

---

# GENERATING ARTIFICIAL CORE USERS FOR INTERPRETABLE CONDENSED DATA

---

A PREPRINT

**Amy Nesky**

Computer Science and Engineering  
University of Michigan, Ann Arbor  
Ann Arbor, MI 48109  
anesky@umich.edu

**Quentin F. Stout**

Computer Science and Engineering  
University of Michigan, Ann Arbor  
Ann Arbor, MI 48109  
qstout@umich.edu

February 9, 2021

## ABSTRACT

Recent work has shown that in a dataset of user ratings on items there exists a group of Core Users who hold most of the information necessary for recommendation. This set of Core Users can be as small as 20 percent of the users. Core Users can be used to make predictions for out-of-sample users without much additional work. Since Core Users substantially shrink a ratings dataset without much loss of information, they can be used to improve recommendation efficiency. We propose a method, combining latent factor models, ensemble boosting and K-means clustering, to generate a small set of Artificial Core Users (ACUs) from real Core User data. Our ACUs have dense rating information, and improve the recommendation performance of real Core Users while remaining interpretable.

**Keywords** Recommender Systems · Core Users · Synthetic Data Generation

## 1 Introduction

Recommendations Systems improve the user experience by providing personalized, curated recommendations in a large and complex information space. Collaborative Filtering methods exploit community data to uncover correlations between users and items that can be used to make recommendations. Within collaborative filtering, latent factor models are considered state-of-the-art; these models approximate highly redundant data with low rank matrix decompositions (Hu et al., 2008; Koren, 2008; Koren et al., 2009; Shi et al., 2014; Aggarwal, 2016).

Unfortunately, many matrix factorizations methods are transductive and cannot be leveraged to make predictions for users outside of the original training set (Aggarwal, 2016). Orthonormal matrix factors can be used more easily to make out-of-sample predictions because the matrix factors capture geometric traits of the item and user spaces. However, obtaining orthonormal factors can be expensive for large datasets.

The majority of research in recommender systems focuses on developing new collaborative filtering and hybrid methods, which are often highly optimized for a specific application. There is a smaller body of research looking at identifying the most useful users who carry most of the relevant information, and separating out those users as Core Users for making recommendations. With a smaller core set of users, making out-of-sample predictions becomes a reasonable task. Reducing a dataset size without much loss of information improves recommendation efficiency as well as storage costs; numerous fields outside of recommender systems would benefit from this ability.

However, available Core User methods have limited representation ability, in that they are selected from existing user data. In other words, the recommendation success of Core Users is bounded by quality of user data available. Improving the recommendation accuracy of Core Users makes data abstraction more effective for applications in data augmentation, bots mimicking population behavior, data mining, privacy, statistics and many more. In this paper, we develop a method of generating Artificial Core Users (ACUs) that improves the recommendation accuracy of real Core Users. We combine latent factor models, ensemble boosting and K-means clustering, to generate a small set of Artificial

Core Users (ACUs) from real Core User data. Our ACUs incur a small amount of additional memory storage when compared to real Core Users, but remain a reduction in memory storage compared to the original dataset. Artificial Core Users improve the recommendation accuracy of real Core Users while remaining good centroids for the complete recommendation dataset. Since ACUs act as good centroids for the complete dataset, ACUs blend in well with the real dataset even though they are generated artificially. But unlike real Core Users, ACUs have complete ratings on all items, providing more immediately interpretable information to scientists.

## 2 Related Work

The inductive matrix completion problem assumes that a ratings matrix is generated by applying feature vectors to a known low-rank matrix (Jain and Dhillon, 2013; Zhang et al., 2018). This is relevant for making out-of-sample recommendations if we assume that the latent factors contain some ground truth about the dataset that applies to out-of-sample users. As mentioned above, latent factors produced by most methods are sadly transductive (Aggarwal, 2016). To combat this, some have worked on improving the efficiency of using a singular value decomposition in recommender systems (Sarwar† et al., 2002).

Using clustering is one of the earliest attempts to decrease the number of users needed to make recommendations (Aggarwal, 2016); rating predictions can be made using only information from the relevant cluster. Alternatively, Zeng et al. (2014) study the relevance of different users and find that there exists an “information core” made up of some key users. They found that the number of the Core Users is around 20% of the entire dataset, and that the recommendation accuracy produced by only relying on the Core Users can reach 90 percent of that produced using every user in the dataset. Zeng et al. (2014) use a generalized K-nearest Neighbor algorithm using various relevancy metrics to measure ‘nearness’. They ran experiments using degree-based, resource-based and similarity-based measures, to select their Core Users; all of these measures are graphical in nature. Since this work a few other methods for selecting Core Users have emerged. Li et al. (2016) use a long-tail-distribution-based measure to select their core uses. Cao and Kuang (2016) introduced a new measure to identify Core Users based on trust relationships and interest similarity; this work extends beyond graphical knowledge to include the semantic meaning of items. Kuang et al. (2017) use a combination of the measures proposed in (Cao and Kuang, 2016) and (Li et al., 2016).

