Nima Mohammadi & Saeed Reza Kheradpisheh
February 21, 2018

Technical Report

"Deep Threat"

1) Personal Infos:

Nima Mohammadi <nima.mohammadi@ut.ac.ir>

Tel: +98939 838 9509

Address: Apt. #2, Eskandarzadeh St., Ashrafi Esfahani Hwy, Tehran

Saeed Reza Kheradpisheh <kheradpisheh@ut.ac.ir>

Tel: +98912 870 8745

Address: No. 303, Entrance #4, Block #2, Baharan Complex, Razi Street, Rey

2) Proposed Model:

We proposed a two-stage model for this challenge. First we use target encoding to transform the categorical features to numerical ones. These encoded features, along those other numerical features are fed to a GBDT (Gradient Boosting Decision Tree). The leaf predictions are then stored and used in a factorization machines model. Factorization machines are closely related to SVM and LR. Linear regression misses the very informative potential interaction between features but is suitable for sparse data. Moreover, poly-2 regression is computationally and practically inappropriate. Factorization Machines formulate the problem of capturing the interactions between features in a tractable framework. We use a variant of FM, called FFM (Field-aware Factorization Machines) which can also take into account the fields each feature belongs to, and better grasp the underlying patterns, leading to better predictions.

3) Feature Engineering:

Users:

We have extracted different statistics from sparse files, including min, max, average, mean absolute deviation, sum, standard deviation and number of NaNs. We also used PCA on these sparse files to extract more compact transformation.

Notifs:

For words, we used TF-IDF to extract more meaningful features for sentences representing each notifs. KMeans clustering was applied to cluster this notification. The number of words each notification is comprised of also proved helpful.

Icons:

We used hashing and clustering on this dataset.

Interactions:

We employed SVD (singular value decomposition) on mean-centered interactions (click/noclick) of users and notifications. Moreover we used clustering on the latent user and notif factors extracted from SVD to find similar users and notifications.

GBDT features:

We employed GBDT to somehow transform our numerical features to categorical ones, suitable for our factorization machines model.

Temporal features:

Various features regarding in the nature of time and order are extracted. We reconstructed the missing timestamp of the notifications. Then we augmented our feature space, with various features representing the history of interactions user had with the system.

Target encoding:

Categorical features are inappropriate for GBDT, so we use a target-encoding approach, which as the name implies, uses targets of the samples, but in a k-folding fashion that prevents leakage.

4) Implementation

The implementation is done using Python, Cython and C++, in order of code written. We used Pandas dataframes for loading the datasets, data manipulation and extracting various stats.

CHPC of University of Tehran, School of Mathematics, Statistics and Computer Science kindly granted us to use the powerful computational infrastructure of theirs to carry out our computations. We had exclusive access to an overhauled Linux server of x86_64 E5-2697 v2 (48 cores) and 200GB RAM during our development. Over last two days of the competition we ran our factorization machines of different parameters and features on another server of x86_64 E7-8890 v4 @ 2.20GHz (192 cores) and 500GB RAM. We also had access to two high-end and powerful Nvidia Titan GPU cards and tried to use them to some extent, but due to the massive size of dataset which made using GPUs of limited memory a hassle, and our aversion to bother with our-of-core training, mainly for our fortunate access to vast amount of CPU and RAM, we decided not to pursue this.

Putting the data preparation phase aside, our decision tree leaf predictions would take around ~10 minutes on LightGBM and ~20 minutes on XGBoost (on 48 cores). Our factorization model also very effectively uses multithreading and SSE instructions and its computation time highly depends on the number of CPU cores. Based on the number of features and the number of cores we assigned (40-90), it would take between 10 minutes to 40 minutes to iterate over each epoch; only 1-3 epochs sufficing for training without overfitting too much.

