Computational tools for data science: Comparing Recommendation Systems for MillionSongsDataset

Alejandra Navarro Castillo
Technical University of Denmark
s222712@dtu.dk

Matija Šipek
Technical University of Denmark
s222736@dtu.dk

Angeliki Kallia Spentza
Technical University of Denmark
s212881@dtu.dk

Andreas Lau Hansen
Technical University of Denmark
s194235@student.dtu.dk

Andrés Sebastián Costa
Technical University of Denmark
s214133@student.dtu.dk

December 2nd 2022

Abstract

Recommendation systems are in these days a very helpful tool because of the expansion of digital music, and the abundance of songs and artists. In this paper we implement different approaches for recommendation systems; a Baseline approach, a Content-based (CB) approach, an Apriori approach, and Collaborative filtering (CF). We then evaluate and compare each approach and discuss the results and limitations.
Code and data available at `https://github.com/matijasipek/comp_tools`.

1 Pre-processing and Efficient Data reading (Matija)

We found the **MillionSongsDataset** [BMEWL11] which contains attributes for one million songs (280GB). The raw data is in H5 format, a hierarchical data format (HDF), which is a well-organized scientific format for retrieval and large-scale analysis. When first trying to read the data we look at **Hadoop/MapReduce** in order to parallelize computations so they can scale in a way that is tolerant to hardware faults. However, since the data is placed on a single computer and not a Distributed File System (DFS), we settle for a simpler version. We optimize all steps, and use python **chunksize** and **interator** functions to split loading the data into smaller chunks in order not to overflow memory. Thus each chunk takes up 1/100 (2.8GB) space. The initial form of the data frame attributes was **artist id, song id, song name**, and **artist_terms**.

Secondly, the **Taste Profile subset**, was retrieved from the same source, with the form **userID, songID, play count**. The songID allows us to connect the two tables and use it as a 'foreign key'.

In our case we are matching two different datasets by the song id. Thus, the first step was to remove all redundant, duplicate, and incomplete data. For the songs, we used a subset (1% of 280GB) of 10 000 songs, with roughly 48mil. users. Matching the user data with the song data, we ended up with 3195 songs and ∼400k users for our recommendation system.

2 Recommendation system

(Alejandra) In a recommendation system we mainly have two classes of entities: *users* and *items*. This type of systems are designed to predict user's responses to options. There are a lot of ways this can be achieved; in this report we present the methods we have used to get recommendations for users [JL19].

Problem Definition: Given a database of users $U = u_1, u_2, ..., u_M$ and a collection of songs $C = c_1, c_2, ..., c_N$ that the users have listened to, let user $u_i \in U$ be a target user and $C_{u_i} = c_1, c_2, ..., c_p \in C$ be the collection of songs he likes. The task is to predict which other songs $C_x \in C$ the target user might like.

2.1 Content-based approach

2.1.1 Content based item similarity (Alejandra)

In a content-based system (CB), the focus is on the properties of the items. The main idea of the CB approach is then to recommend the items to a user that are similar to previous items rated highly by this user. In the case of our dataset we will consider ratings as the number of plays of a song by a user.

Hence, we must create an *item profile* for each song in our dataset. We understand a *profile* as a numerical representation so that we can compute a sort of "similarity" between songs afterwards. In our case, in order to represent each song numerically, we have extracted the set of the `artist_terms` column. Then, each song is represented by a one-hot-encoding[1] technique.

Once we have the vectors describing the items, we then need to create vectors with the same components that describe each user's preferences: *user profiles.* In order to calculate these we need to use the **Taste Profile subset** to know the songs already played by each user. The vector representation of the user profile of each user is then calculated as follows: weighted average of songs the user has listened to.

Lastly, we can obtain the recommendations, i.e. the recommending items for each user. We estimate the degree to which a user would prefer an item by computing the distance[2] between the user's and the item's vectors. In our system we have computed the distance between the *user profile* and the *item profile* and selected the 10 top songs with smaller distance.

2.1.2 Content based cluster approach (Alejandra)

Another method we have implemented to get recommendations based on content (features of the songs) is an approach through clustering techniques on the songs.

