05_1_TimeSeries_HistoricalEvents

January 28, 2021

[ ]:
```
# The MIT License (MIT)

# Copyright (c) 2020, NVIDIA CORPORATION.

# Permission is hereby granted, free of charge, to any person obtaining a copy␣
↪of
# this software and associated documentation files (the "Software"), to deal in
# the Software without restriction, including without limitation the rights to
# use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies␣
↪of
# the Software, and to permit persons to whom the Software is furnished to do␣
↪so,
# subject to the following conditions:

# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,␣
↪FITNESS
# FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
# COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
# IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE
```

1 Tutorial: Feature Engineering for Recommender Systems

2 5. Feature Engineering - TimeSeries

2.1 5.1. Historical Events

[1]:
```
import IPython

import pandas as pd
import numpy as np

import cudf
```

```python
import cupy

np.random.seed(42)
```

[2]:
```python
itemid = [1000001]*10 + [1000002]*5 + [1000001]*5 + [1000002]*5 + [1000001]*1 +␣
↪[1000002]*1 + [1000001]*2 + [1000002]*2
itemid += [1000001]*3 + [1000002]*2 + [1000001]*1 + [1000002]*1 + [1000001]*6 +␣
↪[1000002]*3 + [1000001]*2 + [1000002]*2
userid = np.random.choice(list(range(10000)), len(itemid))
action = np.random.choice(list(range(2)), len(itemid), p=[0.2, 0.8])
timestamp = [pd.to_datetime('2020-01-01')]*15
timestamp += [pd.to_datetime('2020-01-02')]*10
timestamp += [pd.to_datetime('2020-01-03')]*2
timestamp += [pd.to_datetime('2020-01-04')]*4
timestamp += [pd.to_datetime('2020-01-05')]*5
timestamp += [pd.to_datetime('2020-01-07')]*2
timestamp += [pd.to_datetime('2020-01-08')]*9
timestamp += [pd.to_datetime('2020-01-09')]*4

data = pd.DataFrame({
    'itemid': itemid,
    'userid': userid,
    'action': action,
    'timestamp': timestamp
})
```

[3]:
```python
data = cudf.from_pandas(data)
```

2.2 Theory

Many real-world recommendation systems contain time information. The system normally logs events with a timestamp. Tree-based or deep learning based models usually only uses the information from the datapoint itself for the prediction and they have difficulties to capture relationships over multiple datapoints.

Let's take a look at a simple example. Let's assume we have the interaction events of an itemid, userid and action with the timestamp.

[4]:
```python
data[data['itemid']==1000001]
```

[4]:

	itemid	userid	action	timestamp
0	1000001	7270	1	2020-01-01
1	1000001	860	1	2020-01-01
2	1000001	5390	0	2020-01-01
3	1000001	5191	1	2020-01-01
4	1000001	5734	0	2020-01-01
5	1000001	6265	1	2020-01-01

```
6    1000001     466          1  2020-01-01
7    1000001    4426          1  2020-01-01
8    1000001    5578          1  2020-01-01
9    1000001    8322          0  2020-01-01
15   1000001    5051          1  2020-01-02
16   1000001    6420          1  2020-01-02
17   1000001    1184          1  2020-01-02
18   1000001    4555          1  2020-01-02
19   1000001    3385          1  2020-01-02
25   1000001    2047          1  2020-01-03
27   1000001    9167          0  2020-01-04
28   1000001    9998          0  2020-01-04
31   1000001    3005          1  2020-01-05
32   1000001    4658          0  2020-01-05
33   1000001    1899          0  2020-01-05
36   1000001    1528          1  2020-01-07
38   1000001    3890          1  2020-01-08
39   1000001    8838          1  2020-01-08
40   1000001    5393          1  2020-01-08
41   1000001    8792          1  2020-01-08
42   1000001    8433          0  2020-01-08
43   1000001    7513          1  2020-01-08
47   1000001    6235          1  2020-01-09
48   1000001    5486          1  2020-01-09
```

We can extract many interesting features based on the history, such as * the sum number of actions of the last day, last 3 days or last 7 days * the average number of actions of the last day, last 3 days or last 7 days * the average probability of the last day, last 3 days or last 7 days * etc.

In general, these operations are called window function and uses `.rolling()` function. For each row, the function looks at a window (# of rows around it) and apply a certain function to it.

Current, our data is on a userid and itemid level. First, we need to aggregate it on the level, we want to apply the window function.