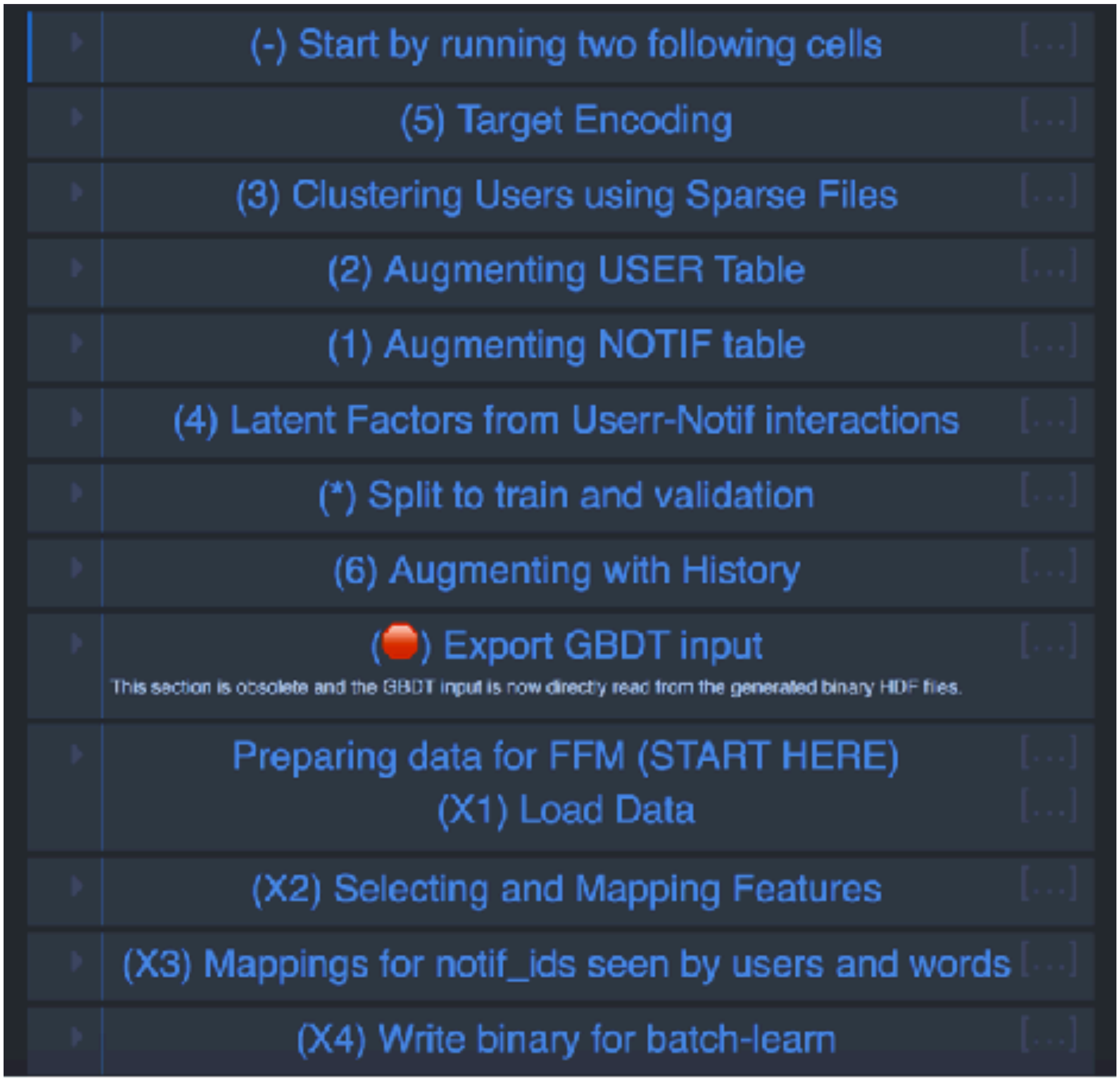
5) Running code

The code is uploaded to a github repository located at
https://github.com/nimamox/deca_challenge
First off, you need to "pip install" the following packages:

* `lightgbm`
* `xgboost`
* `cython`
* `matplotlib`

* pandas
* tqdm
* numpy
* scipy
* sklearn
* jupyter

The codes are located in three files, namely ‘prepare_data.ipynb’, ‘gbdt.ipynb’ and ‘ensemble.ipynb’. Frankly how the development of the code has been evolved makes running it a bit complicated. Most of the codes reside in ‘prepare_data’ notebook. The amount of code in this file is rather enormous. You are advised to install pip-install ‘jupyter-contribs’ and enable the ‘collapsable headings’ nbextension. Upon doing so, you an see that the code is structured in different sections, depicted in the picture below:



(-) Start by running two following cells

(5) Target Encoding

(3) Clustering Users using Sparse Files

(2) Augmenting USER Table

(1) Augmenting NOTIF table

(4) Latent Factors from Userr-Notif interactions

(*) Split to train and validation

(6) Augmenting with History

(🛑) Export GBDT input

This section is obsolete and the GBDT input is now directly read from the generated binary HDF files.

