm
import pickle

import keras
import xgboost
from xgboost.sklearn import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score, mean_squared_error,
classification_report, plot_roc_curve, precision_recall_curve,
average_precision_score, brier_score_loss

from bisect import bisect

from interpret import show
from interpret.data import ClassHistogram
from interpret.glassbox import ExplainableBoostingClassifier

from interpret import set_visualize_provider
from interpret.provider import InlineProvider
set_visualize_provider(InlineProvider())

from interpret.perf import ROC
```

```
import tensorflow as tf
import tensorflow.keras.layers as layers
import tensorflow.keras.models as models
import tensorflow.keras.optimizers as optimizers

# +
SEED = 5299 # a different seed leads to different train/test splits.
Results can vary slightly.
logging.disable(logging.WARNING)

def split(X, Y, seed=SEED):
      '''
      X: 2D float array of training examples with dimensions (n_examples,
n_features)
      Y: 1D binary array of labels with dimension n_examples
      seed: int used for the split. With the same seed, datasets of same
size will be splitted the same

      outputs: X_train, X_test, Y_train, Y_test

      Splits a dataset into training and testing array,
      with median imputation for nans and 0-1 normalization
      (both based on the training set).
      '''
      #Split
      X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size =
0.2, shuffle = True, random_state=seed)

      #Input missing values
      impute_median = SimpleImputer(missing_values = np.nan, strategy =
'median')
      impute_median.fit(X_train)

      X_train_imputed = impute_median.transform(X_train)
      X_test_imputed = impute_median.transform(X_test)

      X_train = np.array(X_train_imputed)
      X_test = np.array(X_test_imputed)

      #0-1 normalization
      train_min, train_max = np.min(X_train, axis=0), np.max(X_train,
axis=0)
```

```
      X_train = (X_train-train_min)/(train_max-train_min)
      X_test = (X_test-train_min)/(train_max-train_min)

#     m, sd = np.mean(X_train, axis=0), np.std(X_train, axis=0)
#     X_train = (X_train-m)/sd
#     X_test = (X_test-m)/sd

      return X_train, X_test, Y_train, Y_test

def brier_skill_score(Y_test, y_pred):
      BS = brier_score_loss(Y_test, y_pred)
      BSS = 1 - BS/brier_score_loss(Y_test,
np.mean(Y_test)*np.ones(len(Y_test)))
      return BSS

def train_lr(X_train, Y_train):
      prevalence = np.mean(Y_train)
      t=time()

      log_reg = SGDClassifier(loss = 'log',
                        class_weight = 'balanced', eta0 = 0.01)

      log_reg_params = {'penalty': ['l2', 'l1', 'elasticnet'],
                  'l1_ratio': [0.01, 0.1, 0.5, 1],
                  'alpha': [0.0001, 0.001, 0.01, 0.1],
                  'learning_rate': ['constant', 'optimal', 'invscaling',
'adaptive'],
                  'eta0': [0.001, 0.01, 0.1, 1]}

#     log_reg_params = {'penalty': ['l2', 'l1', 'elasticnet'],
#                       'l1_ratio': [0.01, 0.1, 0.5, 1],
#                       'alpha': [0.0001],
#                       'learning_rate': ['constant'],
#                       'eta0': [0.01]} #always enough though

      log_reg_clf = GridSearchCV(log_reg, log_reg_params, cv = 4,
                                 scoring = 'roc_auc', n_jobs = -1, verbose=0)
#scoring=roc_auc, average_precision, or neg_brier_score

      log_reg_clf.fit(X_train, Y_train)

      print('Logistic classifier trained in', np.timedelta64(int(time()-t),
's'))
```

```
      return log_reg_clf.best_estimator_, log_reg_clf.best_params_

def train_xgb(X_train, Y_train):
      prevalence = np.mean(Y_train)
      t=time()

      xgb = XGBClassifier(objective = 'binary:logistic',
                  n_estimators = 1000, learning_rate = 0.01,
                  scale_pos_weight = (1-prevalence)/prevalence, penalty =
'elasticnet',
                  use_label_encoder=False, verbosity=0)

      xgb_params = {'max_depth': [5],
            'reg_lambda': [0.01, 0.1, 1],
            'gamma': [0.1, 0.3],
            'colsample_bytree': [0.1, 0.3, 0.5]}

