# 03_2_Categorify

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 3. Feature Engineering - Categorical

### 2.1 3.2. Categorify

[1]:
```
import IPython

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

```python
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')
```

[2]:
```python
df_train.head()
```

[2]:
```
                event_time event_type  product_id    brand   price    user_id  \
0  2019-12-01 00:00:28 UTC       cart    17800342     zeta   66.90  550465671
1  2019-12-01 00:00:39 UTC       cart     3701309  polaris   89.32  543733099
2  2019-12-01 00:00:40 UTC       cart     3701309  polaris   89.32  543733099
3  2019-12-01 00:00:41 UTC       cart     3701309  polaris   89.32  543733099
4  2019-12-01 00:01:56 UTC       cart     1004767  samsung  235.60  579970209

                           user_session  target        cat_0        cat_1  \
0  22650a62-2d9c-4151-9f41-2674ec6d32d5       0    computers      desktop
1  a65116f4-ac53-4a41-ad68-6606788e674c       0   appliances  environment
2  a65116f4-ac53-4a41-ad68-6606788e674c       0   appliances  environment
3  a65116f4-ac53-4a41-ad68-6606788e674c       0   appliances  environment
4  c6946211-ce70-4228-95ce-fd7fccdde63c       0 construction        tools

    cat_2 cat_3           timestamp  ts_hour  ts_minute  ts_weekday  ts_day  \
0    <NA>  <NA> 2019-12-01 00:00:28        0          0           6       1
1  vacuum  <NA> 2019-12-01 00:00:39        0          0           6       1
2  vacuum  <NA> 2019-12-01 00:00:40        0          0           6       1
3  vacuum  <NA> 2019-12-01 00:00:41        0          0           6       1
4   light  <NA> 2019-12-01 00:01:56        0          1           6       1

   ts_month  ts_year
0        12     2019
1        12     2019
2        12     2019
3        12     2019
4        12     2019
```

[3]:
```python
cat = 'product_id'
```

## 2.2 Theory

*Categorifying* is required for using categorical features in deep learning models with Embedding layers. An Embedding layer encodes the category into a hidden latent vector with a smaller dimension. Categorical features can be from datatype String or Integer. The Embedding layer requires that categorical features are continoues, positive Integers from 0 to |C| (number of unique category values).

There are 164453 unique product values but the ProductIDs range from 1000894 to 100144608.

[4]:
```
df_train[cat].unique()
```

[4]:
```
0            1000894
1            1000978
2            1001588
3            1001605
4            1001606
             …
164448     100143856
164449     100143867
164450     100144046
164451     100144443
164452     100144608
Name: product_id, Length: 164453, dtype: int64
```

Using factorize creates continous Integers from a categorical fateature.

[5]:
```
codes, uniques = df_train[cat].factorize()
```

[6]:
```
codes
```

[6]:
```
0            65426
1            10158
2            10158
3            10158
4              775
             …
11461352       862
11461353      1064
11461354       775
11461355     10158
11461356     91487
Length: 11461357, dtype: int32
```

[7]:
```
codes.unique()
```

[7]:
```
0                 0
1                 1
2                 2
3                 3
4                 4
            …
164448     164448
164449     164449
164450     164450
164451     164451
164452     164452
```