Preparing data for FFM (START HERE)

(X1) Load Data

(X2) Selecting and Mapping Features

(X3) Mappings for notif_ids seen by users and words

(X4) Write binary for batch-learn

You start by running the first section, coded idiomatically as (-), by that we mean running the cells in under this header. Then you can run the section (1) through (6) to generate required data. Notice that you need to restart the kernel each time, run section (-) to load common modules, and then run the section in ordering implied here. A description of each section is provided below:

SECTION (1) It prepares some augmenting features on notifications: Mainly it first loads notifs.txt file, converts it to a .csv file after some processing on the words, loads icons, hashes the icons, performs kmeans on icons. We found out that the notif_id seemingly reveals the actual timestamp of the notifications. So we had notif_send_dow before, but now we can find out the actual week number the notification was sent out. After extracting notif_send_week, it hashes the words of each notification, identifying duplicate notification messages. It also performs binning on notif_send_hour. Moreover the words describing the notifications are going through TF-IDF transformation, taking 2-grams into account. We found out that the number of words within a notification is an information feature, so it is also calculated. The resulting file is stored under the name of notifs_table_augmented.h5.

SECTION (2) This section works on the two sparse user files. It loads the files and calculates PCA on the standardized matrices. Ten principal components are calculated, four of which are stored. KMeans clustering is also performed on the two PCA-transformed matrices. On the original matrices, we extract some statistics for each user, namely the number of non-zero columns, max, min, mean absolute deviation, sum, average and standard deviation.

SECTION (3) This section also performs clustering on the sparse files, but the original matrices rather than the PCA-transformed ones.

SECTION (4) This section builds a massive sparse matrix of user-notification interactions. The rows in this matrix indicate users and there is a column for each notification. In case the user and notifications had no interaction, the element is set to zero. M[*i*, *j*] is set to one if user *i* had an interaction with notification *j* (i.e. user had received the notification) and is set to two if user had clicked on the notification. We then mean-center this matrix and run SVD (singular value decomposition) and extract top 10 eigenvectors for users and notifications. SVD is supposed to extract the main latent features representing the affinity and taste of users. We store these eigenvectors. We also cluster these eigenvectors using KMeans, hoping to put users with similar tastes (and notifications with similar audience as well) inside identical clusters.

SECTION (5) This section performs target encoding. GBDT can not easily deal with categorical features. Therefore we use a procedure called target encoding to transform these categorical features to numerical ones. We usually use the mean of target column (interaction) to this end. But obviously this can lead to overfitting. So we will have a two-stage "k-folding". Say we are encoding "C1": We split data to 10 folds first, then for each fold, the selected samples go through another k-folding, splitting them into 20 folds. For each fold, the corresponding numerical value for a C1 categorical feature is equal to the average of the target column, for those C1 features, within the other 19 folds! This makes sure that we don't leak too much information about the target of the sample and still providing a very informative transformation. The missing categorical features, or unseen ones, are replaced with global mean. This process can take a lot of time (~1 hour for each categorical column). So this section writes a python script in local directory which you can run, with appropriate arguments, to get target encoded features. You can run this script multiple times, and concurrently, to more quickly get the results.

There are some other sections in prepare_data (X1, X2, X3 and X4) which we will return to, but we first need to explain GBDT.ipynb as its results are needed before running these remaining sections.

GBDT.ipynb Our factorization machines has a very hard time to deal with numerical features. While GBDT is one of the best models for tabular numerical datasets. We are going to construct GBDT (using both XGBoost and LightGBM) and then instead of using the actual predictions of GBDTs, we perform leaf prediction. In other words, for a GBDT with 30 trees, we ask the model to prediction, given a sample, in which leaf of each tree the sample ends up! In a sense, we use a GBDT as transformation of numerical features to categorical ones suitable to be used with the factorized machines.

Back to prepare_data:

SECTION (X1) Now after restarting the kernel we first run this section. It loads all required data needed for the factorization machines. Up there, there is a gbdt_filename variable which hold the root of the name of the GBDT leaf predictions. Set this according to which GBDT you want to load. It then loads and merges various tables that we've created so far. It also removes all the unseen features from test data which would potentially mislead our model during evaluation on test dataset.

SECTION (X2) This section first selects some of the feature that we are going to include. There is a FILE_NUM variable which can hold 1 or 2 and based on this it can will include two set of different features. We used this to generate two rather diverse datasets whose eventual ensemble would hopefully achieve better results. After selecting features we need to map each field and each feature to a unique number. This transformation is performed in the second part of this section.

SECTION (X3) In case 'words' or 'imps' (short for impression) is selected, these features are also mapped to unique numerical values.

