LTFS Data Science FinHack 3 Approach

By: mdalvi (milind.dalvi14@gmail.com) / https://www.linkedin.com/in/milinddalvi (https://www.linkedin.com/in/milinddalvi) / +91-7738624226

Problem Statement

LTFS Top-up loan Up-sell prediction (Top-up Month)

A loan is when you receive the money from a financial institution in exchange for future repayment of the principal, plus interest. Financial institutions provide loans to the industries, corporates and individuals. The interest received on these loans is one among the main sources of income for the financial institutions.

A top-up loan, true to its name, is a facility of availing further funds on an existing loan. When you have a loan that has already been disbursed and under repayment and if you need more funds then, you can simply avail additional funding on the same loan thereby minimizing time, effort and cost related to applying again.

LTFS provides it’s loan services to its customers and is interested in selling more of its Top-up loan services to its existing customers so they have decided to identify when to pitch a Top-up during the original loan tenure. If they correctly identify the most suitable time to offer a top-up, this will ultimately lead to more disbursals and can also help them beat competing offerings from other institutions.

To understand this behaviour, LTFS has provided data for its customers containing the information whether that particular customer took the Top-up service and when he took such Top-up service, represented by the target variable Top-up Month.

System Requirements

The solution file is devloped and tested on **Free GPU Notebooks** provided by https://gradient.paperspace.com (https://gradient.paperspace.com) with following configuration,

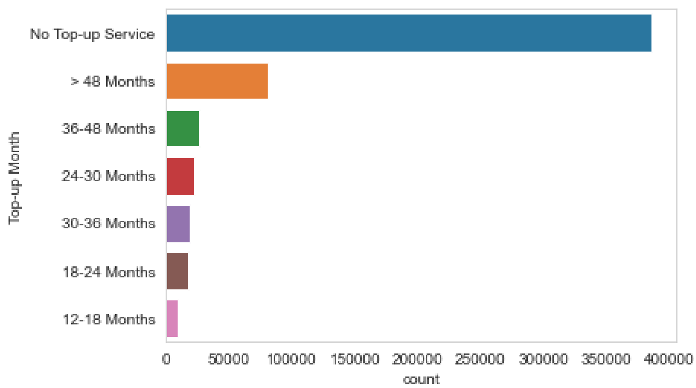
- CPU Cores: 8
- RAM: 30GB
- GPU: NVIDIA M4000 GPU

Plotting Target Distribution

The provided dataset had seven imbalanced categories (multi-class classification problem)

```
plot_countplot(df['Top-up Month'])
```

```
+-------------------+---------+-------------+
| Top-up Month      |   Count |  Percentage |
|-------------------+---------+-------------|
| No Top-up Service |  385604 |       0.688 |
| > 48 Months       |   80114 |       0.143 |
| 36-48 Months      |   26613 |       0.047 |
| 24-30 Months      |   22511 |       0.04  |
| 30-36 Months      |   19044 |       0.034 |
| 18-24 Months      |   17766 |       0.032 |
| 12-18 Months      |    9192 |       0.016 |
+-------------------+---------+-------------+
```



Merging Datasets

Initially the provided datasets were merged with **bureau data** on the left and **ltfs data** on the right with a **"left join"** on **"ID"** column.

```
df = pd.concat([df_train_data, df_test_data], axis=0)
df_bureau = pd.concat([df_train_bureau, df_test_bureau], axis=0)

df.reset_index(drop=True, inplace=True)
df_bureau.reset_index(drop=True, inplace=True)
df = df_bureau.merge(df, how='left', on='ID')
```

```
print(df.shape)
```

```
(624863, 51)
```

The shape of the final dataset is observed to be (624863, 51)

Cleaning Data

1) Some date columns were not appropriately recognized hence the below opertation,

```
print("performing to_datetime...")
source_dt_cols = ['DisbursalDate', 'MaturityDAte', 'AuthDate']
source_dt_cols_seq = ['CLOSE-DT', 'DISBURSED-DT', 'DATE-REPORTED', 'LAST-PAYMENT-DATE']

for col in source_dt_cols + source_dt_cols_seq:
    df[col] = pd.to_datetime(df[col], errors='coerce')
```

