Assignment4_Elkoumy

May 4, 2021

#

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

0.1 Solution GitHub Repository

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

0.2 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.

[1]:
```
import EncoderFactory
from DatasetManager import DatasetManager
import BucketFactory
import pandas as pd
import numpy as np
from sklearn.metrics import roc_auc_score
from sklearn.pipeline import FeatureUnion, Pipeline
import os
import pickle
import xgboost as xgb
```

[2]:
```
df=pd.read_csv(r'C:\Gamal Elkoumy\PhD\OneDrive - Tartu Ülikool\Courses\Process␣
 ↪Mining\Assignment4\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)
```

[3]:
```
"""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
```

0.3 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.

[4]:
```
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.case_end_time >= row.end_timestamp)
                        ].shape[0]

    return wip
```

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

[5]:
```
"""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.

[6]:
```
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\Assignment4\predictive-monitoring-benchmark\experiments\experiment_log\turnaround_an
 ↪csv',index=False, sep=';')
```

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

[8]:
```
dataset_ref = "turnaround_anon_sla_renamed"
params_dir = "optimizer_log"
results_dir = "experiment_log"
bucket_method = "cluster"
cls_encoding = "index"
cls_method = "xgboost"
ngram_size = 4
bucket_encoding = "agg"
method_name = "%s_%s" % (bucket_method, cls_encoding)

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

methods = encoding_dict[cls_encoding]

train_ratio = 0.8
random_state = 22
```

[9]:
```python
# create results directory
if not os.path.exists(os.path.join(params_dir)):
    os.makedirs(os.path.join(params_dir))

dataset_name=dataset_ref
```

We use the DataManager class in order to perform the train/testsplit. We adapted the code to fit the current event log.

[10]:
```python
"""Split into train and test"""
# read the data
dataset_manager = DatasetManager(dataset_name)
data = dataset_manager.read_dataset()
cls_encoder_args = {'case_id_col': dataset_manager.case_id_col,
                    'static_cat_cols': dataset_manager.static_cat_cols,
                    'static_num_cols': dataset_manager.static_num_cols,
                    'dynamic_cat_cols': dataset_manager.dynamic_cat_cols,
                    'dynamic_num_cols': dataset_manager.dynamic_num_cols,
                    'fillna': True}



# split into training and test
train, test = dataset_manager.split_data_strict(data, train_ratio,␣
 ↪split="temporal")
```

```
Index(['Case ID', 'Activity', 'time:timestamp', 'end_timestamp', 'SLA MIN',
       'case_end_time', 'case_start_time', 'duration', 'label', 'WIP'],
      dtype='object')
```

0.4 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.

[11]:
```python
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)
        prefix['prefix_nr'] = idx + 1
        result=pd.concat([result,prefix])

    return result
```

We modified the function generate_prefix_data inside the DatasetManager class in order to apply the ngrams. The new function is provided below.

[ ]:
```
def generate_prefix_data(self,data, ngram_size):
    # generate prefix data (each possible prefix becomes a trace)

    # ngram_size=3
    dt_prefixes=data.groupby(['Case ID']).apply(create_ngrams, ngram_size)

    dt_prefixes=dt_prefixes.rename(columns={'Case ID': 'newcaseid'})
    dt_prefixes=dt_prefixes.reset_index().rename(columns={'Case ID':␣
 ↪'original_caseid'})
    dt_prefixes=dt_prefixes.drop('level_1',axis=1)
    dt_prefixes=dt_prefixes.rename(columns={'newcaseid': 'Case ID'})

    return dt_prefixes
```

0.4.1 Prefix Generation

[12]:
```
#for test prefixes
dt_test_prefixes = dataset_manager.generate_prefix_data(test, ngram_size)
# for train prefixes
dt_train_prefixes = dataset_manager.generate_prefix_data(train, ngram_size)
```

0.5 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.

0.5.1 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

0.5.2 Cluster Bucketing

Following our adaptation for the "experiment.py" file to perform the training with cluster bucketing, index encoding and the XGBoost model.

[14]:
```
# Bucketing prefixes based on control flow
bucketer_args = {'encoding_method': bucket_encoding,
                 'case_id_col': dataset_manager.case_id_col,
                 'cat_cols': [dataset_manager.activity_col],
                 'num_cols': [],
                 'random_state': random_state}