SECTION (X4) This section first shifts the mapped feature values so there is no overlapping feature among fields. Then it writes a binary files which are used by batch-learn to train FFM and NN models. As the datasets are enormous and handling this would take a lot of time (writing a 50GB file with python would take a couple of hours) we wrote a highly optimized code in Cython to handle writing the files. Running the last cell of this section produces output like below:

```
/home/nima_mohammadi/ffm_files_latest/39__full_train_xgb_gbdt_hist_50F.bl
FULL TRAIN: 284.716874123s
/home/nima_mohammadi/ffm_files_latest/39__test_xgb_gbdt_hist_50F.bl
TEST: 31.6942219734s
```

Which gives the path of parameters you need to give to batch-learn. Moreover four files are generated as a result of this:

- `39__full_train_xgb_gbdt_hist_50F.bl.index`
- `39__full_train_xgb_gbdt_hist_50F.bl.data`
- `39__test_xgb_gbdt_hist_50F.bl.index`
- `39__test_xgb_gbdt_hist_50F.bl.data`

We use a modified version of a program called batch-learn to train our models. We have included our cost function (and its gradient) along some minor patches. This program is highly resource-intensive and uses OpenMP for multi-thread programming and SSE instruction set to concurrently calculate vector arithmetics (four operations with a single CPU core). I've included the code in the repository. I've also modified the CMake script to build a statically-lined version. The binary is included in batch-learn/build-static/

You can run **field-aware factorization machines**, as below:

```
$ batch-learn ffm -k 5 -t 48 --epochs 2 --train
ffm_files_latest/39__full_train_xgb_gbdt_hist_50F.bl --test
39__test_xgb_gbdt_hist_50F.bl --pred “preds_test_ffm.txt"
```

The parameters are self-explanatory, except for -k which sets the number of latent factors in the model. Usually 4 or 5 would be a good choice for this argument. Also -t sets the number of computational cores to use during training.

You can run the neural network as below:

```
$ batch-learn nn -t 48 --epochs 2 --train ffm_files_latest/
39__full_train_xgb_gbdt_hist_50F.bl --test
39__test_xgb_gbdt_hist_50F.bl --pred “preds_test_nn.txt"
```

Last bu not least, the ensembling:

ensembling.ipynb This file takes a number of predictions generated by various configurations of model on diversified datasets and simply combine them using geometric means. We also calibrate the results based on our empirical findings.

6) Participants' background

Nima Mohammadi holds a MSc. of Computer Science from University of Tehran. He was formerly a PhD candidate at IPM (and DSN Lab of SUT). He currently manages to find time to amuse himself with data science challenges like yours! His undergrad and grad theses were on Brain-Computer Interfacing and Spiking Neural Networks, respectively. He is Deep Learning enthusiastic and although his Master's thesis is more about biologically inspired neural networks modeling visual cortex, he still very much enjoys the recent progress of the more artificially-inclined neural nets. He had attended Prof. Soleymani's class and that contributes a lot in his appreciation of ML models other than neural networks. Nima has publication on Ensemble methods, BCI and protein-protein interactions.

Saeed Reza Kheradpisheh holds a PhD of Computer Science from University of Tehran. For his sabbatical he spent 9 months at Centre de Recherche Cerveau & Cognition, CNRS, France. He is currently an adjunct professor at University of Tehran. He has publication on Ensemble methods, Cortical Mechanisms and Vision.

Our background in Machine Learning, Ensemble Techniques clearly helped us with this competition. Although we are relatively new to the world of tabular data prevalent in data mining tasks, we are happy with how quickly we managed to adapt.