[5]:
```
data_window = data[['itemid', 'timestamp', 'action']].groupby(['itemid',␣
↪'timestamp']).agg(['count', 'sum']).reset_index()
data_window.columns = ['itemid', 'timestamp', 'count', 'sum']
data_window.index = data_window['timestamp']
```

[6]:
```
data_window
```

[6]:
```
              itemid  timestamp  count  sum
timestamp
2020-01-01   1000001 2020-01-01     10    7
2020-01-02   1000001 2020-01-02      5    5
2020-01-03   1000001 2020-01-03      1    1
2020-01-04   1000001 2020-01-04      2    0
```

```
2020-01-05  1000001 2020-01-05      3    1
2020-01-07  1000001 2020-01-07      1    1
2020-01-08  1000001 2020-01-08      6    5
2020-01-09  1000001 2020-01-09      2    2
2020-01-01  1000002 2020-01-01      5    5
2020-01-02  1000002 2020-01-02      5    5
2020-01-03  1000002 2020-01-03      1    1
2020-01-04  1000002 2020-01-04      2    2
2020-01-05  1000002 2020-01-05      2    2
2020-01-07  1000002 2020-01-07      1    1
2020-01-08  1000002 2020-01-08      3    3
2020-01-09  1000002 2020-01-09      2    2
```

We are interested how many positive interaction an item had on the previous day. Next, we want to groupby our dataframe by itemid. Then we apply the rolling function for two days (`2D`).

Note: To use the rolling function with days, the dataframe index has to by a timestamp.

We can see that every row contains the sum of the row value + the previous row value. For example, `itemid=1000001` for data `2020-01-02` counts 15 observations and sums 12 positive interactions. What happend on the date `2020-01-07`?

[7]:
```
offset = '3D'

data_window_roll = data_window[['itemid', 'count', 'sum']].groupby(['itemid']).
↪rolling(offset).sum().drop('itemid', axis=1)
data_window_roll
```

[7]:
```
                    count  sum
itemid  timestamp
1000001 2020-01-01     10    7
        2020-01-02     15   12
        2020-01-03     16   13
        2020-01-04      8    6
        2020-01-05      6    2
        2020-01-07      4    2
        2020-01-08      7    6
        2020-01-09      9    8
1000002 2020-01-01      5    5
        2020-01-02     10   10
        2020-01-03     11   11
        2020-01-04      8    8
        2020-01-05      5    5
        2020-01-07      3    3
        2020-01-08      4    4
        2020-01-09      6    6
```

If we take a look on the calculations, we see that the `.rolling()` inclues the value from the current row, as well. This could be a kind of data leakage. Therefore, we shift the values by one row.

[8]:
```
data_window_roll = data_window_roll.reset_index()
data_window_roll.columns = ['itemid', 'timestamp', 'count_' + offset, 'sum_' +␣
 ↪offset]
data_window_roll[['count_' + offset, 'sum_' + offset]] =␣
 ↪data_window_roll[['count_' + offset, 'sum_' + offset]].shift(1)
data_window_roll.loc[data_window_roll['itemid']!=data_window_roll['itemid'].
 ↪shift(1), ['count_' + offset, 'sum_' + offset]] = 0
data_window_roll['avg_' + offset] = data_window_roll['sum_' + offset]/
 ↪data_window_roll['count_' + offset]
```

[9]:
```
data_window_roll
```

[9]:

	itemid	timestamp	count_3D	sum_3D	avg_3D
0	1000001	2020-01-01	0	0	NaN
1	1000001	2020-01-02	10	7	0.700000
2	1000001	2020-01-03	15	12	0.800000
3	1000001	2020-01-04	16	13	0.812500
4	1000001	2020-01-05	8	6	0.750000
5	1000001	2020-01-07	6	2	0.333333
6	1000001	2020-01-08	4	2	0.500000
7	1000001	2020-01-09	7	6	0.857143
8	1000002	2020-01-01	0	0	NaN
9	1000002	2020-01-02	5	5	1.000000
10	1000002	2020-01-03	10	10	1.000000
11	1000002	2020-01-04	11	11	1.000000
12	1000002	2020-01-05	8	8	1.000000
13	1000002	2020-01-07	5	5	1.000000
14	1000002	2020-01-08	3	3	1.000000
15	1000002	2020-01-09	4	4	1.000000

After we calculated the aggregated values and applied the window function, we want to merge it to our original dataframe.