2) **"MaturityDAte"** for certain records were missing, hence a median maturity was added w.r.t. **"DisbursalDate"**

```
df.loc[df['MaturityDAte'].isnull(), 'MaturityDAte'] = df['DisbursalDate'] + (df['MaturityDAte'] - df['DisbursalDate']).median()
```

3) **"ASSET_CLASS"** column had some values like "01", "1" and "2". Assumed them to coincide with exising majority classes, **"Standard"** and **"SubStandard"**

```
print("performing cleaning...")
df.loc[df['ASSET_CLASS'].isin(['1', '01']), 'ASSET_CLASS'] = 'Standard'
df.loc[df['ASSET_CLASS'] == '2', 'ASSET_CLASS'] = 'SubStandard'
```

4) Various amount related columns in bureau data had to be converted to **"float"** dtype.

```
df['DISBURSED-AMT/HIGH CREDIT'] = df['DISBURSED-AMT/HIGH CREDIT'].str.replace(',', '').astype(float)
df['CURRENT-BAL'] = df['CURRENT-BAL'].str.replace(',', '').astype(float)
df['OVERDUE-AMT'] = df['OVERDUE-AMT'].str.replace(',', '').astype(float)
df['CREDIT-LIMIT/SANC AMT']= df['CREDIT-LIMIT/SANC AMT'].str.replace(',', '').astype(float)
```

5) **City** names were spotted to be duplicates across multiple states hence a unique identification had to be created by merging with state names.

```
df['City'] = df['City'] +'-'+ df['State']
```

6) Missing categorical features were filled with default value **"NOT_AVAILABLE"**

```
source_cat_cols = ['Frequency', 'InstlmentMode', 'LoanStatus', 'PaymentMode', 'BranchID', 'Area',
                   'ManufacturerID', 'SupplierID', 'SEX', 'City', 'State', 'ZiPCODE']
source_cat_cols_seq = ['SELF-INDICATOR', 'MATCH-TYPE', 'ACCT-TYPE', 'CONTRIBUTOR-TYPE', 'OWNERSHIP-IND',
                       'ACCOUNT-STATUS', 'INSTALLMENT-FREQUENCY', 'ASSET_CLASS']

df[source_cat_cols] = df[source_cat_cols].fillna(DEFAULT_CATEGORICAL_FILL)
df[source_cat_cols_seq] = df[source_cat_cols_seq].fillna(DEFAULT_CATEGORICAL_FILL)
```

7) **"INSTALLMENT-AMT"** column had to be treated with regular expression to actully segregate installment and frequency.

```python
print('performing regex and definitions...')
def strip(value, remove_spaces=False):
    result = re.sub(r"\s+", "" if remove_spaces else " ", value)
    result = result.strip()
    return result

def installment_freq(value):
    result = re.search(r"[a-z]+", str(value).lower())
    if result:
        return strip(result.group(0))
    else:
        return np.nan

def installment_amt(value):
    result = re.search(r"[\d,]+", str(value).lower())
    if result:
        return strip(result.group(0))
    else:
        return np.nan

df['p_cat_seq_INSTALLMENT_FREQ'] = df['INSTALLMENT-AMT'].map(installment_freq)
df['INSTALLMENT_AMT_CLEAN'] = df['INSTALLMENT-AMT'].map(installment_amt).str.replace(',', '').astype(float)
df['ACTIVE_INSTALLMENT_AMT'] = df[['ACCOUNT-STATUS', 'INSTALLMENT_AMT_CLEAN']].apply(lambda r: r[1] if r[0] == 'Active' else 0.,
```

