[PMLDL] D1.2 Report

Team

Team name: *16th Data Science Division*
Team members:
- Bulat Sharipov (DS-01); b.sharipov@innopolis.university
- Dinar Yakupov (DS-01); d.yakupov@innopolis.university
- Danil Fathutdinov (DS-01); d.fathutdinov@innopolis.university

Github

Link

Project Topic

Creating a Recommendation Systems using Graph Neural Networks and Yambda Dataset. We will basically create a web-service that will recommend you next song to listen to, given your previous preferences.

Current progress

With team members we have outlined three main tasks, which we want to accomplish during this checkpoint.

Creation of a basic Recommendation Web-Service

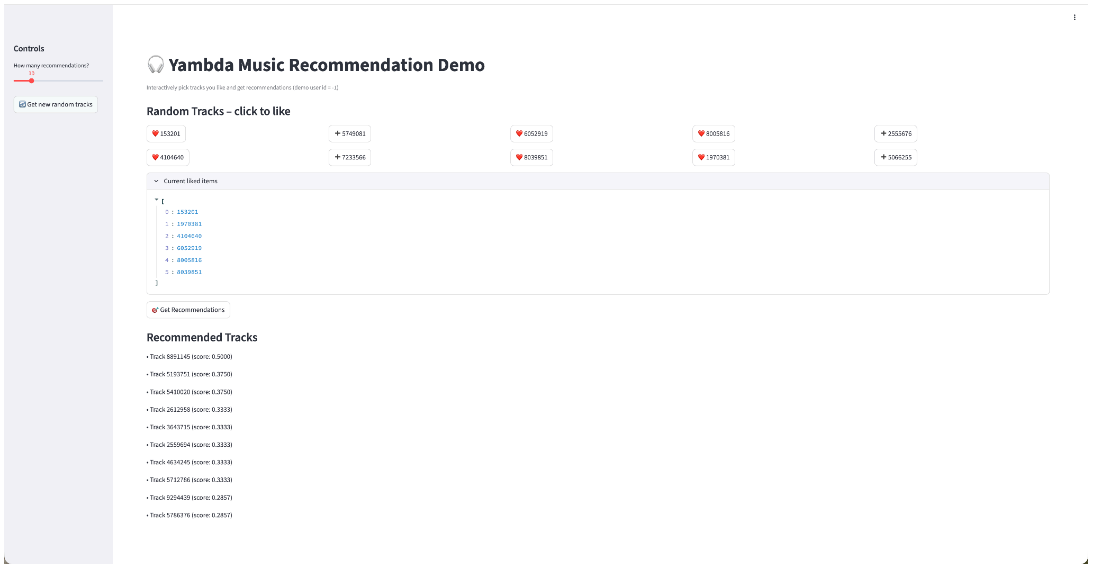
Our project implies building a web-service at the end. Therefore, we decided to create for now a basic web-service to make recommendations. Here is the outline for design choices:
- **Backend Framework**: FastAPI, since most of the team members are familiar with it, and it also provides a great and comprehensive documentation about the library. We can think of it like a standard choice for deploying ML application
- **RecSys Algorithm:** We have decided to use the ItemKNN algorithm. In the paper, authors showed that the ItemKNN achieves best metrics for the 50M version of the dataset. While with the increase of dataset size NN approaches dominate, ItemKNN still provides strong results. We have chosen to implement the ItemKNN from scratch. The process of recommendation outlined below:

1. Initially, build a user-item matrix from the dataset, and store it in memory for all the queries. We did this very efficiently by utilizing SciPy Sparse Matrix, and creating appropriate mappings. Here, we also calculate each item's popularity, and reuse it across all queries.
2. For each query, obtain a set of items, which were liked by the user. This is done very efficiently through Pandas vectorized operations.
3. For each liked item, retrieve top-k similar to its items. Similarity of two items are calculated using Jaccard Similarity. Similarity calculation was optimized using Sparse Matrix properties and SciPy;
4. Accumulate the total scores from retrieved similar items, and return the top-n of best matches
5. The total time complexity T = O(Σ_{s in S} (Σ_{u in U_s} deg(u) + I + k log k) + C log N), S - set of liked items, U_s - users who interacted with some item s in S; I - set of items, k - number of neighbors for each item; C - number of unique candidates item produced, N - number of recommended tracks to retrieve. Generating one recommendation with top-n = 10, on 50M dataset with M3 Pro takes < 200 ms;

- **User Interface Framework:** For user interface framework we have used Streamlit, since it provides convenient utilities for building a basic show-case User Interface. When someone enters the UI, he should firstly get a set of random items, from which it will take some of them as an initial liked track. Later, we will compute recommendations based on the chosen items.