#     xgb_params = {'max_depth': [5],
#                   'reg_lambda': [1],
#                   'gamma': [0.1, 0.3],
#                   'colsample_bytree': [0.1]} #always enough though

      xgb_clf = GridSearchCV(xgb, xgb_params, scoring='roc_auc', n_jobs =
1, cv = 4, verbose=3)

      xgb_clf.fit(X_train, Y_train)

      print('XGBoost trained in', np.timedelta64(int(time()-t), 's'))
      return xgb_clf.best_estimator_, xgb_clf.best_params_

def train_ebm(X_train, Y_train, feature_names):
      t=time()
      def scorer(estimator, X, y):
      y_pred = estimator.predict_proba(X)[:, 1]
      auc = roc_auc_score(y, y_pred)
      return auc

      ebm_params = {'learning_rate': [0.01, 0.1, 0.5],
                  'min_samples_leaf': [2, 4],
                  'max_leaves': [3, 5],
                  'max_bins':[256]}
```

```
        param_list = []
        score_list = []
        for params in ParameterGrid(ebm_params):
        param_list.append(params)
        ebm = ExplainableBoostingClassifier(n_jobs=-1, interactions=20,
feature_names=feature_names, **params)
        cv_scores = cross_val_score(ebm, X_train, Y_train, scoring=scorer)
#ebm implements 'fit' so it works
        score_list.append(cv_scores)

        i = np.argmax(np.mean(score_list, axis=1))
        best_params = param_list[i]

        ebm = ExplainableBoostingClassifier(n_jobs=-1, interactions=20,
feature_names=feature_names, **best_params)
        ebm.fit(X_train, Y_train)

        print('EBM trained in', np.timedelta64(int(time()-t), 's'))

        return ebm, best_params

def train(model_type, X_train, Y_train, feature_names=None):
        if model_type=='lr':
        return train_lr(X_train, Y_train)
        elif model_type=='xgb':
        return train_xgb(X_train, Y_train)
        elif model_type=='ebm':
        return train_ebm(X_train, Y_train, feature_names)




def load_data(k, data_folder = '~/MIMIC-III/learning_data/'):
        data = load_from_file(data_folder + 'data_{}.dat'.format(k))
        X, Y, header, ID = data['X'], data['Y'], data['header'], data['ID']
        return X, Y, header, ID


def save_model(model, model_type, k, seed, path = '~/MIMIC-III/models/'):
        with open(path + '{}_{}_{}.pkl'.format(model_type, k, seed), 'wb') as
f:
        pickle.dump(model, f)
```

```
def load_model(model_type, k, seed, path = '~/MIMIC-III/models/'):
      with open(path + '{}_{}_{}.pkl'.format(model_type, k, seed), 'rb') as
f:
      model = pickle.load(f)
      return model


# -

# # Training

for k in [12, 24, 36, 48]:
      X, Y, header, ID =  load_data(k)
      X_train, X_test, Y_train, Y_test = split(X, Y)

      print('Training on {} hours data...'.format(k))

      for model_type in ['lr', 'xgb', 'ebm']:
      if model_type=='lr' and k==12: continue #####WARNING: remove that
line if you rerun this, I had a problem the first time (but the model
exists)
      print('Training {}...'.format(model_type))
      clf, best_params = train(model_type, X_train, Y_train, header)
      print('Best params: {}'.format(best_params))

      save_model(clf, model_type, k, SEED, path = '~/MIMIC-III/models/')

      y_pred = clf.predict_proba(X_test)[:, 1]

      auc = roc_auc_score(Y_test, y_pred)
      ap = average_precision_score(Y_test, y_pred)
      bss = brier_skill_score(Y_test, y_pred)

      print('AUC: {} ; AP: {} ; BSS: {}'.format(round(auc, 3), round(ap,
3), round(bss, 3)))
      print('\n')

