SJSU SAN JOSÉ STATE UNIVERSITY

Computer Engineering

CMPE 297-02 Final Project - Fall 2022

Collaborative filtering using optimized data structures on Spark

Kin Wo Chang
013783848

supervised by
Professor Gheorghi Guzun

Contents

1 Data pre-processing and other approaches

1.1 Basic Statistics

First, as shown below, I use `pandas`'s library function to read the raw dat file. With header names "uid", "iid", "ratings", "ts" in the dataframe, it makes examining the data easy.

```
 train_raw = pd.read_csv('train.dat', header=None, names=["uid", "iid", "rating",
     "ts"], sep='\t', engine='python')
```

After simple inspection, we have a total of **85724** rating data, with **943** number of unique users and **1659** number of unique items. Ratings are integer values ranging from 1 to 5. Timestamps are given in Unix epoch time.

1.2 Data Transformation

To use our dataset in training the LSH or Matrix Decomposition model, we need to represent each user as a feature factor. That is, the item rating history of that specific user. For this purpose, I chose to transform the above dataframe into a dictionary of key `userid` to a dictionary of key `itemid` to `rating`. Code shown below.

```
u2irmap = {}
i2umap = {}
for _, r in train_raw.iterrows():
    uid, iid = int(r.uid), int(r.iid)
    if uid not in u2irmap:
        u2irmap[uid] = {}
    u2irmap[uid][iid] = int(r.rating)
    if iid not in i2umap:
        i2umap[iid] = set()
    i2umap[iid].add(uid)
```

To save this transformed dataset for easy and efficient access in training our model, I save it in `.json` format. Both the saving and retrieval code are shown below. Note that the `str->int` conversion when loading, because `json` requires string as key.

```
 # Saving
with open("all.json", "w") as outfile:
    json.dump(u2irmap, outfile)
# Loading
```

```
with open("all.json", "r") as content:
    all = json.load(content)
u2irmap = {int(k): {int(mk): int(mv) for mk, mv in v.items()} for k, v in all.
    items()}
```

1.3 Train/Validation Split

We split the training data into train / validation set for cross-validation. This is done on a user level and is only used in the `LSH-NN` model. 90 : 10 split is done, and the resulting training set has 858 and testing set 85. The main function used in this split from **`sklearn`** is shown below.

```
from sklearn.model_selection import train_test_split

s = pd.Series(u2irmap)
train_set, valid_set  = [i.to_dict() for i in train_test_split(s, test_size=0.1)]
train_set = {int(k): {int(mk): int(mv) for mk, mv in v.items()} for k, v in
    train_set.items()}
valid_set = {int(k): {int(mk): int(mv) for mk, mv in v.items()} for k, v in
    valid_set.items()}
```

Code from this section can be found in preprocess.ipynb on Github.

2 User-based LSH with Nearest Neighbor Search

2.1 Prediction Algorithm

I chose user-based LSH over item-based because it is more straight forward. First, the LSH model is built using users feature factors to approximate nearest neighbor search. Then, given a user, user-history of item rating, and a new item, we search the closest neighbors using LSH that has rated this item before, and take the average of neighbors' ratings as the predicted for this user. The above overview may look simple, but there are a lot of nuances in the algorithm. For example, how do we condition the closest neighbors search returned by LSH has to have rating on the predicting item? Based on my research, I have not found a library function that provide this kind of condition on NN search. Hence, there are only two options, 1) pre-process the pool of neighbors so that they all have rated the predicting item, or 2) post-process the returned closest neighbors from LSH and only pick those that have rated the predicting item. Option 2 first seemed most plausible but soon I was bothered by the idea of picking the hyper-parameter numNeighbors. What if the it is set to 10, but the first 10 closest neighbors to user all haven't rated the predicting item?

The above concern regarding option 2 drives me to go with the first options. Therefore, an item to [user] map is created to filter out neighbors before passing it into the main spark MinHashLSH approxNearestNeighbors function (line 8 and 13). Normal spark operations are also used such as Vectors.sparse() and spark.createDataFrame(). The referenced predict_rating function is shown below.

```
1  from pyspark.ml.linalg import Vectors
2  from pyspark.ml.feature import MinHashLSH
3  # expect to have:
4  # maxiid, u2irmap (user-to-item-ratings map), and i2umap (item-to-users map)
5  def predict_rating(itemid, userhistory, num_neighbors=10, num_hash_tables=20, max_iid=1699):
6      if itemid not in i2umap or len(userhistory) == 0: # no rating has given for this item, can't predict
7          return None
8      userDs = [(int(uid), Vectors.sparse(max_iid+1, u2irmap[uid])) for uid in u2irmap if uid in i2umap[itemid]]
9      dfUsers = spark.createDataFrame(userDs, ["uid", "features"])
10     mh = MinHashLSH(inputCol="features", outputCol="hashes", numHashTables=num_hash_tables)
11     model = mh.fit(dfUsers)
12     key = Vectors.sparse(max_iid+1, userhistory) # item_history expect to be a dict {<item>: rating}
13     rows = model.approxNearestNeighbors(dfUsers, key, num_neighbors)
14     ratings = []
15     for r in rows.collect():
16         ratings.append(u2irmap[r["uid"]][itemid])
17     return int(round(sum(ratings) / len(ratings))) if ratings else None
```