Below are some screenshots of the work

User Interface



🎧 Yambda Music Recommendation Demo

Interactively pick tracks you like and get recommendations (demo user id = -1)

Random Tracks – click to like

❤️ 153201	+ 5749081	❤️ 6052919	❤️ 8005816	+ 2555676
❤️ 4104640	+ 7233566	❤️ 8039851	❤️ 1970381	+ 5066255

Current liked items

```
[
  0 : 153201
  1 : 1970381
  2 : 4104640
  3 : 6052919
  4 : 8005816
  5 : 8039851
]
```

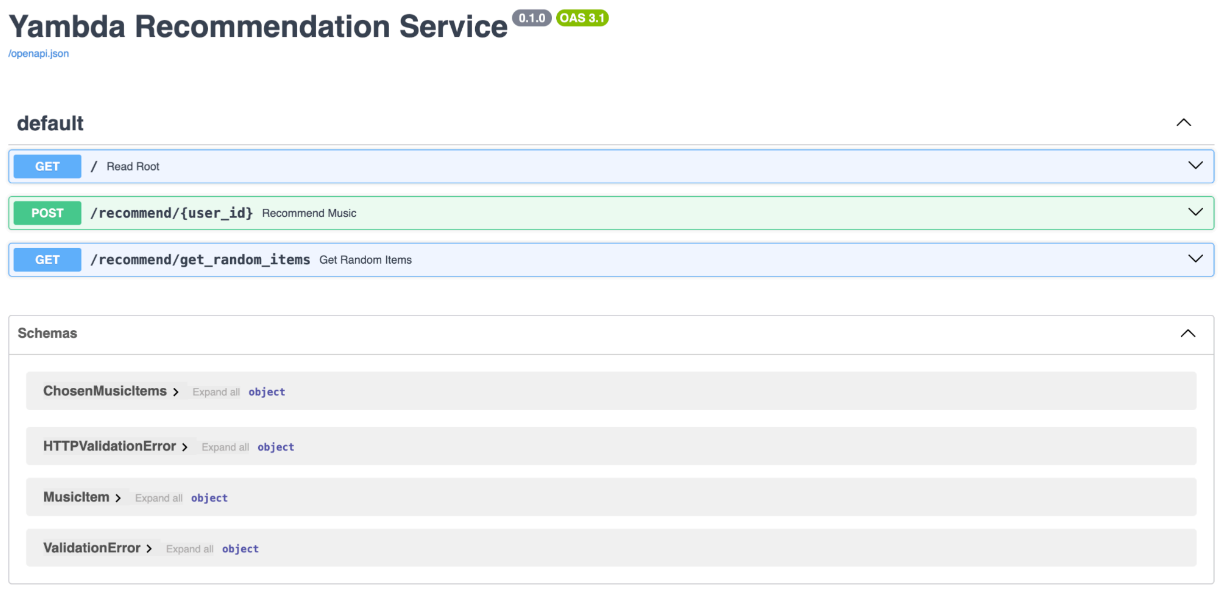
Recommended Tracks

• Track 8891145 (score: 0.5000)
• Track 5193751 (score: 0.3750)
• Track 5410020 (score: 0.3750)
• Track 2612958 (score: 0.3333)
• Track 3643715 (score: 0.3333)
• Track 2559694 (score: 0.3333)
• Track 4634245 (score: 0.3333)
• Track 5712786 (score: 0.3333)
• Track 9294439 (score: 0.2857)
• Track 5786376 (score: 0.2857)