[10]:
```
data = data.merge(data_window_roll, how='left', on=['itemid', 'timestamp'])
```

[11]:
```
data
```

[11]:

	itemid	userid	action	timestamp	count_3D	sum_3D	avg_3D
0	1000001	4658	0	2020-01-05	8	6	0.750000
1	1000001	1899	0	2020-01-05	8	6	0.750000
2	1000002	7734	1	2020-01-05	8	8	1.000000
3	1000002	1267	1	2020-01-05	8	8	1.000000
4	1000001	1528	1	2020-01-07	6	2	0.333333
5	1000002	3556	1	2020-01-07	5	5	1.000000
6	1000001	3890	1	2020-01-08	4	2	0.500000
7	1000001	8838	1	2020-01-08	4	2	0.500000
8	1000001	5393	1	2020-01-08	4	2	0.500000

9	1000001	8792	1	2020-01-08	4	2	0.500000
10	1000001	8433	0	2020-01-08	4	2	0.500000
11	1000001	7513	1	2020-01-08	4	2	0.500000
12	1000002	2612	1	2020-01-08	3	3	1.000000
13	1000002	7041	1	2020-01-08	3	3	1.000000
14	1000002	9555	1	2020-01-08	3	3	1.000000
15	1000001	6235	1	2020-01-09	7	6	0.857143
16	1000001	5486	1	2020-01-09	7	6	0.857143
17	1000002	7099	1	2020-01-09	4	4	1.000000
18	1000002	9670	1	2020-01-09	4	4	1.000000
19	1000001	7270	1	2020-01-01	0	0	NaN
20	1000001	860	1	2020-01-01	0	0	NaN
21	1000001	5390	0	2020-01-01	0	0	NaN
22	1000001	5191	1	2020-01-01	0	0	NaN
23	1000001	5734	0	2020-01-01	0	0	NaN
24	1000001	6265	1	2020-01-01	0	0	NaN
25	1000001	466	1	2020-01-01	0	0	NaN
26	1000001	4426	1	2020-01-01	0	0	NaN
27	1000001	5578	1	2020-01-01	0	0	NaN
28	1000001	8322	0	2020-01-01	0	0	NaN
29	1000002	1685	1	2020-01-01	0	0	NaN
30	1000002	769	1	2020-01-01	0	0	NaN
31	1000002	6949	1	2020-01-01	0	0	NaN
32	1000002	2433	1	2020-01-01	0	0	NaN
33	1000002	5311	1	2020-01-01	0	0	NaN
34	1000001	5051	1	2020-01-02	10	7	0.700000
35	1000001	6420	1	2020-01-02	10	7	0.700000
36	1000001	1184	1	2020-01-02	10	7	0.700000
37	1000001	4555	1	2020-01-02	10	7	0.700000
38	1000001	3385	1	2020-01-02	10	7	0.700000
39	1000002	6396	1	2020-01-02	5	5	1.000000
40	1000002	8666	1	2020-01-02	5	5	1.000000
41	1000002	9274	1	2020-01-02	5	5	1.000000
42	1000002	2558	1	2020-01-02	5	5	1.000000
43	1000002	7849	1	2020-01-02	5	5	1.000000
44	1000001	2047	1	2020-01-03	15	12	0.800000
45	1000002	2747	1	2020-01-03	10	10	1.000000
46	1000001	9167	0	2020-01-04	16	13	0.812500
47	1000001	9998	0	2020-01-04	16	13	0.812500
48	1000002	189	1	2020-01-04	11	11	1.000000
49	1000002	2734	1	2020-01-04	11	11	1.000000
50	1000001	3005	1	2020-01-05	8	6	0.750000

We can apply the same technique for the last 7 days.

[12]:
```
offset = '7D'
```

```python
data_window_roll = data_window[['itemid', 'count', 'sum']].groupby(['itemid']).
↪rolling(offset).sum().drop('itemid', axis=1)
data_window_roll = data_window_roll.reset_index()
data_window_roll.columns = ['itemid', 'timestamp', 'count_' + offset, 'sum_' +␣
↪offset]
data_window_roll[['count_' + offset, 'sum_' + offset]] =␣
↪data_window_roll[['count_' + offset, 'sum_' + offset]].shift(1)
data_window_roll.loc[data_window_roll['itemid']!=data_window_roll['itemid'].
↪shift(1), ['count_' + offset, 'sum_' + offset]] = 0
data_window_roll['avg_' + offset] = data_window_roll['sum_' + offset]/
↪data_window_roll['count_' + offset]
data = data.merge(data_window_roll, how='left', on=['itemid', 'timestamp'])
data
```

