Assignment 4 : Predictive Process Monitoring

Gamal Elkoumy

University of Tartu, 2021

In this document, we report the our solution to the predictive process monitoring presented here https://courses.cs.ut.ee/LTAT.05.025/2021_spring/uploads/Main/2021Homework4.pdf

Solution GitHub Repository

Our solution is available using the following URL. https://github.com/Elkoumy/predictive-monitoring-benchmark

Task 1

(1 point)

As part of the log preprocessing, it is necessary to categorize the process traces as deviant or regular. This log contains a column called SLA. it is a "case attribute," which indicates how many minutes each case must complete. You must create a new column in the log that contains a case attribute called label, which contains a value of 1 for deviant cases or 0 for regular ones. This column's value is 0 if the duration of the case (in minutes) is less than or equal to the SLA; otherwise, this column's value must be 1 (the SLA has not been met). NB! If there are cases that do not have SLA, categorize them as 0.

In [4]:

```
import numpy as np
import pandas as pd
```

In [5]:

```
df=pd.read_csv(r'C:\Gamal Elkoumy\PhD\OneDrive - Tartu Ülikool\Courses\Process Mining\Ass
ignment4\predictive-monitoring-benchmark\data\turnaround_anon_sla.csv')

#converting datatypes , timestamps
df.start_timestamp= pd.to_datetime(df.start_timestamp,utc=True)
df.end_timestamp= pd.to_datetime(df.end_timestamp,utc=True)

df=df.sort_values(['start_timestamp']).reset_index()
df=df.drop('index',axis=1)
```

In [6]:

```
"""Q1"""

#calculating the start time and end time of every case
df['case_end_time']=df.groupby(['caseid']).end_timestamp.transform('max')
df['case_start_time']=df.groupby(['caseid']).start_timestamp.transform('min')

#calculating case duration in minutes ( the same time unit as the SLA)
df['duration']=(df.case_end_time-df.case_start_time).astype('timedelta64[m]')


#creating the label column
df['label']=1
df.loc[df.duration<=df['SLA MIN'],'label']=0
```

```
df.loc[df['SLA MIN'].isna(),'label']=0
```

Task 2

(2 points)

Add a column to the event log that captures the WIP of the process at the moment where the last eventin the prefix occurs. Remember that the WIP refers to the number of active cases, meaning the number of cases that have started but not yet completed.

First, we define a funtion that performs the estimation of wip for each activity.

In [8]:

```
def count_wip(row, case_times):
    wip=0
    #started before start and ended after end
    #started after start and ended before end
    #started before start and ended before end
    #started before end and ended after end
    wip=case_times.loc[(case_times.case_start_time<= row.start_timestamp) & (case_times.
case_end_time>=row.end_timestamp) |
                       (case_times.case_start_time >= row.start_timestamp) & (case_times
.case_end_time <= row.end_timestamp)|
                       (case_times.case_start_time <= row.start_timestamp) & (case_times
.case_end_time >= row.start_timestamp)|
                       (case_times.case_start_time <= row.end_timestamp) & (case_times.c
ase_end_time >= row.end_timestamp)
                       ].shape[0]

    return wip
```

We then use the pandas apply function to execute the count_wip function as follows.

In [9]:

```
"""Q2"""
case_times= pd.DataFrame()
case_times['case_end_time']=df.groupby(['caseid']).end_timestamp.max()
case_times['case_start_time']=df.groupby(['caseid']).start_timestamp.min()
case_times=case_times.reset_index()

df['WIP']=df.apply(count_wip,case_times=case_times ,axis=1)
```

We export the result in order to use it separately to optimize the model parameters as we will mention later.

In [10]:

```
df=df.rename(columns={'caseid': 'Case ID','activity':'Activity', 'start_timestamp':'time
:timestamp'})
df.to_csv(r'C:\Gamal Elkoumy\PhD\OneDrive - Tartu Ülikool\Courses\Process Mining\Assignme
nt4\predictive-monitoring-benchmark\experiments\experiment_log\turnaround_anon_sla_rename
d.csv',index=False, sep=';')
```

As a preprocessing for the next step, we prepare the data for the train/test split.

In [12]:

```
# split into training and test
def split_data_strict(data, train_ratio, split="temporal"):
    # split into train and test using temporal split and discard events that overlap the
periods
    data = data.sort_values(['time:timestamp', 'Activity'], ascending=True, kind='merges
```

```
ort')
    grouped = data.groupby('Case ID')
    start_timestamps = grouped['time:timestamp'].min().reset_index()
    start_timestamps = start_timestamps.sort_values('time:timestamp', ascending=True, ki
nd='mergesort')
    train_ids = list(start_timestamps['Case ID'])[:int(train_ratio*len(start_timestamps)
)]
    train = data[data['Case ID'].isin(train_ids)].sort_values(['time:timestamp', 'Activi
ty'], ascending=True, kind='mergesort')
    test = data[~data['Case ID'].isin(train_ids)].sort_values(['time:timestamp', 'Activi
ty'], ascending=True, kind='mergesort')
    split_ts = test['time:timestamp'].min()
    train = train[train['time:timestamp'] < split_ts]
    return (train, test)
```

In [13]:

```
"""Split into train and test"""
train_ratio = 0.8
n_splits = 2
random_state = 22

train, test = split_data_strict(df, train_ratio, split="temporal")
```

Task 3

(4 points)

Currently, the work proposed by Taineema et al. performs the extraction of the prefixes of the traces registered in the log to train the classification models. For large logs, this approach leads to an increase in the dimensionality of the models' input (too many features) without necessarily improving its precision, especially in cases in which the event traces are very long. You must modify this technique to extract subsequences of size n (n-grams), where n is a userdefined parameter, instead of encoding entire prefixes. An n-gram is a contiguous sequence of n items from a given trace.