for k in [12, 24, 36, 48]:
      X, Y, header, ID =  load_data(k)
      X_train, X_test, Y_train, Y_test = split(X, Y)

      print('Training on {} hours data...'.format(k))
```

```
      for model_type in ['lr', 'xgb']:
      clf = load_model(model_type, k, SEED)
      clf_isotonic = CalibratedClassifierCV(clf, cv=4, method='isotonic')
      clf_isotonic.fit(X_train, Y_train)
      save_model(clf_isotonic, model_type, k, SEED, path =
'~/MIMIC-III/calibrated(iso)/')
      y_pred = clf_isotonic.predict_proba(X_test)[:, 1]

      auc = roc_auc_score(Y_test, y_pred)
      ap = average_precision_score(Y_test, y_pred)
      bss = brier_skill_score(Y_test, y_pred)

      print('AUC: {} ; AP: {} ; BSS: {}'.format(round(auc, 3), round(ap,
3), round(bss, 3)))
      print('\n')


# # Few features model

# +
def few_features(X, Y, header, ID):
      cols = []
      for i in range(len(header)):
      if 'MV' in header[i] or 'GCS' in header[i] or 'WBC' in header[i]:
            cols.append(i)
      return X[:, cols], Y, np.array(header)[cols], ID


for k in [12, 24, 36, 48]:
      X, Y, header, ID =  few_features(*load_data(k)) # <---
      X_train, X_test, Y_train, Y_test = split(X, Y)

      print('Training on {} hours data...'.format(k))

      for model_type in ['lr', 'xgb', 'ebm']:
      print('Training {}...'.format(model_type))
      clf, best_params = train(model_type, X_train, Y_train, header)
      print('Best params: {}'.format(best_params))

      save_model(clf, model_type, k, SEED, path =
'~/MIMIC-III/few_features/')

      y_pred = clf.predict_proba(X_test)[:, 1]
```

```
      auc = roc_auc_score(Y_test, y_pred)
      ap = average_precision_score(Y_test, y_pred)
      bss = brier_skill_score(Y_test, y_pred)

      print('AUC: {} ; AP: {} ; BSS: {}'.format(round(auc, 3), round(ap,
3), round(bss, 3)))
      print('\n')


# +
def few_features(X, Y, header, ID):
      cols = []
      for i in range(len(header)):
      if 'MV' in header[i] or 'GCS' in header[i] or 'WBC' in header[i]:
            cols.append(i)
      return X[:, cols], Y, np.array(header)[cols], ID


for k in [12, 24, 36, 48]:
      X, Y, header, ID =  few_features(*load_data(k)) # <---
      X_train, X_test, Y_train, Y_test = split(X, Y)

      print('Training on {} hours data...'.format(k))

      for model_type in ['ebm']:
      print('Training {}...'.format(model_type))
      clf, best_params = train(model_type, X_train, Y_train, header)
      print('Best params: {}'.format(best_params))

      save_model(clf, model_type, k, SEED, path = '~/MIMIC-III/ebm_env1/')

      y_pred = clf.predict_proba(X_test)[:, 1]

      auc = roc_auc_score(Y_test, y_pred)
      ap = average_precision_score(Y_test, y_pred)
      bss = brier_skill_score(Y_test, y_pred)

      print('AUC: {} ; AP: {} ; BSS: {}'.format(round(auc, 3), round(ap,
3), round(bss, 3)))
      print('\n')

# -
```

```
# # Thinned datasets

# +
k=24

for thinning_rate in [25, 50, 75]:
      X, Y, header, ID =  load_data(k, data_folder =
'~/MIMIC-III/thinned_{}/'.format(thinning_rate))
      X_train, X_test, Y_train, Y_test = split(X, Y)

      print('Training on {} hours data...'.format(k))

      for model_type in ['ebm']:
      print('Training {}...'.format(model_type))
      clf, best_params = train(model_type, X_train, Y_train, header)
      print('Best params: {}'.format(best_params))

      save_model(clf, model_type, k, SEED, path =
'~/MIMIC-III/thinned_{}/'.format(thinning_rate))

      y_pred = clf.predict_proba(X_test)[:, 1]

      auc = roc_auc_score(Y_test, y_pred)
      ap = average_precision_score(Y_test, y_pred)
      bss = brier_skill_score(Y_test, y_pred)

      print('AUC: {} ; AP: {} ; BSS: {}'.format(round(auc, 3), round(ap,
3), round(bss, 3)))
      print('\n')