Main code for recommendation

```python
def recommend_items_to_user(self, user_id, neighbors_per_liked=50, top_n=100, prebuilt=None, user_items=None):
    # Generate top-N recommended items for the given user via item-based neighborhood aggregation.
    # Steps:
    #   1. Obtain user profile (set of liked items)
    #   2. For each seed item s in profile, fetch top-k similar items
    #   3. Accumulate similarity scores for candidate items not already liked
    #   4. Return highest scoring unseen items
    if not user_items:
        user_items = set(self.likes_df.loc[self.likes_df['uid'] == user_id, 'item_id'].unique())
        if not user_items:
            return []

    # Seed items = user profile
    seeds = np.array(list(user_items), dtype=np.int64)
    # Accumulate candidate item scores
    scores = defaultdict(float)

    for seed in seeds:
        # For each seed item compute neighbors (see complexity in helper): O(Σ deg + I + k log k)
        neigh = self._nearest_items_topk_jaccard_sparse(seed, neighbors_per_liked)
        # Aggregate over k neighbors: O(k)
        for nid, s in neigh:
            if nid in user_items or nid == seed:
                continue
            scores[nid] += s
    # Overall loop complexity ≈ O( Σ_{s in S} ( Σ_{u in U_s} deg(u) + I + k log k ) )
    # For sparse data and small |S| this is typically manageable; dominated by intersection computations.

    if not scores:
        return []

    # Select top-N candidates via heapq.nlargest: O(C log N) where C = number of candidate items (distinct neighbors across seeds)
    top = heapq.nlargest(top_n, scores.items(), key=lambda kv: kv[1])

    # Total recommendation complexity summarized:
    #   O( Σ_{s in S} ( Σ_{u in U_s} deg(u) + I + k log k ) + C log N )
    # Notation:
    #   S = set of seed items (user profile size)
    #   U_s = users who interacted with seed s
    #   deg(u) = number of items user u interacted with
    #   I = total number of items
    #   k = neighbors_per_liked
    #   C = number of unique candidate items produced
    return top
```

Backend API



Yambda Recommendation Service 0.1.0 OAS 3.1
/openapi.json

default

GET / Read Root

POST /recommend/{user_id} Recommend Music

GET /recommend/get_random_items Get Random Items

Schemas

ChosenMusicItems > Expand all object

HTTPValidationError > Expand all object

MusicItem > Expand all object

ValidationError > Expand all object

Experiment with MB-GCN

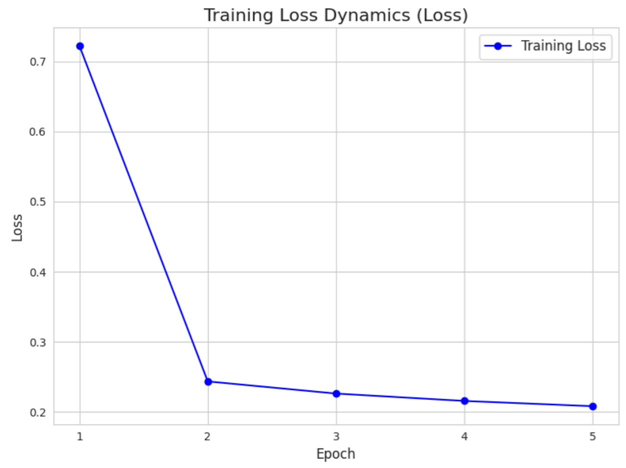
The research on the three most promising graph architectures for recommendation systems was conducted, such as R-GCN (Relational Graph Convolution Network), CompGCN (Composition-based Graph Convolution Network), and MBGCN (Multi-Behaviour Graph Convolution Network). All these architectures are addressed to use different types of relationships and their features, not only structural information about the graph. The short description of the architectures are provided below:

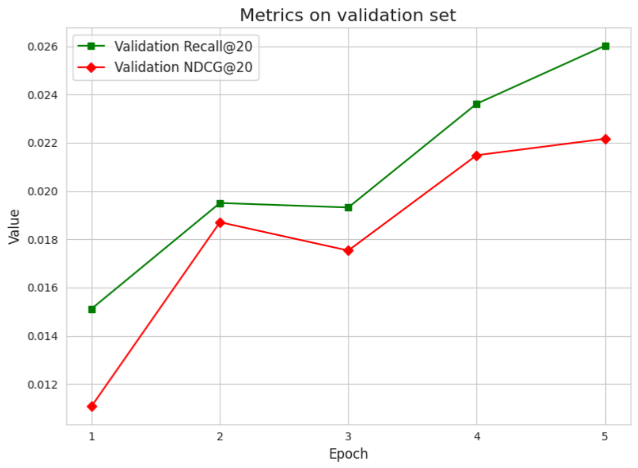
1. R-GCN is an architecture that learns a unique neural network transformation for each type of the relation in the graph. The transformation is applied to the neighbor node's embedding based on this transformation during the message passing stage.
2. CompGCN is an architecture that represents the relations as embeddings. The composition operation is then used to combine node's and relation's embeddings before passing the message.
3. MBGCN is an architecture that builds separate convolution blocks for each of the relationship's types. The unique process in the architecture is to aggregate outputs of each of the blocks to update the node's embedding. The aggregation is flexible, because it can be parallel or cascade. The parallel aggregation is based on applying the aggregation function directly on the parameters. The cascade aggregation assumes that there is a strict sequence between relations (e.g. time sequence in the actions "listen->like->unlike") and applies the aggregation function specifically.

The MBGCN is chosen as the most promising architecture for the given task.
The experiment consists of dataset preprocessing, model implementation, and model training and evaluation. The data are processed to setup "Listen+", where the only "Listen" relationship with played track ratio not less than 50% was considered. Also, the dataset was sampled for time performance goals.
The training procedure consists of 5 epochs. The model's results on the validation are following:

- Recall@20 = 0.026
- NDCG@20 = 0.0222

The training dynamics is shown below.

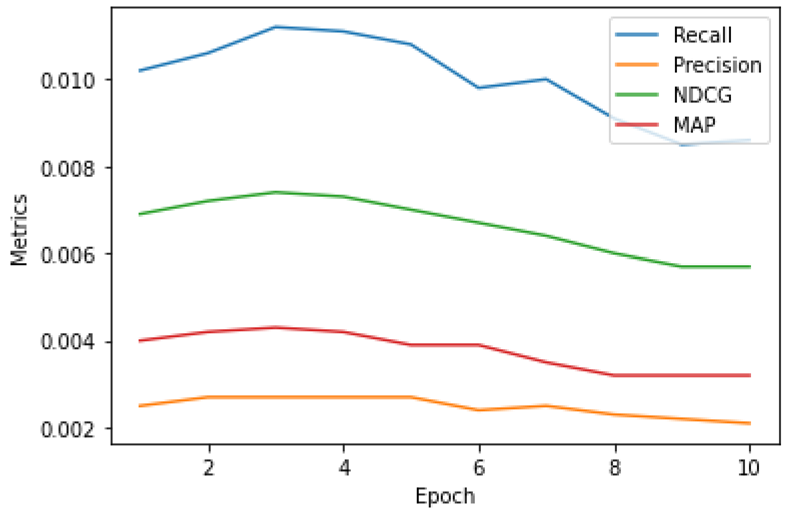




Experiment with LightGCN

This experiment reproduced and benchmarked the LightGCN (Light Graph Convolutional Network) model for recommendation, based on the Kaggle notebook *“LightGCN PyTorch from Scratch”* by Dipanjan Das. The model was trained for 10 epochs on the Yambda dataset, which contains approximately 50 million user–item interactions represented as nodes in a bipartite graph. Training used the Bayesian Personalized Ranking (BPR) loss to learn user and item embeddings through layer-wise propagation without nonlinear transformations.

After 10 epochs, the model achieved the following performance metrics on the test set: Recall = 0.0086, Precision = 0.0021, NDCG = 0.0057, and MAP = 0.0032, with a final training loss of 0.0124. These results demonstrate stable convergence and validate the effectiveness of LightGCN for large-scale implicit feedback data. In our project, we aim to replicate or surpass these benchmark results by refining model tuning, sampling strategies, and graph preprocessing techniques.



Work distribution

- Bulat Sharipov: Writing report, creation of a basic web-service.
- Danil Fathutdinov: Experiment with MB-GCN
- Dinar Yakupov: Experiment with LightGCN

Plan for next week

- Continue to make experiments with other GraphNN models
- Create a uniform evaluation script
- Try implementing some models from scratch and training them
- Integrating new models into recommendation web-service