We have clustered songs with the K-means algorithm. Then, since we have computed the *user profile* for every user in the previous section, we can compute the distance of each user to the centroid of each cluster in order to obtain the cluster to which each user is "assign" to. The final recommendation to a user will be a random set of songs of the assigned the cluster.

2.1.3 Other Content based cluster approach/separate notebook/ (Andrés)

Recommendation systems are often heavy computationally, as in the previous presented content-based method where it was necessary to calculate the similarity between all songs in the data set in order to make a recommendation. But when a user wants a song recommended this should preferably be done in a short time-span even if the user suddenly changes their taste in music. Therefore we wanted to explore the benefits of using clustering. The clustering method has been done with the objective of reducing computational time and creating a playlist suggestion option.

In order to create the cluster we use a K-means clustering algorithm using sklearn which is known for working well with high amounts of data. In order to make a good K-means algorithm and clustering analysis two different methods to finding the optimal K was used - the Silhouette method and the Elbow method. When using these method we see that the method fails to converge within 50 clusters. To improve on this additional cleaning of the data was attempted. For the 3500 song almost as many different terms where found making the cases of each different term extremely rare. Additional cleaning of outliers and complete similar terms could further increase data quality. Finally the recommendation was based on 46 clusters decreasing the computation of recommending a song significantly. Finally

[1] *One-hot-encoding*: we convert each categorical value (each term in all artist terms) into a new categorical column and assign a binary value of 1 or 0 to those columns. We will assign a 1 if the song has the term and a 0 otherwise.

[2] Various distance measures can be used for this step. In our project, we have used the euclidean distance between two points but other measures like cosine similarity or Jaccard similarity can be used.

two recommendation where made.The first one suggesting a playlist to the user, and the second one, recommending the 10 songs more similar to the users taste.

2.2 Collaborative Filtering

2.2.1 Introduction (Angeliki)

In collaborative filtering (CF) the most important notion is that of neighborhood creation. Similarity between items or between users is calculated using the preferred distance measure (e.g. Cosine, Pearson Correlation, etc). We start by building the utility matrix which contains users or items as vectors. The values in the utility matrix, for each user-item pair, represent what is known about the degree of preference of that user for that item. In our study case, the matrix is sparse, meaning most values are "unknown", and the preference is represented by play counts of songs.

The goal here is to predict the blanks in the utility matrix. In reality, it is not feasible to predict every blank entry for the whole dataset as this is computationally too demanding. Following, we adopted three approaches: (a.) compute the adjusted cosine similarity matrix, (b.) apply brute force nearest neighbor search, and finally (kNN) (c.) locality sensitive hashing.

2.2.2 Utility matrix based approach (Andreas L.)

Within this genre there are two different ways to compute the utility matrix: A user to user approach and an item to item based approach. Songs are recommended to a user by considering their play counts compared to the other users and finding other users that have listened to many of the same songs but some songs that the query user has not listened to. Table 1 shows an example of a small utility matrix. Elements in the matrix are counts of how many times each user has listened to each song. This sparse matrix quickly becomes unfeasible to do computations on, since the dimensions of the matrix is $\mathbb{R}^{n_{\text{users}} \times m_{\text{songs}}}$, where $n \gg m$. It is evident that table 2 is the transpose of table 1. Similarity is thus found within items instead of users, which is more meaningful and is why this approach is used.

		Songs			
		I1	I2	I3	I4
Users	U1		3		4
	U2			1	
	U3		8	2	
	U4	5	4		12

Table 1: Example of a utility matrix used in user-user collaborative filtering with 4 users and 4 songs

		Users			
		U1	U2	U3	U4
Songs	I1				5
	I2	3		8	4
	I3		1	2	
	I4	4			12

Table 2: Example of utility matrix used in collaborative filtering in the item-item based approach

2.2.3 LSH in item-based Collaborative Filtering (Angeliki)

Locality sensitive hashing (LSH) is an approximate similarity search approach based on hash functions [GIM99]. Building a recommendation system with LSH has a complexity of $O(n)$ and thus improves on the KNN method ($O(n^2)$) tremendously.