# +
def few_features(X, Y, header, ID):
      cols = []
      for i in range(len(header)):
      if 'MV' in header[i] or 'GCS' in header[i] or 'WBC' in header[i]:
            cols.append(i)
      return X[:, cols], Y, np.array(header)[cols], ID

for thinning_rate in [25, 50, 75]:
      X, Y, header, ID =  few_features(*load_data(k, data_folder =
'~/MIMIC-III/thinned_{}/'.format(thinning_rate)))
```

```
      X_train, X_test, Y_train, Y_test = split(X, Y)

      print('Training on {} hours data...'.format(k))

      for model_type in ['ebm']:
      print('Training {}...'.format(model_type))
      clf, best_params = train(model_type, X_train, Y_train, header)
      print('Best params: {}'.format(best_params))

      save_model(clf, model_type, k, SEED, path =
'~/MIMIC-III/thinned_{}/'.format(thinning_rate))

      y_pred = clf.predict_proba(X_test)[:, 1]

      auc = roc_auc_score(Y_test, y_pred)
      ap = average_precision_score(Y_test, y_pred)
      bss = brier_skill_score(Y_test, y_pred)

      print('AUC: {} ; AP: {} ; BSS: {}'.format(round(auc, 3), round(ap,
3), round(bss, 3)))
      print('\n')
# -

# # Cross performance (train everything on k=48 patient population)

for k in [12, 24, 36, 48]:
      X, Y, header, ID =  load_data(k)
      _, _, _, ID48 = load_data(48)
      i = np.where(np.isin(ID, ID48))
      X, Y, ID = X[i], np.array(Y)[i], np.array(ID)[i]

      X_train, X_test, Y_train, Y_test = split(X, Y)

      print('Training on {} hours data...'.format(k))

      for model_type in ['lr', 'xgb', 'ebm']:
      print('Training {}...'.format(model_type))
      clf, best_params = train(model_type, X_train, Y_train, header)
      print('Best params: {}'.format(best_params))

      save_model(clf, model_type, k, SEED, path =
'~/MIMIC-III/robustness/')
```

```
        y_pred = clf.predict_proba(X_test)[:, 1]

        auc = roc_auc_score(Y_test, y_pred)
        ap = average_precision_score(Y_test, y_pred)
        bss = brier_skill_score(Y_test, y_pred)

        print('AUC: {} ; AP: {} ; BSS: {}'.format(round(auc, 3), round(ap,
3), round(bss, 3)))
        print('\n')


##############################
# _6_Analysis
# ##############################

# This part is used to generate all the figures for the article. The last
bit (risk vs feature plots) need to run on env1 for work (python library
versions problem), everything else on env2, as all the parts. Some cells
have to be modified to generate different variations of the figures
(everything is indicated)

# +
import sys
import utils
import collections
import os

import numpy as np
import pandas as pd
import logging
logging.basicConfig(level = logging.WARNING)

from tqdm import tqdm, trange
from time import time
import matplotlib.pyplot as plt
# %matplotlib inline
plt.style.use('seaborn-darkgrid')

import load_from_file, get_detailed_performance
import sklearn
from sklearn.linear_model import LogisticRegression, SGDClassifier

from sklearn.model_selection import train_test_split, GridSearchCV
```

```
from sklearn.impute import SimpleImputer
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler# , Imputer
from sklearn.calibration import calibration_curve
from sklearn import svm
import pickle

import keras
import xgboost
from xgboost.sklearn import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score, mean_squared_error,
classification_report, plot_roc_curve
from sklearn.metrics import precision_recall_curve,
average_precision_score, brier_score_loss