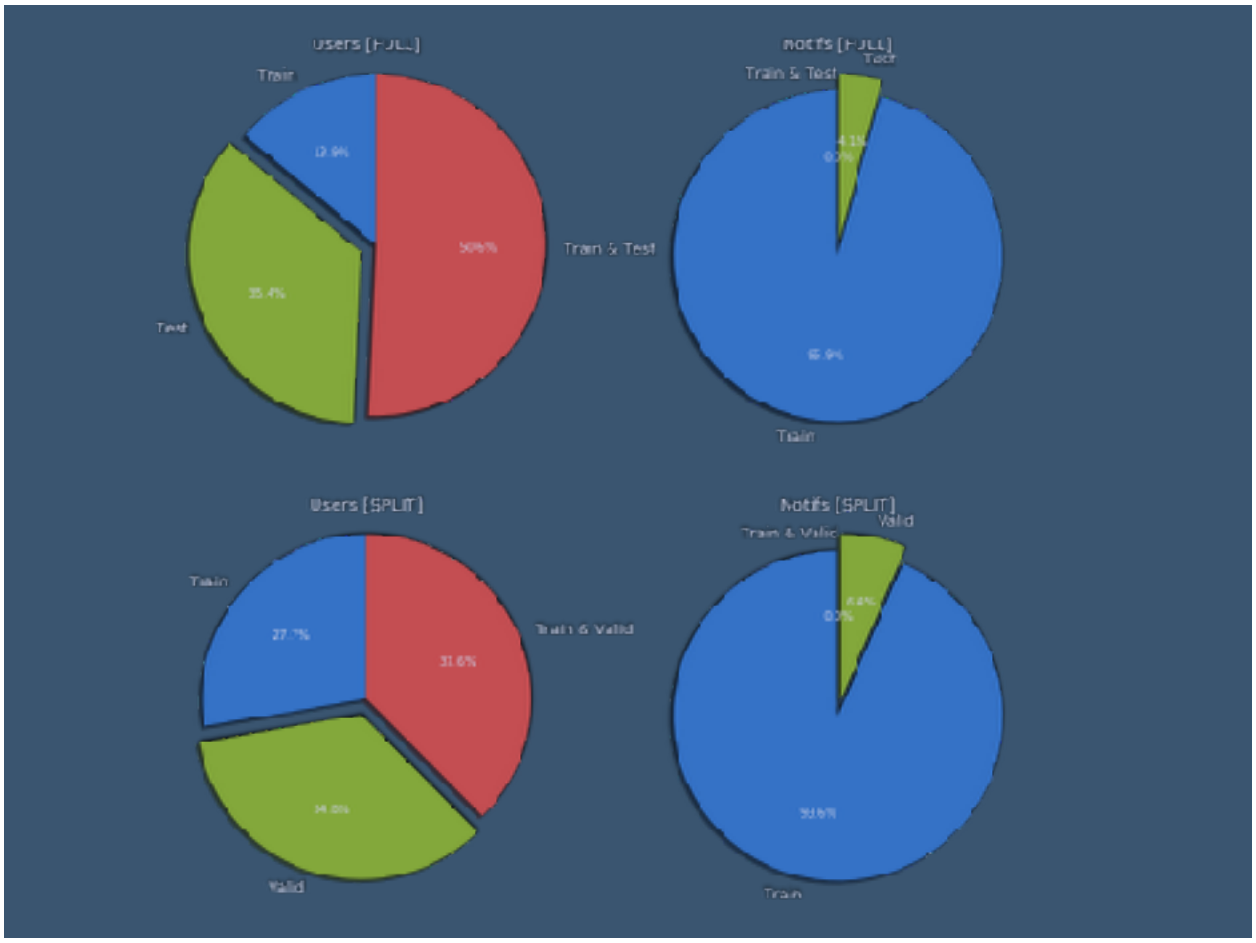
7) Challenge Participation

We were informed about this challenged by a dear friend (Dr. Eslamimehr, former vice president of innovation center of Sharif University). You could say that, for the first month, we only dabbled with the data and explored some crazy ideas. The second month we started to take the task more seriously and incorporate the seminal works of giants like Facebook and Google on similar problems. Being the very occupied people that we are (very cool and handsome as well, needless to say!) we could only spend ~7-8 hours per week on this challenge.

We were disappointed by how much potential the challenge had, and how less the hosts were willing to deliver. Noticeably, the amount of features provided for each notification was by no means enough. There was only one categorical feature for each notification (with cardinality of seven!) The corpus of words are limited to 2000 words and no description on how they are selected is provided. Have you removed stopwords or most frequent words? You could a method like word2vec to deliver word vectors without leaking sensitive informations and still provide some semantic meaning of the words. Icons could be potentially used to identify notifications of same origin, but the identical ones among train and test datasets were disappointing. Moreover, I'd put the notification timestamps without breaking it into day of week, hour and minute. Although notif_id could be used to deduce the actual day the notification was sent, it proved to have some inconsistencies. The sparse files were real headaches though. We couldn't wrap our heads around what information they might be holding. Having anonymized the features, it would pose any threats to you to give some hints on what they could possibly hold. But the mind-numbingly decision made was to choose a different distribution for the test data and not describing your way of choosing the samples for test dataset. This is a great shortcoming which can make the machine learning solutions irrelevant depending on how much the test data deviates the train data; that is deviation imposed by human interference with sampling, not the natural variance between train and test. Anyways, we are glad that you hosted this challenge. Thank you :)

Appendix:

We needed to create a validation set to mimic the test dataset. It wasn't easy as the sampling process during test was vague and hadn't been explained by the hosts. The picture below shows how similar our train-valid sets are to full_train-test sets provided by the hosts. Our score was considerably different between test and valid (test set seemed to be easier for our model) but fortunately an improvement of score on validation set would suggest an improvement on test set.


Users [FULL]
Train
Test
Train & Test
Notifs [FULL]
Train & Test
Test
Train
Users [SPLIT]
Train
Train & Valid
Valid
Notifs [SPLIT]
Train & Valid
Valid
Train