[12]:

	itemid	userid	action	timestamp	count_3D	sum_3D	avg_3D	count_7D	\
0	1000001	4658	0	2020-01-05	8	6	0.750000	18	
1	1000001	1899	0	2020-01-05	8	6	0.750000	18	
2	1000002	7734	1	2020-01-05	8	8	1.000000	13	
3	1000002	1267	1	2020-01-05	8	8	1.000000	13	
4	1000001	1528	1	2020-01-07	6	2	0.333333	21	
5	1000002	3556	1	2020-01-07	5	5	1.000000	15	
6	1000001	3890	1	2020-01-08	4	2	0.500000	22	
7	1000001	8838	1	2020-01-08	4	2	0.500000	22	
8	1000001	5393	1	2020-01-08	4	2	0.500000	22	
9	1000001	8792	1	2020-01-08	4	2	0.500000	22	
10	1000001	8433	0	2020-01-08	4	2	0.500000	22	
11	1000001	7513	1	2020-01-08	4	2	0.500000	22	
12	1000002	2612	1	2020-01-08	3	3	1.000000	16	
13	1000002	7041	1	2020-01-08	3	3	1.000000	16	
14	1000002	9555	1	2020-01-08	3	3	1.000000	16	
15	1000001	6235	1	2020-01-09	7	6	0.857143	18	
16	1000001	5486	1	2020-01-09	7	6	0.857143	18	
17	1000002	7099	1	2020-01-09	4	4	1.000000	14	
18	1000002	9670	1	2020-01-09	4	4	1.000000	14	
19	1000001	7270	1	2020-01-01	0	0	NaN	0	
20	1000001	860	1	2020-01-01	0	0	NaN	0	
21	1000001	5390	0	2020-01-01	0	0	NaN	0	
22	1000001	5191	1	2020-01-01	0	0	NaN	0	
23	1000001	5734	0	2020-01-01	0	0	NaN	0	
24	1000001	6265	1	2020-01-01	0	0	NaN	0	
25	1000001	466	1	2020-01-01	0	0	NaN	0	
26	1000001	4426	1	2020-01-01	0	0	NaN	0	
27	1000001	5578	1	2020-01-01	0	0	NaN	0	
28	1000001	8322	0	2020-01-01	0	0	NaN	0	
29	1000002	1685	1	2020-01-01	0	0	NaN	0	
30	1000002	769	1	2020-01-01	0	0	NaN	0	
31	1000002	6949	1	2020-01-01	0	0	NaN	0	

32	1000002	2433	1	2020-01-01	0	0	NaN	0
33	1000002	5311	1	2020-01-01	0	0	NaN	0
34	1000001	5051	1	2020-01-02	10	7	0.700000	10
35	1000001	6420	1	2020-01-02	10	7	0.700000	10
36	1000001	1184	1	2020-01-02	10	7	0.700000	10
37	1000001	4555	1	2020-01-02	10	7	0.700000	10
38	1000001	3385	1	2020-01-02	10	7	0.700000	10
39	1000002	6396	1	2020-01-02	5	5	1.000000	5
40	1000002	8666	1	2020-01-02	5	5	1.000000	5
41	1000002	9274	1	2020-01-02	5	5	1.000000	5
42	1000002	2558	1	2020-01-02	5	5	1.000000	5
43	1000002	7849	1	2020-01-02	5	5	1.000000	5
44	1000001	2047	1	2020-01-03	15	12	0.800000	15
45	1000002	2747	1	2020-01-03	10	10	1.000000	10
46	1000001	9167	0	2020-01-04	16	13	0.812500	16
47	1000001	9998	0	2020-01-04	16	13	0.812500	16
48	1000002	189	1	2020-01-04	11	11	1.000000	11
49	1000002	2734	1	2020-01-04	11	11	1.000000	11
50	1000001	3005	1	2020-01-05	8	6	0.750000	18

	sum_7D	avg_7D
0	13	0.722222
1	13	0.722222
2	13	1.000000
3	13	1.000000
4	14	0.666667
5	15	1.000000
6	15	0.681818
7	15	0.681818
8	15	0.681818
9	15	0.681818
10	15	0.681818
11	15	0.681818
12	16	1.000000
13	16	1.000000
14	16	1.000000
15	13	0.722222
16	13	0.722222
17	14	1.000000
18	14	1.000000
19	0	NaN
20	0	NaN
21	0	NaN
22	0	NaN
23	0	NaN
24	0	NaN
25	0	NaN

```
26       0       NaN
27       0       NaN
28       0       NaN
29       0       NaN
30       0       NaN
31       0       NaN
32       0       NaN
33       0       NaN
34       7  0.700000
35       7  0.700000
36       7  0.700000
37       7  0.700000
38       7  0.700000
39       5  1.000000
40       5  1.000000
41       5  1.000000
42       5  1.000000
43       5  1.000000
44      12  0.800000
45      10  1.000000
46      13  0.812500
47      13  0.812500
48      11  1.000000
49      11  1.000000
50      13  0.722222
```