from bisect import bisect

from interpret import show
from interpret.data import ClassHistogram
from interpret.glassbox import ExplainableBoostingClassifier

from interpret import set_visualize_provider
from interpret.provider import InlineProvider
set_visualize_provider(InlineProvider())

from interpret.perf import ROC

# +
SEED = 5299
logging.disable(logging.WARNING)

def split(X, Y, seed=SEED):
      '''
      X: 2D float array of training examples with dimensions (n_examples,
n_features)
      Y: 1D binary array of labels with dimension n_examples
      seed: int used for the split. With the same seed, datasets of same
size will be splitted the same

      outputs: X_train, X_test, Y_train, Y_test

      Splits a dataset into training and testing array,
```

```
        with median imputation for nans and 0-1 normalization
        (both based on the training set).
        '''
        #Split
        X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size =
0.2, shuffle = True, random_state=seed)

        #Input missing values
        impute_median = SimpleImputer(missing_values = np.nan, strategy =
'median')
        impute_median.fit(X_train)

        X_train_imputed = impute_median.transform(X_train)
        X_test_imputed = impute_median.transform(X_test)

        X_train = np.array(X_train_imputed)
        X_test = np.array(X_test_imputed)

        #0-1 normalization
        train_min, train_max = np.min(X_train, axis=0), np.max(X_train,
axis=0)
        X_train = (X_train-train_min)/(train_max-train_min)
        X_test = (X_test-train_min)/(train_max-train_min)

#       m, sd = np.mean(X_train, axis=0), np.std(X_train, axis=0)
#       X_train = (X_train-m)/sd
#       X_test = (X_test-m)/sd

        return X_train, X_test, Y_train, Y_test

def brier_skill_score(Y_test, y_pred):
        BS = brier_score_loss(Y_test, y_pred)
        BSS = 1 - BS/brier_score_loss(Y_test,
np.mean(Y_test)*np.ones(len(Y_test)))
        random_bss = 1 - (1/3) / brier_score_loss(Y_test,
np.mean(Y_test)*np.ones(len(Y_test)))
        return BSS, random_bss

def load_data(k, data_folder = '~/MIMIC-III/learning_data/'):
        data = load_from_file(data_folder + 'data_{}.dat'.format(k))
        X, Y, header, ID = data['X'], data['Y'], data['header'], data['ID']
        return X, Y, header, ID
```

```
def save_model(model, model_type, k, seed, path = '~/MIMIC-III/models/'):
      with open(path + '{}_{}_{}.pkl'.format(model_type, k, seed), 'wb') as
f:
      pickle.dump(model, f)

def load_model(model_type, k, seed, path = '~/MIMIC-III/models/'):
      with open(path + '{}_{}_{}.pkl'.format(model_type, k, seed), 'rb') as
f:
      model = pickle.load(f)
      return model

def few_features(X, Y, header, ID):
      cols = []
      for i in range(len(header)):
      if 'MV' in header[i] or 'GCS' in header[i] or 'WBC' in header[i]:
            cols.append(i)
      return X[:, cols], Y, np.array(header)[cols], ID

def get_roc(model_type, k, seed, path = '~/MIMIC-III/models/'):
      X, Y, header, ID =  load_data(k) #### MODIFY here for few_features
roc curves: few_features(*load_data(k))
      X_train, X_test, Y_train, Y_test = split(X, Y)
      clf = load_model(model_type, k, seed, path = path)
      y_pred = clf.predict_proba(X_test)[:, 1]
      report = get_detailed_performance(Y_test, y_pred)
      return report['roc_x'], report['roc_y'], round(report['auroc'], 3)

def get_pr(model_type, k, seed, path = '~/MIMIC-III/models/'):
      X, Y, header, ID =  load_data(k)
      X_train, X_test, Y_train, Y_test = split(X, Y)
      clf = load_model(model_type, k, seed, path = path)
      y_pred = clf.predict_proba(X_test)[:, 1]
      precision, recall, _ = precision_recall_curve(Y_test, y_pred)
      ap = average_precision_score(Y_test, y_pred)
      return precision, recall, round(ap, 3), np.mean(Y_train)

def get_f(model_type, k, seed, path = '~/MIMIC-III/models/'):
      clf = load_model(model_type, k, seed, path=path)
      _, _, standard_header, _ = load_data(48) #disgusting but hey
      if model_type=='lr':
      return np.array(standard_header[:87]), np.array(clf.coef_[0])
#warning: pos and neg values: take abs
```