8) Monthly installment and Monthly installement of only **"Active"** accounts were then created based on interpreting the **"INSTALLMENT-AMT"** by frequency.

```python
def active_installment_amt_monthly(r):
    if r[1] == 'biweekly':
        return r[0] * 2
    elif r[1] == 'weekly':
        return r[0] * 4
    elif r[1] == 'quarterly':
        return r[0] / 3
    elif r[1] == 'annually':
        return r[0] / 12
    elif r[1] == 'bimonthly':
        return r[0] / 2
    elif r[1] == 'semi':
        return r[0] / 6
    else:
        return r[0]

df['p_con_seq_ACTIVE_MONTHLY_INSTALLMENT_AMT'] = df[['ACTIVE_INSTALLMENT_AMT', 'p_cat_seq_INSTALLMENT_FREQ']].apply(active_insta
df['p_con_seq_MONTHLY_INSTALLMENT_AMT'] = df[['ACTIVE_INSTALLMENT_AMT', 'p_cat_seq_INSTALLMENT_FREQ']].apply(active_installment_a
```

9) In ltfs dataset some "EMI" values were found to be misleading, such that even after multiplying with "Tenure (in months)" the amount won't add-up to **"DisbursalAmount"** hence assumed corrections by multiplying with 10.

A	E	F	G	H	I	J	K	L	M
ID	PaymentMo	Branch	Area	Tenu	AssetCc	AmountFinan	DisbursalAmou	EMI	DisbursalDate
4765	PDC	3	JABALPUR	12	500000	200000	200000	370	2012-12-10 00:00:00

```python
print('correcting EMI...')

def func(args):
    total_repayment = args[0] * args[1]
    if total_repayment < args[2]:
        return args[0] * 10.
    return args[0]

df['EMI'] = df[['EMI', 'Tenure', 'DisbursalAmount']].apply(func, axis=1)
```

10) Took the highest credit or the DISBURSED-AMT as name implies,

```
print('correcting DISBURSED-AMT/HIGH CREDIT...')

df['DISBURSED-AMT/HIGH CREDIT'] = df[['CREDIT-LIMIT/SANC AMT', 'DISBURSED-AMT/HIGH CREDIT']].apply(lambda args: args[0] if args[
```

Engineering basic features

1) Early features engineering include finding certain **"ratios", "differences" and "multiplicative"** features

```
print("extracting dual features...")

ratio_features_list = [
    ('p_ra_con_DisbursalAmt_AssetCost', 'DisbursalAmount', 'AssetCost'),
    ('p_ra_con_AssetCost_MonthlyInc', 'AssetCost', 'MonthlyIncome'),
    ('p_ra_con_EMI_MonthlyInc', 'EMI', 'MonthlyIncome'),
    ('p_ra_con_seq_ACTIVE_MONTHLY_INSMT_AMT_MonthlyInc', 'p_con_seq_ACTIVE_MONTHLY_INSTALLMENT_AMT', 'MonthlyIncome')
]

ratio_features = get_dual_features(df, ratio_features_list)

multiply_features_list = [
    ('p_mu_con_EMI_Tenure', 'EMI', 'Tenure'),
]
multiply_features = get_dual_features(df, multiply_features_list, 'multiply')
df = pd.concat([df, ratio_features, multiply_features], axis=1)

difference_features_list = [
    ('p_di_con_AssetCost_DisbursalAmt', 'AssetCost', 'DisbursalAmount'),
    ('p_di_con_AmtFinance_DisbursalAmt', 'AmountFinance', 'DisbursalAmount'),
    ('p_di_con_AssetCost_AmtFinance', 'AssetCost', 'AmountFinance'),
    ('p_di_con_Totalrepayment_AssetCost', 'p_mu_con_EMI_Tenure', 'AssetCost'),
    ('p_di_con_seq_TT', 'Tenure', 'TENURE'),
]
difference_features = get_dual_features(df, difference_features_list, 'difference')
df = pd.concat([df, difference_features], axis=1)
```