2.3 Practice

[13]:
```
### loading
import pandas as pd
import cudf
import numpy as np
import cupy
import matplotlib.pyplot as plt

df_train = cudf.read_parquet('./data/train.parquet')
df_valid = cudf.read_parquet('./data/valid.parquet')
df_test = cudf.read_parquet('./data/test.parquet')

df_train['brand'] = df_train['brand'].fillna('UNKNOWN')
df_valid['brand'] = df_valid['brand'].fillna('UNKNOWN')
df_test['brand'] = df_test['brand'].fillna('UNKNOWN')

df_train['cat_0'] = df_train['cat_0'].fillna('UNKNOWN')
df_valid['cat_0'] = df_valid['cat_0'].fillna('UNKNOWN')
df_test['cat_0'] = df_test['cat_0'].fillna('UNKNOWN')
```

```
df_train['cat_1'] = df_train['cat_1'].fillna('UNKNOWN')
df_valid['cat_1'] = df_valid['cat_1'].fillna('UNKNOWN')
df_test['cat_1'] = df_test['cat_1'].fillna('UNKNOWN')

df_train['cat_2'] = df_train['cat_2'].fillna('UNKNOWN')
df_valid['cat_2'] = df_valid['cat_2'].fillna('UNKNOWN')
df_test['cat_2'] = df_test['cat_2'].fillna('UNKNOWN')
```

cuDF does not support date32, right now. We use pandas to transform the timestamp in only date values.

[14]:
```
df_train['date'] = cudf.from_pandas(pd.to_datetime(df_train['timestamp'].
↪to_pandas()).dt.date)
```

```
/conda/envs/nvtabular/lib/python3.7/site-
packages/cudf/core/column/column.py:1396: UserWarning: Date32 values are not yet
supported so this will be typecast to a Date64 value
  UserWarning,
```

Let's get the # of purchases per product in the 7 days before.

ToDo:

Calculate the # of purchases of an item of the 7 previous days for each datapoint

2.4 Optimisation

Let's compare a CPU with the GPU version.

[17]:
```
def rolling_window(df, col, offset):
    data_window = df[[col, 'date', 'target']].groupby([col, 'date']).
↪agg(['count', 'sum']).reset_index()
    data_window.columns = [col, 'date', 'count', 'sum']
    data_window.index = data_window['date']

    data_window_roll = data_window[[col, 'count', 'sum']].groupby([col]).
↪rolling(offset).sum().drop(col, axis=1)
    data_window_roll = data_window_roll.reset_index()
    data_window_roll.columns = [col, 'date', 'count_' + offset, 'sum_' + offset]
    data_window_roll[['count_' + offset, 'sum_' + offset]] =␣
↪data_window_roll[['count_' + offset, 'sum_' + offset]].shift(1)
    data_window_roll.loc[data_window_roll[col]!=data_window_roll[col].shift(1),␣
↪['count_' + offset, 'sum_' + offset]] = 0
    data_window_roll['avg_' + offset] = data_window_roll['sum_' + offset]/
↪data_window_roll['count_' + offset]
    data = df.merge(data_window_roll, how='left', on=[col, 'date'])
    return(data)
```

[18]:
```
df_train_pd = df_train.to_pandas()
```

[19]:
```
%%time

_ = rolling_window(df_train_pd, 'product_id', '5D')
```

```
CPU times: user 37.5 s, sys: 5.04 s, total: 42.5 s
Wall time: 42.5 s
```

[20]:
```
%%time

_ = rolling_window(df_train, 'product_id', '5D')
```

```
CPU times: user 424 ms, sys: 232 ms, total: 656 ms
Wall time: 655 ms
```

In our experiments, we achieved a speedup of 372x

We shutdown the kernel.

[21]:
```
app = IPython.Application.instance()
app.kernel.do_shutdown(False)
```

[21]:
```
{'status': 'ok', 'restart': False}
```