```
        if model_type=='xgb':
        return np.array(standard_header[:87]),
np.array(clf.feature_importances_)
        if model_type=='ebm':
        report = clf.explain_global().data()
        return np.array(report['names']), np.array(report['scores'])

def get_calibration(model_type, k, seed, path = '~/MIMIC-III/models/'):
        X, Y, header, ID =  load_data(k)
        X_train, X_test, Y_train, Y_test = split(X, Y)
        prevalence = np.mean(Y_test)
        clf = load_model(model_type, k, seed, path=path)

        y_pred = clf.predict_proba(X_test)[:, 1]

        fop, mpv = calibration_curve(Y_test, y_pred, normalize = False,
n_bins=8, strategy='quantile')
        bss, random_bss = brier_skill_score(Y_test, y_pred)
        return mpv, fop, prevalence, round(bss, 3), round(random_bss, 3)

results_path = '~/MIMIC-III/results/'
# -

# # Performance curves

# +
k = 12 #Set that

model_titles = {'lr':'Logistic Regression', 'xgb':'XGBoost', 'ebm':'EBM'}

for model_type in ['lr', 'xgb', 'ebm']:
        roc_x, roc_y, auc = get_roc(model_type, k, seed=SEED)
        plt.plot(roc_x, roc_y, label='{}
(AUC={})'.format(model_titles[model_type], auc))
plt.plot([0, 1], [0, 1], label='Random (AUC=0.5)', linestyle='--')
plt.xlabel('1-Specificity')
plt.ylabel('Sensitivity')
plt.title('ROC curves for {}h window'.format(k))
plt.legend()
plt.savefig(results_path + 'ROC_' + '{}.png'.format(k))
plt.show()

# +
```

```
k = 48 #set that

model_titles = {'lr':'Logistic Regression', 'xgb':'XGBoost', 'ebm':'EBM'}

for model_type in ['lr', 'xgb', 'ebm']:
      precision, recall, ap, prevalence = get_pr(model_type, k, seed =
SEED)
      plt.plot(recall, precision, label='{}
(AP={})'.format(model_titles[model_type], ap))
plt.plot([0, 1], [prevalence, prevalence], label='Random', linestyle='--')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('PR curves for {}h window'.format(k))
plt.legend()
plt.savefig(results_path + 'PR_' + '{}.png'.format(k))
plt.show()

# +
k = 48 #set that modify where indicated for uncalibrated/calibrated models.
ebm is already calibrated

model_titles = {'lr':'Logistic Regression', 'xgb':'XGBoost', 'ebm':'EBM'}

for model_type in ['lr', 'xgb', 'ebm']:
      if model_type=='ebm':
      mpv, fop, prevalence, bss, random_bss = get_calibration(model_type,
k, seed = SEED, path = '~/MIMIC-III/models/')
      else: #####here
      mpv, fop, prevalence, bss, random_bss = get_calibration(model_type,
k, seed = SEED, path = '~/MIMIC-III/models/calibrated(iso)/')
      plt.plot(mpv, fop, label='{}
(BSS={})'.format(model_titles[model_type], bss))
plt.xlabel('Mean Predicted Value')
plt.ylabel('Fraction of Positive')
plt.plot([0, 1], [prevalence, prevalence], label='Random
(BSS={})'.format(random_bss), linestyle='--', linewidth=0.75)
plt.plot([0, 1], [0, 1], label='Perfectly calibrated (BSS=1)',
linestyle='--', linewidth=0.75)
plt.plot([prevalence], [prevalence], marker='*', label='Dummy (BSS=0)',
linewidth=0)
plt.title('Calibration curves for {}h window (isotonic
calibration)'.format(k)) #####here
plt.legend()
```