First, we define the function that calculates the n-grams. The following function calculates the prefixes using the n-grams for every case separately.

In [20]:

```
def create_ngrams(data, ngram_size):
    result=pd.DataFrame()


    for idx in range(0,data.shape[0]- ngram_size +1):

        prefix=data.iloc[idx:idx+ngram_size].copy()
        prefix=prefix.reset_index()

        prefix['Case ID']=prefix['Case ID']+'_'+str(idx)

        result=pd.concat([result,prefix])

    return result
```

As a helper function, we adapted the following method to the new label values.

In [21]:

```
def get_class_ratio(data):
    class_freqs = data['label'].value_counts()
    return class_freqs[1] / class_freqs.sum()
```

We then follow the same CV method as the practice session 10.

In [22]:

```
from sklearn.model_selection import StratifiedKFold
def get_stratified_split_generator(data, n_splits=5, shuffle=True, random_state=22):
    grouped_firsts = data.groupby('Case ID', as_index=False).first()
    skf = StratifiedKFold(n_splits=n_splits, shuffle=shuffle, random_state=random_state)

    for train_index, test_index in skf.split(grouped_firsts, grouped_firsts['label']):
        current_train_names = grouped_firsts['Case ID'][train_index]
        train_chunk = data[data['Case ID'].isin(current_train_names)].sort_values('time:
timestamp', ascending=True, kind='mergesort')
        test_chunk = data[~data['Case ID'].isin(current_train_names)].sort_values('time:
timestamp', ascending=True, kind='mergesort')
        yield (train_chunk, test_chunk)
```

In [23]:

```
# prepare chunks for CV
dt_prefixes = []
class_ratios = []
min_prefix_length = 1
ngram_size=5

for train_chunk, test_chunk in get_stratified_split_generator(train, n_splits=n_splits):
    class_ratios.append(get_class_ratio(train_chunk))
    # generate data where each prefix is a separate instance
    dt_prefixes.append(generate_prefix_data(test_chunk, ngram_size))
del train
```

Task 4

(3 points)

Test the results of your modifications with the Turnaround process event log using cluster bucketing, index encoding, and the XGboost model.

Model Parameter Optimization

Taineema et al provide a method for optimizing the model parameters for predictive process monitoring. The file [optimize_params.py]() performs the parameter optimization. We adopted the file by adding the required parameters for the input event log "turnaround_anon_sla.csv".

We needed also to perform adaptations in the file [dataset_confs.py]() in order to enable the parameter tuning for the dataset "turnaround_anon_sla.csv".

We used the following command to execute the optimizer: python optimize_params.py turnaround_anon_sla_renamed optimizer_log 10 cluster index xgboost

The output of the optimizer could be found in the folder [optimizer_log](). Also, the optimial parameters are in the pickle file optimal_params_xgboost_turnaround_anon_sla_renamed_cluster_index.pickle

Cluster Bucketing

We used the Cluster Bucketing methods provided in practice session 10.

In [24]:

```
import BucketFactory

# encoding_method = "last", "agg", "index"
# Bucketing prefixes based on control flow
bucketer_args = {'encoding_method': 'last',
                 'case_id_col': 'Case ID',
                 'cat_cols':['Activity'],
                 'num_cols':[],
                 'random_state':random_state}
```

```
cv_iter = 0
dt_test_prefixes = dt_prefixes[cv_iter]
dt_train_prefixes = pd.DataFrame()
for cv_train_iter in range(n_splits):
    if cv_train_iter != cv_iter:
        dt_train_prefixes = pd.concat([dt_train_prefixes, dt_prefixes[cv_train_iter]], a
xis=0)


cv_iter = 0
dt_test_prefixes = dt_prefixes[cv_iter]
dt_train_prefixes = pd.DataFrame()
for cv_train_iter in range(n_splits):
    if cv_train_iter != cv_iter:
        dt_train_prefixes = pd.concat([dt_train_prefixes, dt_prefixes[cv_train_iter]], a
xis=0)
```