# load optimal params
optimal_params_filename = os.path.join(params_dir,
                                       "optimal_params_%s_%s_%s.pickle" %␣
 ↪(cls_method, dataset_name, method_name))

with open(optimal_params_filename, "rb") as fin:
    args = pickle.load(fin)

if bucket_method == "cluster":
    bucketer_args["n_clusters"] = int(args["n_clusters"])
bucketer = BucketFactory.get_bucketer(bucket_method, **bucketer_args)
```

Performing Bucketing for both the train and test data.

[15]:
```
bucket_assignments_train = bucketer.fit_predict(dt_train_prefixes)

bucket_assignments_test = bucketer.predict(dt_test_prefixes)
```

[16]:
```
"""Caching the results for AUC score"""

preds_all = []
test_y_all = []
nr_events_all = []
```

0.5.3 Iterating over every bucket to perform index encoding and training XGBoost

In the following code, we build our data processing pipeline. First we iterate over each buacket. We perform index encoding using the `EncoderFactory` class. We then train the classifier for the buacket using XGBoost. The XGBoost parameters are optimized using the "optimize_params.py" module, as we have 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`

[17]:
```
""" ************* Perfroming Index Encoding per bucket*****************"""
for bucket in set(bucket_assignments_test):
    if bucket_method == "prefix":
        current_args = args[bucket]
    else:
        current_args = args
    relevant_train_cases_bucket = dataset_manager.
 ↪get_indexes(dt_train_prefixes)[
        bucket_assignments_train == bucket]
    relevant_test_cases_bucket = dataset_manager.get_indexes(dt_test_prefixes)[
        bucket_assignments_test == bucket]
    dt_test_bucket = dataset_manager.
 ↪get_relevant_data_by_indexes(dt_test_prefixes, relevant_test_cases_bucket)

    nr_events_all.extend(list(dataset_manager.
 ↪get_prefix_lengths(dt_test_bucket)))
    if len(relevant_train_cases_bucket) == 0:
        preds = [dataset_manager.get_class_ratio(train)] *␣
 ↪len(relevant_test_cases_bucket)

    else:
        dt_train_bucket = dataset_manager.
 ↪get_relevant_data_by_indexes(dt_train_prefixes,                    ␣
 ↪relevant_train_cases_bucket)  # one row per event
        train_y = dataset_manager.get_label_numeric(dt_train_bucket)

        if len(set(train_y)) < 2:
            preds = [train_y[0]] * len(relevant_test_cases_bucket)

            test_y_all.extend(dataset_manager.get_label_numeric(dt_test_bucket))
        else:

            feature_combiner = FeatureUnion(
                [(method, EncoderFactory.get_encoder(method,␣
 ↪**cls_encoder_args)) for method in methods])


            cls = xgb.XGBClassifier(objective='binary:logistic',
                                    n_estimators=500,
                                    learning_rate=current_args['learning_rate'],
                                    subsample=current_args['subsample'],
                                    max_depth=int(current_args['max_depth']),
                                             ␣
↪colsample_bytree=current_args['colsample_bytree'],
                                             ␣
↪min_child_weight=int(current_args['min_child_weight']),
                                  seed=random_state)

            pipeline = Pipeline([('encoder', feature_combiner), ('cls', cls)])

            pipeline.fit(dt_train_bucket, train_y)


            # predict separately for each prefix case
            preds = []
            test_all_grouped = dt_test_bucket.groupby(dataset_manager.
↪case_id_col)
            for _, group in test_all_grouped:

                test_y_all.extend(dataset_manager.get_label_numeric(group))


                _ = bucketer.predict(group)

                preds_pos_label_idx = np.where(cls.classes_ == 1)[0][0]
                pred = pipeline.predict_proba(group)[:, preds_pos_label_idx]


                preds.extend(pred)

    preds_all.extend(preds)
```

```
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 constructing 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)
```

```
[10:38:05] WARNING: C:/Users/Administrator/workspace/xgboost-
win64_release_1.4.0/src/learner.cc:1095: Starting in XGBoost 1.3.0, the default
evaluation metric used with the objective 'binary:logistic' was changed from
'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the
old behavior.
```

```
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 constructing 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)
```

```
[10:39:32] WARNING: C:/Users/Administrator/workspace/xgboost-
win64_release_1.4.0/src/learner.cc:1095: Starting in XGBoost 1.3.0, the default
evaluation metric used with the objective 'binary:logistic' was changed from
'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the
old behavior.
```

0.5.4 Evaluation

We evaluate the trained model using the ROC AUC score as follows:

[18]:
```
dt_results = pd.DataFrame({"actual": test_y_all, "predicted": preds_all,␣
 ↪"nr_events": nr_events_all})

print("The AUC is: %s\n" % (roc_auc_score(dt_results.actual, dt_results.
 ↪predicted)))
```

```
The AUC is: 0.9328061413244543
```