```
plt.savefig(results_path + 'Cal_' + '{}.png'.format(k))
plt.show()

# +
k = 48 #set that
model_type = 'xgb' #set that
n_features = 20

model_titles = {'lr':'Logistic Regression', 'xgb':'XGBoost', 'ebm':'EBM'}

plt.rcParams["figure.figsize"] = (8, 6) #change if needed, but prettier
like that
names, scores = get_f(model_type, k, SEED, path = '~/MIMIC-III/models/')
sorted_idx = np.abs(scores).argsort()
plt.barh(names[sorted_idx][-n_features:],
100*np.abs(scores[sorted_idx][-n_features:])/np.sum(np.abs(scores)))
plt.xlabel('Feature importance (% of absolute total score)')
plt.title('{}, k={}'.format(model_titles[model_type], k))
plt.savefig(results_path + 'Features_' + '{}_{}.png'.format(model_type, k))
plt.show()
plt.rcParams["figure.figsize"] = (6.4, 4.8) #default fig size
# -

# # Few features EBM

# +
model_type = 'ebm' #set that

model_titles = {'lr':'Logistic Regression', 'xgb':'XGBoost', 'ebm':'EBM'}

for k in [12, 24, 36, 48]:
      #######WARNING: modify the get_roc function as indicated for this
cell (JUST this one)
      roc_x, roc_y, auc = get_roc(model_type, k, seed=SEED,
path='~/MIMIC-III/models/few_features/')
      plt.plot(roc_x, roc_y, label='k={} (AUC={})'.format(k, auc))
plt.plot([0, 1], [0, 1], label='Random (AUC=0.5)', linestyle='--')
plt.xlabel('1-Specificity')
plt.ylabel('Sensitivity')
plt.title('EBM ROC curves (few features)'.format(k))
plt.legend()
plt.savefig(results_path + 'EBM_ROC_few.png')
plt.show()
```

```
# -

# # Robustness

# ### Thinning

# +
def doublesplit(X_1, Y_1, X_2, Y_2):
      X_train, X_ref, Y_train, Y_ref = train_test_split(X_1, Y_1, test_size
= 0.2, shuffle = True, random_state = SEED)
      _, X_test, _, Y_test = train_test_split(X_2, Y_2, test_size = 0.2,
shuffle = True, random_state = SEED)

      impute_median = SimpleImputer(missing_values = np.nan, strategy =
'median')
      impute_median.fit(X_train)

      X_train = np.array(impute_median.transform(X_train))
      X_ref = np.array(impute_median.transform(X_ref))
      X_test = np.array(impute_median.transform(X_test))

      #0-1 normalization
      train_min, train_max = np.min(X_train, axis=0), np.max(X_train,
axis=0)
      X_train = (X_train-train_min)/(train_max-train_min)
      X_ref = (X_ref-train_min)/(train_max-train_min)
      X_test = (X_test-train_min)/(train_max-train_min)

      return X_train, X_ref, X_test, Y_train, Y_ref, Y_test


####change where it is written for few_features: add to path, or
few_features(*load_data(...)). 6 changes in total

default_model = load_model('ebm', 24, seed=SEED, path =
'~/MIMIC-III/models/few_features/')  ######here
default_X, default_Y, header, ID =  few_features(*load_data(24)) #####here
X_train, X_test, Y_train, Y_test = split(default_X, default_Y)

y_pred = default_model.predict_proba(X_test)[:, 1]
auc = roc_auc_score(Y_test, y_pred)
```

```
default_auc = [auc]
thinned_auc = [auc]


for thinning_rate in [25, 50, 75]:
      #####here
      X, Y, header, ID =  few_features(*load_data(24, data_folder =
'~/MIMIC-III/learning_data/thinned_{}/'.format(thinning_rate)))
      _, _, X_test, _, _, Y_test = doublesplit(default_X, default_Y, X, Y)
#the "unthinned" model needs to see the thinned data under its perspective
for a fair comparison
      default_auc.append(roc_auc_score(Y_test,
default_model.predict_proba(X_test)[:, 1]))