In [25]:

```
""" ************* Performing Cluster Bucketing ****************"""

#bucket_methods = "single", "prefix", "state", "cluster", "knn"
bucket_method = 'cluster'
if bucket_method == "cluster":
    bucketer_args["n_clusters"] = 3
bucketer = BucketFactory.get_bucketer(bucket_method, **bucketer_args)
bucket_assignments_train = bucketer.fit_predict(dt_train_prefixes)
bucket_assignments_test = bucketer.predict(dt_test_prefixes)

""" Train buckets"""
bucket_number = 2
bucket_indexes = dt_train_prefixes.groupby('Case ID').first().index
bucket_indexes = bucket_indexes[bucket_assignments_train == bucket_number]
print(bucket_indexes)
bucket_data = dt_train_prefixes[dt_train_prefixes['Case ID'].isin(bucket_indexes)]

def get_label_numeric(data):
    y = data.groupby('Case ID').first()['label'] # one row per case
    return y
train_y = get_label_numeric(bucket_data)
```

```
Index(['Case00_6', 'Case01_18', 'Case02_10', 'Case03_20', 'Case04_14',
       'Case08_9', 'Case10_12', 'Case11_18', 'Case13_18', 'Case17_7',
       'Case19_5', 'Case22_23', 'Case24_11', 'Case28_14', 'Case30_11',
       'Case31_24', 'Case34_20', 'Case35_23', 'Case40_13', 'Case41_19',
       'Case42_16', 'Case44_15', 'Case47_2', 'Case48_13', 'Case51_16'],
      dtype='object', name='Case ID')
```

In [27]:

```
"""Test Buckets"""

bucket_indexes = dt_test_prefixes.groupby('Case ID').first().index
bucket_indexes = bucket_indexes[bucket_assignments_test == bucket_number]
bucket_data_test = dt_test_prefixes[dt_test_prefixes['Case ID'].isin(bucket_indexes)]
test_y = get_label_numeric(bucket_data_test)
```

Performing Encoding Indexing

We used the Encoding Indexing methods provided in practice session 10.

In [28]:

```
""" ************* Perfroming Index Encoding ********************"""
import EncoderFactory
from sklearn.pipeline import FeatureUnion, Pipeline

cls_encoder_args = {'case_id_col': 'Case ID',
                    'static_cat_cols': [],
```

```
                    'static_num_cols': [],
                    'dynamic_cat_cols': ['Activity'],
                    'dynamic_num_cols': ["WIP"],
                    'fillna': True}

encoding_dict = {
    "laststate": ["static", "last"],
    "agg": ["static", "agg"],
    "index": ["static", "index"],
    "combined": ["static", "last", "agg"]
}

methods = encoding_dict['index']

feature_combiner = FeatureUnion([(method, EncoderFactory.get_encoder(method, **cls_encod
er_args)) for method in methods])

encoding = feature_combiner.fit_transform(bucket_data, train_y)

pd.DataFrame(encoding).to_csv('encoding.csv')
```

XGBoost Training

In the following code, we use the model parameters optimized as mentioned above. The output of the optimizer could be found in the folder ```optimizer_log```. Also, the optimial parameters are in the pickle file ```optimal_params_xgboost_turnaround_anon_sla_renamed_cluster_index.pickle```

In [29]:

```
"""******** Perfroming training **************"""

import xgboost as xgb



model_parameters=pd.read_pickle(r'C:\Gamal Elkoumy\PhD\OneDrive - Tartu Ülikool\Courses\P
rocess Mining\Assignment4\predictive-monitoring-benchmark\experiments\optimizer_log\optim
al_params_xgboost_turnaround_anon_sla_renamed_cluster_index.pickle')

model= xgb.XGBClassifier(**model_parameters)


pipeline = Pipeline([('encoder', feature_combiner), ('cls', model)])
pipeline.fit(bucket_data, train_y)

preds_pos_label_idx = np.where(model.classes_ == 1)[0][0]
preds = pipeline.predict_proba(bucket_data_test)[:,preds_pos_label_idx]

from sklearn.metrics import roc_auc_score
score = roc_auc_score(test_y, preds)
print("The ROC AUC is : %s"%(score))
```

```
C:\ProgramData\Anaconda3\lib\site-packages\xgboost\sklearn.py:1146: UserWarning: The use
of label encoder in XGBClassifier is deprecated and will be removed in a future release.
To remove this warning, do the following: 1) Pass option use_label_encoder=False when con
structing XGBClassifier object; and 2) Encode your labels (y) as integers starting with 0
, i.e. 0, 1, 2, ..., [num_class - 1].
  warnings.warn(label_encoder_deprecation_msg, UserWarning)
```

```
[12:36:29] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/lear
ner.cc:573:
Parameters: { "n_clusters" } might not be used.

  This may not be accurate due to some parameters are only used in language bindings but
  passed down to XGBoost core.  Or some parameters are not used but slip through this
  verification. Please open an issue if you find above cases.


[12:36:29] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.4.0/src/lear
ner.cc:1095: Starting in XGBoost 1.3.0, the default evaluation metric used with the objec
```

```
tive 'binary:logistic' was changed from 'error' to 'logloss'. Explicitly set eval_metric
if you'd like to restore the old behavior.
The ROC AUC is : 0.5
```

```
C:\ProgramData\Anaconda3\lib\site-packages\xgboost\data.py:112: UserWarning: Use subset (
sliced data) of np.ndarray is not recommended because it will generate extra copies and i
ncrease memory consumption
  warnings.warn(
```