2) Then some additonal features based on dates from bureau and ltfs loan tenure were re-calculated.

```
print('extracting more features...')

df['p_con_REPAYMENT_RATIO'] = df[['EMI', 'Tenure', 'DisbursalAmount']].apply(lambda args: (args[0] * args[1]) / args[2], axis=1)
df['p_cat_seq_IS_NEW_LOAN'] = (df['DISBURSED-DT'] > df['DisbursalDate']).map(int)
df['p_con_seq_NB_DAYS_SINCE_LTFS'] = (df['DISBURSED-DT'] - df['DisbursalDate'].shift(1)) / np.timedelta64(1, 'D')

df['p_con_LOAN_TENURE'] = (df['MaturityDAte'] - df['DisbursalDate']).dt.days
df['p_con_AGE_OF_PERSON_AT_MATURITY'] = df['AGE'] + (df['Tenure'] / 12)
```

3) Binning continuous values like **"LTV"** and **"AGE"** features,

```
print('categorical binning...')
df['LTV_BINS'] = df['LTV'].map(int)
df['AGE_BINS'] = df['AGE'].fillna(0).map(int)
```

```
categorical binning...
```

4) Preparing composite categorical features like combination of "ACC-TYPE" AND "CONTRIBUTER-TYPE". These features shall be later used for finding new insights into continouous features by grouping on them.

```
print('categorical pairing...')
df['ACCT_CONTRIBUTOR_ID'] = df['ACCT-TYPE'].map(str) + df['CONTRIBUTOR-TYPE'].map(str)
df['ACCT_OWNERSHIP_ID'] = df['ACCT-TYPE'].map(str) + df['OWNERSHIP-IND'].map(str)
df['CONTRIBUTOR_OWNERSHIP_ID'] = df['CONTRIBUTOR-TYPE'].map(str) + df['OWNERSHIP-IND'].map(str)
```

```
categorical pairing...
```

5) Binary bins from target were created, these will be used later in **FeatureTools"** to create **percent_true** primitive features

```
print('binary target binning...')
df['F_NO_TOP_UP'] = df['Top-up Month'].map(lambda t: 1 if t == 'No Top-up Service' else 0)
df['F_48_MONTHS'] = df['Top-up Month'].map(lambda t: 1 if t == ' > 48 Months' else 0)
df['F_24_30_MONTHS'] = df['Top-up Month'].map(lambda t: 1 if t == '24-30 Months' else 0)
df['F_12_18_MONTHS'] = df['Top-up Month'].map(lambda t: 1 if t == '12-18 Months' else 0)
df['F_18_24_MONTHS'] = df['Top-up Month'].map(lambda t: 1 if t == '18-24 Months' else 0)
df['F_30_36_MONTHS'] = df['Top-up Month'].map(lambda t: 1 if t == '30-36 Months' else 0)
```

Engineering normalized continuous values using DataFrame.groupby

1) A groupby over **"ACCT-TYPE"** to determine normalized disbursal rates, normalized overdue amounts over account types and so on...

```
def func(df):
    columns_ = [
        ('p_con_seq_NORM_DISBURSED_AMT', 'DISBURSED-AMT/HIGH CREDIT'),
        ('p_con_seq_NORM_INSTALLMENT_AMT', 'p_con_seq_MONTHLY_INSTALLMENT_AMT'),
        ('p_con_seq_NORM_CURRENT_BAL', 'CURRENT-BAL'),
        ('p_con_seq_NORM_OVERDUE_AMT', 'OVERDUE-AMT'),
    ]
    for cnew, c in columns_:
        df[cnew] = np.nan if df[c].isnull().all() else df[c] - df[c].mean()

    return df

essential_columns = [
    'REPORT_ID',
    'ACCT-TYPE',
    'DISBURSED-AMT/HIGH CREDIT',
    'p_con_seq_MONTHLY_INSTALLMENT_AMT',
    'CURRENT-BAL',
    'OVERDUE-AMT',
]

jb = ['REPORT_ID']

file_path = f'{path}/df_groupby_ACCT_TYPE.pkl'
if os.path.isfile(file_path):
    df_grp = pd.read_pickle(file_path)
else:
    df_grp = df[essential_columns].groupby('ACCT-TYPE').progress_apply(func).reset_index(drop=True)
    df_grp.to_pickle(file_path)

df = df.merge(df_grp.drop(list(set(essential_columns)^set(jb)), axis=1), on=jb, how='left', suffixes=('_suffix', None))
df.drop([c for c in df.columns if '_suffix' in c], axis=1, inplace=True)
```