We propose an approach inspired by Liang H. et al. 2014, where for a given user u_i we compute a hash signature for each item he has liked [LDW14]. The purpose of the hash functions (H) are to convert an item vector (a song), c_i to a small signature $H(c_i)$ that takes up less memory. Then, if similarity between two songs, $\text{sim}(c_1, c_2)$ is high, the $\Pr(H(c_1) == H(c_2))$ will be also high and the opposite, if $\text{sim}(c_1, c_2)$ is low $\Pr(H(c_1) == H(c_2))$ will be also low. The hashed item is then allocated to a bucket. Items that exist in the same bucket (placed during the model building phase using the database) are neighbors of the query item and the top N items are selected as recommendations for the target user. This process is repeated for all items user u_i has liked, and recommendations are returned as output.

2.3 Apriori & tf–idf (Matija)

This approach combines both apriori [Ras18] and tf-idf techniques. Due to the unique form of our data, the artists' term column is formed with genres eg. [blue-eyed soul, pop rock, british ...]. We are able to use both techniques to create a recommendation system that aims for all parts of the distribution, see fig. 1. In the apriori problem we form the artist genres as the items. And, in the second part, user data; where each user has a list of songs and the number of times played, and we extract the tags for each song to get the tf-idf of the most popular tags in the users' arsenal.

2.3.1 Part A: Apriori

Apriori algorithm usually follow three steps: Support, Confidence and Lift. The support measures the proportion where a tag appears in tag sets, the confidence measures the combination of pairs (if there is rock is there also blues), and finally lift where rock is rises the probability of having blues as well. In the first step we extract the tags into a list and count the frequency for each item. Next, we find the frequency of pairs tags, and, if support is above a certain threshold **(minSupport = 0.05)**, put into a new candidate set. Finally, we can find frequent tag sets for genres, we decided to take sets of **length 5** as a basis for a further recommendation since it gave us a large enough pool (1434 tag sets) while preserving distinctness between songs.

2.3.2 Part B: tf-idf (term frequency-inverse document frequency)

TF-IDF gives us the relevance of a word to a document, in a set of documents. The first part is TF = how frequent a term is within a document, and the second part IDF how rare a word is across a set of documents. We calculated tf-idf for each user. Firstly, we took a user and found all songs and number of times he played each song. And, since each song has a list of tags (same as Apriori dataset), we were able to calculate the weighted average of tags for each user. Thus, the result for each user would be something in the area; **rock: 166.28, pop: 110.85, alternative: 89.30, metal: 61.58 etc.**

2.3.3 End results

As can be seen from the Figure 2 we try to find buckets which do not contain only the genres that the user prefers, but the 'subgenre' of the liked songs. The terms found by TF-IDF looks into the "subgenre" buckets, while maintaining the users preference since we are looking for a match of at least 4. For example if a song from subgenre buckets gives at least 6 matches, with the terms form TF-IDF, we recommend it. This approach allows us to not only recommend the songs that the user has already listened to but also the songs which are on the thinner part of the "long-tail" distribution pattern.

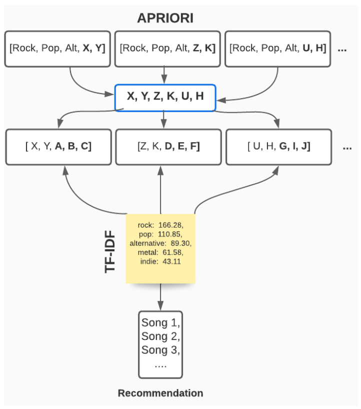

Figure 1: Combination of Apriori and TFIDF to recommend songs

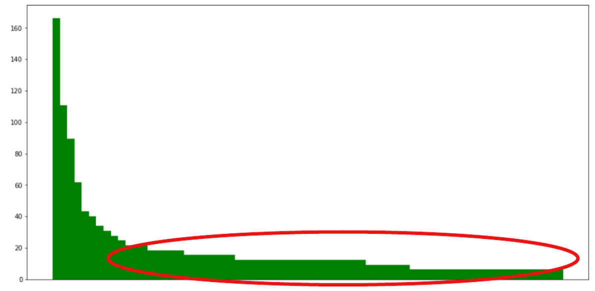

Figure 2: Songs' tag distribution with red circle as the aim of our recommendation system