```
Length: 164453, dtype: int32
```

Another important reason to Categorify categorical features is to reduce the size of the dataset. Often categorical features are of the datatype String and sometimes, they are hashed to protect the user / dataset privacy.

[8]:
```
import hashlib
from sys import getsizeof
```

For example, we can hash the Integer 0 to a md5 hash

[9]:
```
hashlib.md5(b'0').hexdigest()
```

[9]:
```
'cfcd208495d565ef66e7dff9f98764da'
```

We can hash the full product_id column

[10]:
```
hashSeries = df_train[cat].to_pandas().apply(lambda x: hashlib.
 ↪md5(bytes(str(x), encoding='utf-8')).hexdigest())
```

[11]:
```
hashSeries
```

[11]:
```
0           5ebc4b45850c48658af86229318ccbea
1           4b9dde859aa2809cc367fc44aa05eb4a
2           4b9dde859aa2809cc367fc44aa05eb4a
3           4b9dde859aa2809cc367fc44aa05eb4a
4           e5e26a76d8aee9c4f8b4cd9cb8633577
                          …
11461352    4202ee67e0c3f9b1ebcfb622d9974e07
11461353    5a06406ed78aab3c94bfefcdeb528eaf
11461354    e5e26a76d8aee9c4f8b4cd9cb8633577
11461355    4b9dde859aa2809cc367fc44aa05eb4a
11461356    388318441a4807e254acf9c3f207969d
Name: product_id, Length: 11461357, dtype: object
```

[12]:
```
getsizeof(hashSeries)
```

[12]:
```
1020060933
```

[13]:
```
codes, uniques = hashSeries.factorize()
```

[14]:
```
getsizeof(pd.DataFrame(codes)[0])
```

[14]:
```
91691016
```

We require only 9% of the original DataSeries memory.

[15]:
```
91691016/1020060933
```

[15]: 0.08988778320363339

Finally, we can prevent overfitting for low frequency categories. Categories with low frequency can be grouped together to an new category called 'other'. In the previous exercise we learned that it is powerful to combine categorical features together to create a new feature. However, combining categories increases the cardinality of the new feature and the number of obersations per category will decrease. Therefore, we can apply a treshhold to group all categories with lower frequency count to the the new category. In addition, categories, which occure in the validation dataset and do not occur in the trainint dataset, should be mapped to the 'other' category as well. We use in our example the categoryIds 0 or 1 for a placeholder for the low frequency and unkown category. Then our function is independent of the cardinality of the categorical feature and we do not keep records of the cardinality to know the low frequency/unkown category.

In our dataset, we see that multiple product_ids occure only once in the training dataset. Our model would overfit to these low frequent categories.

[16]:
```
df_train[cat].value_counts()
```

[16]:
```
1004767      317711
1005115      251189
1004856      227432
4804056      224545
1005100      180072
              …
100143590         1
100143856         1
100143867         1
100144046         1
100144443         1
Name: product_id, Length: 164453, dtype: int32
```

[17]:
```
freq = df_train[cat].value_counts()
```

[18]:
```
freq = freq.reset_index()
freq.columns = [cat, 'count']
freq = freq.reset_index()
freq.columns = [cat + '_Categorify', cat, 'count']
freq_filtered = freq[freq['count']>5]
freq_filtered[cat + '_Categorify'] = freq_filtered[cat + '_Categorify']+1
freq_filtered = freq_filtered.drop('count', axis=1)
df_train = df_train.merge(freq_filtered, how='left', on=cat)
df_train[cat + '_Categorify'] = df_train[cat + '_Categorify'].fillna(0)
```

[19]:
```
df_train['product_id_Categorify'].min(), df_train['product_id_Categorify'].
 ↪max(), df_train['product_id_Categorify'].drop_duplicates().shape
```

[19]:
```
(0, 76404, (76405,))
```

We need to apply the categorify to our validation and test sets.

[20]:
```
df_valid = df_valid.merge(freq_filtered, how='left', on=cat)
df_valid[cat + '_Categorify'] = df_valid[cat + '_Categorify'].fillna(0)

df_test = df_test.merge(freq_filtered, how='left', on=cat)
df_test[cat + '_Categorify'] = df_test[cat + '_Categorify'].fillna(0)
```

**Summary** Categorify is important to enable deep learning models to use categorical features

Categorify can significantly reduce the dataset size by tranforming categorical features from String datatypes to Integer datatypes

Categorify can prevent overfitting by grouping categories with low frequency into one category together

### 2.3 Practice

Now, it is your turn

**ToDo:**

Categorify the category features brand

Apply a frequency treshhold of minimum 20

Map low frequency categories to the id=0

Map unkown categories to the id=1 in the validation and test set

**Question:**

How many data points have an unknown category in the test dataset?

How many data points have a low frequency category in the test dataset?

How many data points have a low frequency category in the training dataset?

[21]:
```
### ToDo
```

[22]:
```
############### Solution ###############
```

[27]:
```
############### Solution End ###########
```

### 2.4 Optimization

Let's compare the runtime between pandas and cuDF. The implementation depends only on the DataFrame object (calling function of the object) and does not require any pd / cuDF function. Therefore, we can use the same implementation and just use pandas.DataFrame and cuDF.DataFrame.

[28]:
```

```

```
def categorify(df_train, df_valid, df_test, cat, freq_treshhold=20,␣
↪unkown_id=1, lowfrequency_id=0):
    freq = df_train[cat].value_counts()
    freq = freq.reset_index()
    freq.columns = [cat, 'count']
    freq = freq.reset_index()
    freq.columns = [cat + '_Categorify', cat, 'count']
    freq[cat + '_Categorify'] = freq[cat + '_Categorify']+2
    freq.loc[freq['count']<freq_treshhold, cat + '_Categorify'] =␣
↪lowfrequency_id

    freq = freq.drop('count', axis=1)
    df_train = df_train.merge(freq, how='left', on=cat)
    df_train[cat + '_Categorify'] = df_train[cat + '_Categorify'].
↪fillna(unkown_id)

    df_valid = df_valid.merge(freq, how='left', on=cat)
    df_valid[cat + '_Categorify'] = df_valid[cat + '_Categorify'].
↪fillna(unkown_id)

    df_test = df_test.merge(freq, how='left', on=cat)
    df_test[cat + '_Categorify'] = df_test[cat + '_Categorify'].
↪fillna(unkown_id)
```

[29]:
```
df_train_pd = df_train.to_pandas()
df_valid_pd = df_valid.to_pandas()
df_test_pd = df_test.to_pandas()
```

[30]:
```
%%time

categorify(df_train_pd, df_valid_pd, df_test_pd, 'user_id')
```

```
CPU times: user 15.5 s, sys: 5.77 s, total: 21.3 s
Wall time: 21.2 s
```

[31]:
```
%%time

categorify(df_train, df_valid, df_test, 'user_id')
```

```
CPU times: user 168 ms, sys: 296 ms, total: 464 ms
Wall time: 463 ms
```

In our experiments, running the same implementation is 63x times faster with cuDF instead of pandas.

We shutdown the kernel.

[32]:
```python
app = IPython.Application.instance()
app.kernel.do_shutdown(False)
```

[32]: {'status': 'ok', 'restart': False}