2) Similarly over **"AGE_BINS"** to determine the monthly income worth normalized across age of other loan owners.

```
def func(df):
    if df['MonthlyIncome'].isnull().all():
        df['p_con_NORM_MonthlyIncome'] = np.nan
    else:
        df['p_con_NORM_MonthlyIncome'] = df['MonthlyIncome'] - df['MonthlyIncome'].mean()

    return df

essential_columns = [
    'REPORT_ID',
    'AGE_BINS',
    'MonthlyIncome',
]

jb = ['REPORT_ID']


file_path = f'{path}/df_groupby_AGE_BINS.pkl'
if os.path.isfile(file_path):
    df_grp = pd.read_pickle(file_path)
else:
    df_grp = df[essential_columns].groupby('AGE_BINS').progress_apply(func).reset_index(drop=True)
    df_grp.to_pickle(file_path)

df = df.merge(df_grp.drop(list(set(essential_columns)^set(jb)), axis=1), on=jb, how='left', suffixes=('_suffix', None))
df.drop([c for c in df.columns if '_suffix' in c], axis=1, inplace=True)
```

3) Similarly over **"BranchID"** to determine the disbural amount normalized across all branches.

```python
def func(df):
    if df['DisbursalAmount'].isnull().all():
        df['p_con_NORM_DisbursalAmount'] = np.nan
    else:
        df['p_con_NORM_DisbursalAmount'] = df['DisbursalAmount'] - df['DisbursalAmount'].mean()

    return df

essential_columns = [
    'REPORT_ID',
    'BranchID',
    'DisbursalAmount',
]

jb = ['REPORT_ID']


file_path = f'{path}/df_groupby_BranchID.pkl'
if os.path.isfile(file_path):
    df_grp = pd.read_pickle(file_path)
else:
    df_grp = df[essential_columns].groupby('BranchID').progress_apply(func).reset_index(drop=True)
    df_grp.to_pickle(file_path)

df = df.merge(df_grp.drop(list(set(essential_columns)^set(jb)), axis=1), on=jb, how='left', suffixes=('_suffix', None))
df.drop([c for c in df.columns if '_suffix' in c], axis=1, inplace=True)
```

Enter FeatureTools

- [https://www.featuretools.com (https://www.featuretools.com)](https://www.featuretools.com)

1) FeatureTools requires you to have a identifier for every record, thus creating **"REPORT_ID"**

```
print('preparing data for featuretools...')
df['REPORT_ID'] = range(df.__len__())
```

```
preparing data for featuretools...
```

2) Converting date into ordinals, these will be used for **"min", "max", "mean" and "std"** aggregation primitives

```
print('creating temperory date ordinals...') # only for boostings algo's
for col in ['CLOSE-DT', 'DISBURSED-DT', 'DATE-REPORTED', 'LAST-PAYMENT-DATE']:
    df[f"temp_{col}"] = df[col].map(date_to_integer)
```

```
creating temperory date ordinals...
```

3) In featuretools the dataset can be divide into entities. These entities are then related and features can be extracted by targeting them. Using a custom mapping dictionary we then create an entitiy set for our data which looks like below,