      #####here
      thinned_model = load_model('ebm', 24, seed=SEED, path =
'~/MIMIC-III/models/few_features/thinned_{}/'.format(thinning_rate))
      X_train, X_test, Y_train, Y_test = split(X, Y)
      thinned_auc.append(roc_auc_score(Y_test,
thinned_model.predict_proba(X_test)[:, 1]))

default_auc, thinned_auc = np.array(default_auc), np.array(thinned_auc)
plt.scatter(np.arange(4), default_auc, label='Model trained on unthinned
data', s=50)
plt.scatter(np.arange(4), thinned_auc, label='Model trained on the thinned
data', s=50)
plt.xticks(np.arange(4), label=[0, 25, 50, 75])
plt.xlabel('Thinning rate')
plt.ylabel('AUC')
plt.ylim(0.5, 1)
plt.legend()
plt.title('EBM, 24h window (few features)') ####here
plt.savefig(results_path + 'Thinning_few_features.png') ####here
plt.show()
print(default_auc, thinned_auc)

# -

# ### Cross performance

# +
results = {} #keys: model then data
```

```
for k in [12, 24, 36, 48]:
      for model_type in ['lr', 'xgb', 'ebm']:

      model = load_model(model_type, k, seed=SEED, path =
'~/MIMIC-III/models/robustness/')
      results[model_type+'_'+str(k)] = {}

      for data_k in [12, 24, 36, 48]:

            X, Y, header, ID =  load_data(data_k)
            _, _, _, ID48 = load_data(48)
            i = np.where(np.isin(ID, ID48))
            X, Y, ID = X[i], np.array(Y)[i], np.array(ID)[i]

            X_train, X_test, Y_train, Y_test = split(X, Y)

            y_pred = model.predict_proba(X_test)[:, 1]

            auc = roc_auc_score(Y_test, y_pred)
            ap = average_precision_score(Y_test, y_pred)
            bss = brier_skill_score(Y_test, y_pred)

            results[model_type+'_'+str(k)][data_k]=(auc, ap, bss)


# +
import seaborn as sns
model_titles = {'lr':'Logistic Regression', 'xgb':'XGBoost', 'ebm':'EBM'}

model_type = 'ebm' #set that

table = np.zeros((4, 4))
for i in range(4):
      for j in range(4):
      table[i, j] = results[model_type+'_'+str([12, 24, 36, 48][i])][[12,
24, 36, 48][j]][0]
sns.heatmap(table, vmin=0.5, vmax=1, annot=True, yticklabels = [str([12,
24, 36, 48][i]) for i in range(4)],
            xticklabels = [str([12, 24, 36, 48][i]) for i in range(4)])
plt.xlabel('Time window for testing')
plt.ylabel('Time window for training')
plt.title('Cross performance of {} for different time
```

```
windows'.format(model_titles[model_type]))
plt.savefig(results_path + 'grid_{}.png'.format(model_type))
plt.show()
# -

# # Risk vs feature plot for few_features model

# +
from interpret import show
from interpret.data import ClassHistogram
from interpret.glassbox import ExplainableBoostingClassifier
from interpret import set_visualize_provider
from interpret.provider import InlineProvider
set_visualize_provider(InlineProvider())

ebm = load_model('ebm', 24, seed=SEED, path =
'~/MIMIC-III/models/few_features/ebm_env1/')
explain = ebm.explain_global()

report = explain.data()
names, scores = np.array(report['names']), np.array(report['scores'])

n_features = 20
model_titles = {'lr':'Logistic Regression', 'xgb':'XGBoost', 'ebm':'EBM'}

plt.rcParams["figure.figsize"] = (8, 6)
sorted_idx = np.abs(scores).argsort()
plt.barh(names[sorted_idx][-n_features:],
100*np.abs(scores[sorted_idx][-n_features:])/np.sum(np.abs(scores)))
plt.xlabel('Feature importance (% of absolute total score)')
plt.title('{}, k={}'.format(model_titles['ebm'], 24))
plt.show()
plt.rcParams["figure.figsize"] = (6.4, 4.8)

#Look at top 3 features
# -

show(explain)
```