3 Results (Alejandra & Andreas)

We have used the metrics intra-list similarity and average similarity as evaluation methods to compare the performance of the different methods used. "Intra-list similarity" is the average cosine similarity of all items in a list of recommendations. Intra-list similarity can be calculated for each user, and averaged over all users in the test set to get an estimate of intra-list similarity for the model. "Average similarity" consists of calculating the euclidean distance of the songs recommended to a user and the user profile. Then, we average these distances. If we compute this measure for every user, we then get the score for our recommendations model.

	Intra list similarity	Average similarity
Baseline	0.1711	6.2242
CB with item similarity	0.5169	**3.1304**
CB with clustering	0.1904	6.0843
CF with Adj. Cos.	0.1729	5.0472
CF with KNN	0.1760	5.9723
CF with LSH	0.1668	6.1033
Apriori	**0.5599**	4.3650

Table 3: Results of the different methods using the two different metrics, best result in **bold**.

The content based and apriori methods have significantly higher intra list similarity of ≈ 0.5 compared to the other methods, which are all roughly equal to the baseline method with a score of ≈ 0.17. Additionally, when comparing the average similarity the content based approach receives a score of 3.13 while the clustering and vanilla CF methods are worse, with scores ≈ 6.08 and ≈ 5.05 respectively. The apriori method is again almost as good as the content based method. It is noteworthy that the CF method with KNN is worse than the vanilla version.

4 Discussion & Conclusion (Andreas)

The LSH and clustering methods were extremely fast, while the content based and vanilla collaborative filtering methods were reasonable. However, the apriori and CF KNN methods were already close to infeasible at 5k users. Especially the CF methods without LSH suffer from extremely high memory usage, since at some point you need to calculate the pairwise distances to all other users. For us this became unfeasible at around 30k users, for the full dataset we would need 1.5 TB ram just to hold the distances between users. Meanwhile, the LSH and apriori methods are much more efficient due to the use of hashing. Overall the best method presented here in terms of speed versus usefulness is probably the content based method.

All the methods mentioned here suffer to some degree of the *cold start problem* where if a user has only listened to a very low number of songs they do not have a "profile" in that case these methods are not especially useful approaches. There are many more aspects to recommendation not explored here, for example if the methods just recommend the most popular songs or the merging of two different techniques to get better results.

5 Contributions

Code and data https://github.com/matijasipek/comp_tools

Report

	Alejandra	Matija	Angeliki	Andreas L.	Andrés C.
Pre-processing and data reading		100%			
Content-based item similarity	100%				
Content-based cluster	100%				
Other content-based cluster approach					100%
Intro CF			100%		
Utility matrix approach				100%	
LSH in item-based CF			100%		
Apriori and tf-idf		100%			
Results and discussion	20%			80%	

Code in Jupyter Notebook

	Alejandra	Matija	Angeliki	Andreas L.	Andrés C.
Pre-processing and data reading		100%			
Content-based item similarity	100%				
Content-based cluster	100%				
Custom Kmeans algorithm				100%	
CF adjusted cosine based			100%		
CF kNN based			100%		
CF LSH based			100%		
Apriori and tf-idf		100%			
Evaluation of models	80%			20%	
Cleaning and formating of notebook	50%			50%	
Extra notebook CB clustering methods					100%

References

[BMEWL11] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[GIM99] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *Vldb*, 99(6):518–529, 1999.

[JL19] Jeffrey D. Ullman Jure Leskovec, Anand Rajaraman. *Mining of Massive Datasets.* Cambridge University Press., 2019.

[LDW14] H. Liang, H. Du, and Q. Wang. Real-time Collaborative Filtering Recommender Systems. *AusDM*, pages 227–231, 2014.

[Ras18] Sebastian Raschka. Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. *Journal of Open Source Software*, 3(24):638, 2018.