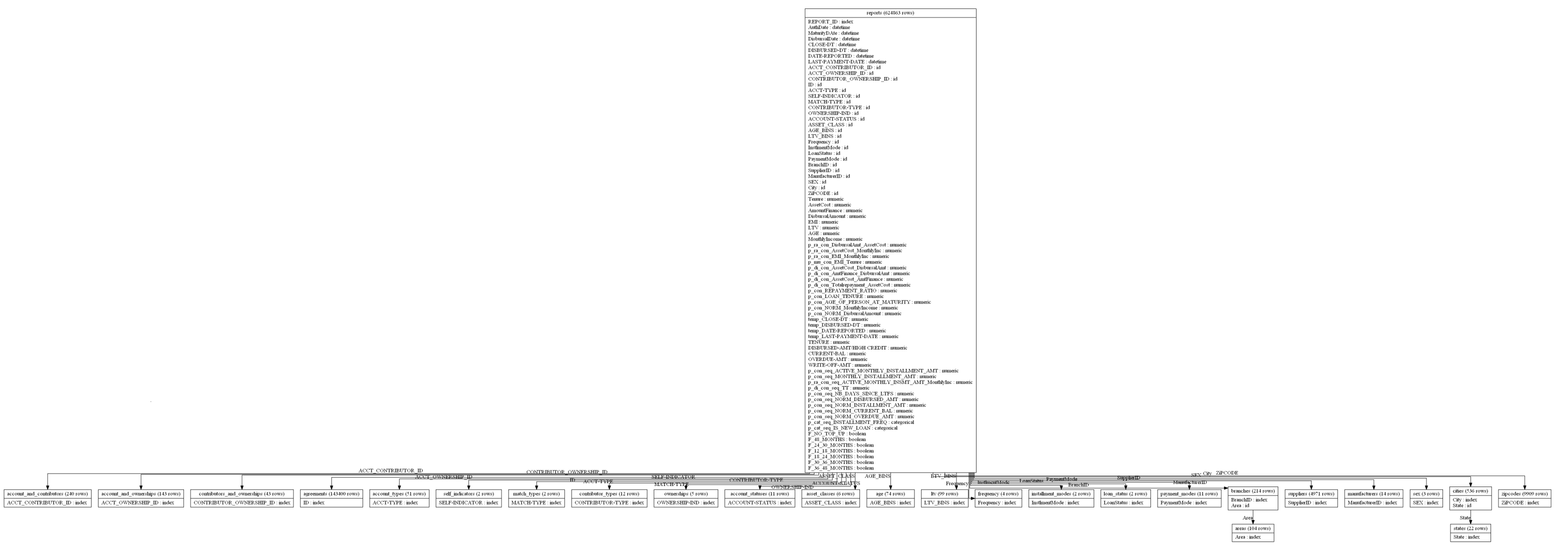
```
es = ft.EntitySet()

# entity_id: [entity_index, [entity_id, {entity_variables}]]
main_mapping = {
    'account_and_contributors': ['ACCT_CONTRIBUTOR_ID', {
        'ACCT_CONTRIBUTOR_ID': ft.variable_types.Index,
    }],
    'account_and_ownerships': ['ACCT_OWNERSHIP_ID', {
        'ACCT_OWNERSHIP_ID': ft.variable_types.Index,
    }],
    'contributors_and_ownerships': ['CONTRIBUTOR_OWNERSHIP_ID', {
        'CONTRIBUTOR_OWNERSHIP_ID': ft.variable_types.Index,
    }],
    'agreements': ['ID', {
        'ID': ft.variable_types.Index,
    }],
    'account_types': ['ACCT-TYPE', {
        'ACCT-TYPE': ft.variable_types.Index,
    }],
    'self_indicators': ['SELF-INDICATOR', {
        'SELF-INDICATOR': ft.variable_types.Index,
    }],
```