2.2 Validation Algorithm

Another big question I had is how do I evaluate my model performance, that is, how do I pass the validation set into the model to get error metrics? The solution I came up with is, for each user's feature vector (items to rating history map), manually hold out one entry from that map. The hold out rating will be the truth rating, and passing this user and its held-out feature vector into our model will give us the predicted rating. Iterating through holding out each item and associate rating from the history map for this user will give us the error metrics for this user. The error metrics algorithm and the core idea of “holding out rating” is implemented in the function `get_user_history_without_item()` shown below in line 3-6.

```
1  import time
2  from math import sqrt
3  def get_user_history_without_item(user_history, itemid):
4      user_history_without_item = user_history.copy()
5      user_history_without_item.pop(itemid)
6      return user_history_without_item
7  rmse_total = []
8  start_time = time.time()
9  for index, uid in enumerate(list(vu2irmap.keys())):
10     print(f"Processing user {uid}, {index+1}/{len(vu2irmap)}")
11     user_history = vu2irmap[uid]
12     rmse_user = []
13     for iid in user_history:
14         expected = user_history[iid]
15         uh = get_user_history_without_item(user_history, iid)
16         predicted = predict_rating(iid, uh)
17         if predicted:
18             rmse_user.append((expected - predicted)**2)
19         else:
20             print(f"No predicted for user: {uid}, item: {iid}")
21     if rmse_user:
22         rmse_total.append(sqrt(sum(rmse_user) / len(rmse_user)))
23     else:
24         print(f"no rmse_user for user {uid}")
25 print("--- %s seconds ---" % (time.time() - start_time))
```

Code from this section can be found in lsh_nn.ipynb on Github.

3 Latent Factor using Matrix Decomposition

Latent factors are factors that can only be indirectly inferred from other observable variables. For our use case, these factors can be the type, cost, quality of items that affects the rating given by users, which isn’t exactly obvious when looking at our dataset. Extracting these latent

factors can help us predict new item ratings for users. A common method for extracting latent factors is Matrix Decomposition. The broad idea is to split the adjacency matrix (in our case the user-item-rating table) into User feature matrix and Item feature matrix. This is achieved by using Stochastic Gradient Decent on minimizing the error of historic item ratings for each user (not the un-rated items). Combining these two matrix back will automatically predicts all missing / unknown item ratings stored in the resulting predicted adjacency matrix.
Despite tireless effort, I couldn't get the SVD algorithm from `Spark` to work. For the interest of time, an algorithm in simple Python is used for this algorithm. Work on both ends can be found in latent_factor.ipynb on Github.

4 Results

4.1 RMSE score

Submitted on the CLP, the RMSE score for the LSH model is 1.0617 with hashTables=20 and nearNeigbhors=10, while the Latent Factor model scores a 1.0037, with steps=300, alpha=0.0002, and beta=0.02. The Latent Factor model seems to be doing better based on my limited testing. However, it is very hard to say which one is better after proper hyper-parameter tuning, such as `numNeighbors` and `numHashTables` in LSH model or `iteration, learning rate, and regularization params` in the Latent Factor model.
Moreover, the validation method mentioned in sec 2.2 yields a RMSE score of 1.021, with same settings as above.

4.2 Training Time

Using the simple `time.time()` in `Python`, building the LSH model (`model.fit`) takes only 0.039 seconds; while running the matrix decomposition with 300 steps take a whooping 704 seconds to complete. This huge gap in training time difference could be because of the naive implementation of matrix decomposition without any optimization. I believe the training time will be greatly improve if I could get `computeSVD` from `Spark` to work.

4.3 Prediction Time

In contrast to the short training time, the LSH-NN model takes 230 seconds to predict all 2154 entries from the test set. Moreover, in contrast to the long training time, the latent factor model takes a mere 0.033 seconds to predict all 2154 entries, mainly because all the items are pre-computed and prediction only takes a matrix look-up. The big contrast of time in both algorithm reflects a trade off between training and prediction efficiency when building models.

5 References

- Ahmet& Tevfik: Real-time recommendation with locality sensitive hashing
- Dennis Chen: Recommender System — Matrix Factorization
- Shashwat Tiwari: Crafting Recommendation Engine in PySpark
- Spark: LSH
- Spark: Dimensionality Reduction -
- Spark: MinHashLSHModel
- Spark: Vector