Intialize new entitiy set

Telling featuretools that this is the identifier of this entity

```python
# ==========================================
# Adding numeric variables
# ==========================================
main_variables['Tenure'] = ft.variable_types.Numeric
main_variables['AssetCost'] = ft.variable_types.Numeric
main_variables['AmountFinance'] = ft.variable_types.Numeric
main_variables['DisbursalAmount'] = ft.variable_types.Numeric
main_variables['EMI'] = ft.variable_types.Numeric
main_variables['LTV'] = ft.variable_types.Numeric
main_variables['AGE'] = ft.variable_types.Numeric
main_variables['MonthlyIncome'] = ft.variable_types.Numeric

for c in additional_con_cols:
    main_variables[c] = ft.variable_types.Numeric

for c in ['CLOSE-DT', 'DISBURSED-DT', 'DATE-REPORTED', 'LAST-PAYMENT-DATE']:
    main_variables[f"temp_{c}"] = ft.variable_types.Numeric

for c in source_con_cols_seq + additional_con_seq_cols:
    main_variables[c] = ft.variable_types.Numeric

for c in additional_cat_seq_cols + additional_cat_cols:
    main_variables[c] = ft.variable_types.Categorical

main_variables['F_NO_TOP_UP'] = ft.variable_types.Boolean
main_variables['F_48_MONTHS'] = ft.variable_types.Boolean
main_variables['F_24_30_MONTHS'] = ft.variable_types.Boolean
main_variables['F_12_18_MONTHS'] = ft.variable_types.Boolean
main_variables['F_18_24_MONTHS'] = ft.variable_types.Boolean
main_variables['F_30_36_MONTHS'] = ft.variable_types.Boolean
main_variables['F_36_48_MONTHS'] = ft.variable_types.Boolean
```

Assigning appropriate datatypes to features



4) Then we configure our target entitites like below,

```
    ft_args('agreements_et',
            target_entity='agreements',
            target_entity_id='ID',
            ignore_variables=None,
            ignore_entities=[],
            agg_primitives=['count', 'mean', 'std', 'num_unique', 'min', 'max', 'sum', n_most_common],
            trans_primitives=[],
            interesting_values=BUREAU_LOAN_CAT,
            where_primitives=['count'],
            drop_exact=[],
            max_depth=2,
            primitive_options={
                ('max', 'mean', 'min', 'std'): {
                    'ignore_variables': {
                        'reports': LTFS_LOAN_SPECIFICS_CON + ['p_ra_con_seq_ACTIVE_MONTHLY_INSMT_AMT_MonthlyInc'],
                    }
                },
                ('sum'): {
                    'include_variables': {
                        'reports': ['CURRENT-BAL', 'OVERDUE-AMT', 'p_con_seq_ACTIVE_MONTHLY_INSTALLMENT_AMT'],
                    }
                },
                ('num_unique', n_most_common): {
                    'ignore_variables': {
                        'reports': [c for c in LTFS_LOAN_SPECIFICS_CAT if c not in ['Area', 'State']],
                        'branches': ['Area'],
                        'cities': ['State'],
                    }
                },
            }
    ),
```

GroupBy on this column

Aggregation primitives (functions) to apply

Pivot on these columns

Ignore these columns during aggregating

Only include these columns during aggregating

5) Then we create the features using featuretools.dfs method

- [https://featuretools.alteryx.com/en/stable/generated/featuretools.dfs.html (https://featuretools.alteryx.com/en/stable/generated/featuretools.dfs.html)](https://featuretools.alteryx.com/en/stable/generated/featuretools.dfs.html)

```
print('actual run to create the features...')
features = {te.entity_name : dict() for te in target_entities}
for te in tqdm(target_entities):

    file_path = Path(f'{path}/ftools_{te.entity_name}.pkl')
    if os.path.isfile(file_path):
```

Encoding Features

1) We now encode our categorical features using **category_encoders.OridnalEncoder"**, also encode the target usign **sklearn.preprocessing.LabelEncoder"** and also make sure not to leave an np.inf or -np.inf behind in continouous data,

```
print("encoding target...")
target_encoder = LabelEncoder()
df.loc[df['source'] == 'train', 'Top-up Month'] = target_encoder.fit_transform(df[df['source'] == 'train']['Top-up Month'])
```

```
encoding target...
```

```
print("encoding date and categorical features...")
df[cat_cols + cat_cols_seq] = df[cat_cols + cat_cols_seq].fillna(DEFAULT_CATEGORICAL_FILL)

fucnT = FunctionTransformer(lambda X: X.astype('U'))
df[cat_cols + cat_cols_seq] = fucnT.fit_transform(df[cat_cols + cat_cols_seq])

oe = OrdinalEncoder(return_df=False, handle_missing='return_nan', handle_unknown='error', drop_invariant=False)
df[cat_cols + cat_cols_seq] = oe.fit_transform(df[cat_cols + cat_cols_seq]) - 1

fucnT = FunctionTransformer(lambda X: X.astype('int'))
df[cat_cols + cat_cols_seq] = fucnT.fit_transform(df[cat_cols + cat_cols_seq])

for col in source_dt_cols + source_dt_cols_seq:
    df[col] = df[col].map(date_to_integer)
```

```
encoding date and categorical features...
```

```
print('replacing infinity (if any) with np.nan...')
df[con_cols + con_cols_seq] = df[con_cols + con_cols_seq].replace([np.inf, -np.inf], np.nan)
```

```
replacing infinity (if any) with np.nan...
```

Modelling

1) Worked with three algorithms with **GroupKFold** over **"ID"** column viz. xgboost, catboost and lightgbm. We use **catboost** for creating our submissing file. We equally ensemble 10 catboost models trained over 10 folds of the training data **i.e. preds/n_folds**

```
def run_CGB(params, train, test, feature_names, n_folds = 10, seed=0, cat_cols=None, return_models=False):
    skf = GroupKFold(n_splits=n_folds)
    X, y, groups = train[feature_names], train['Top-up Month'].values.astype('int'), train['ID'].values.astype('int')

    preds = np.zeros((test.shape[0], params['classes_count']))

    models = list()
    for i, (train_index, test_index) in enumerate(skf.split(X, y, groups)):
        X_train, X_val = X.iloc[train_index, :], X.iloc[test_index, :]
        y_train, y_val = y[train_index], y[test_index]

        dtrain = cgb.Pool(data=X_train, label=y_train, cat_features=cat_cols)
        dval = cgb.Pool(data=X_val, label=y_val, cat_features=cat_cols)

        bst = cgb.train(
                params                = params,
                dtrain                = dtrain,
                num_boost_round       = 30_000,
                early_stopping_rounds = 100,
                evals                 = [dval],
                verbose_eval          = 100
            )

        score_, iter_ = bst.get_best_score(), bst.get_best_iteration()
        test_preds = bst.predict(cgb.Pool(test[feature_names], cat_features=cat_cols))
        preds += test_preds
        models.append(bst)

    if return_models:
        return preds / n_folds, models
    return preds / n_folds
```

2) Training model

```python
print('modelling...')
train, test = df[df['source'] == 'train'], df[df['source'] == 'test']
```

```
modelling...
```

```python
config = dict()
config["lgb_params"] = {
    "num_classes": train['Top-up Month'].unique().size,
    "objective": "multiclass",
    "boosting_type": "gbdt",
    "metric" : "multi_logloss",
    "num_threads" : psutil.cpu_count()
}
config["xgb_params"] = {
    "num_class": train['Top-up Month'].unique().size,
    "objective": "multi:softprob",
    "eval_metric": "mlogloss",
    "nthread" : psutil.cpu_count()
}
config["cgb_params"] = {
    "classes_count": train['Top-up Month'].unique().size,
    "objective": "MultiClass",
    "eval_metric": "MultiClass",
    "thread_count" : psutil.cpu_count(),
}

if True:
#     config["lgb_params"]['device'] = 'gpu'
#     config["lgb_params"]['gpu_platform_id'] = 1
#     config["lgb_params"]['gpu_device_id'] = 0
```

Submission

Created a submission file by takeing the most common prediction across records for particular **"ID"**

```python
test['final_preds'] = target_encoder.inverse_transform(np.argmax(preds_, axis=1))

df_submit = test.groupby('ID')['final_preds'].agg(lambda x:x.value_counts().index[0]).to_frame().reset_index()
df_submit.columns = ['ID', 'Top-up Month']
df_submit.to_csv(f"submissions/submission_{datetime.now().strftime('%Y%m%d%H%M%S')}.csv", index = False)
```

Public LeaderBoard Score: 0.389901574412076