Recently, deep learning has made an appearance in recommender systems either in the form of integration models or neural network models (Zhang et al., 2017). Integration models use neural networks to uncover features in auxiliary information, like item descriptions (Wang et al., 2015, 2016), user profiles (Li et al., 2015) and knowledge bases (Zhang et al., 2016). The uncovered features are then incorporated into a collaborative filtering framework to produce hybrid recommendations. Neural network models on the other hand perform collaborative filtering directly via modeling the interaction function between users and items (Sedhain et al., 2015; Strub et al., 2016; Wu et al., 2016; He et al., 2017; Li et al., 2018). Using deep learning to make recommendations means the models are better equipped to recognize nonlinear relationships in the data, but it also means that the models inherit all the training difficulties neural networks face compounded by the difficulties of working with sparse data.

There is a small body of work that injects fake users into a recommendation system either for adversarial goals (OMahony et al., 2002; Lam and Riedl, 2004; Christakopoulou and Banerjee, 2018) or utilitarian data augmentation goals (Sarwar et al., 1998). Typically, the fake users are hand-coded, but (Christakopoulou and Banerjee, 2018) create adversarial fake user profiles for a recommendation system using generative adversarial nets. Sarwar et al. (1998) on the other hand used simple content filtering bots to generate a few users with dense ratings to improve the recommendations. Tackling the new user, or cold start, problem is an issue in recommender systems that has brought about some work on determining which item preferences and item features are most informative (Rashid et al., 2002, 2008; Seroussi et al., 2011).

## 3 Methodology

We propose an offline technique to generate a small set of Artificial Core Users (ACUs) who’s dense ratings matrix can be stored in condensed, low order form and used to make recommendations for out-of-sample users without much additional work. This method uses latent factor models, ensemble boosting and K-means clustering to learn a small group of artificial users who’s feature vectors capture correlations in the full dataset. This is a collaborative filtering method that uses an existing sparse ratings matrix for a set of users and items.

The algorithm requires a set of training users represented by their ratings on a given set of items, which will be used to update ACU ratings during learning. To measure the abstraction of the ACUs, we split the dataset into two mutually exclusive sets: testing users and training users. Let  $R$  be an  $m \times n$  sparse ratings matrix for  $m$  training users and  $n$

items.<sup>1</sup> The ACU set size will be small relative to the size of the dataset; if  $s$  is the number of ACUs, we pick  $s$  such that  $s \ll m$ . Let  $R_{ACU}$  denote the  $s \times n$  ratings matrix for our ACUs. There are a number of ways one could initialize the ACU ratings; in our work, ACU ratings will be initialized from Core User ratings.<sup>2</sup>

The training algorithm is given in Algorithm 1. We trained row blocks of  $R_{ACU}$  together using batches of training users. By learning ACUs in blocks, we incorporate boosting results and reduce our workload (Schapire, 1990; Breiman, 1998). Algorithm 1 learns orthonormal decompositions for each block, leveraging the relevance of orthonormal decompositions to out-of-sample relationships.<sup>3</sup> Let  $R_{ACU}^{(b)}$  denote the  $b^{th}$  row block in  $R_{ACU}$ , and  $R_i$  denote the sparse ratings matrix of the  $i^{th}$  batch of training users. For simplicity, we assume that each block of  $R_{ACU}$  has the same number of rows and similarly each training batch has the same number of users. We divide  $R_{ACU}$  into  $\zeta$  equal sized row blocks such that for  $b \in \{0, \dots, \zeta\}$   $R_{ACU}^{(b)}$  is a  $\left(\frac{s}{\zeta} \times n\right)$  matrix. Let  $I$  denote the training batch index set such that for  $i \in I$ ,  $R_i$  is a  $\left(\frac{m}{|I|} \times n\right)$  matrix. The variables  $\alpha$  and  $\beta$  are learning rates, while the variables  $\lambda$  and  $\gamma$  are regularization coefficients. Lines 15 and 20 in Algorithm 1 are derived from stochastic gradient descent algorithms. Line 18 updates the rows of  $U_{ACU}$  as the  $\frac{s}{\zeta}$ -means of  $\frac{m}{|I|}$  row vectors in  $U_i$ , which is described in detail in Algorithm 3. Embedding Algorithm 3 inside Algorithm 1 effectively trains the Artificial Core User user-space matrix,  $U_{ACU}$ , with Mini-Batch K-means (Sculley, 2010). By incorporating Algorithm 3 into the learning process we expect to keep the resulting Artificial Core User ratings matrix interpretable. That is, we expect the resulting Artificial Core User ratings to resemble that of real users, but provide complete rating information in place of sparse data. As with any learning method, the amount of regularization will be problem dependent; one may find that they can get away with  $\lambda, \gamma = 0$  because lines 16 and 23-24 in Algorithm 1 have a regularizing effect. Lines 23-24 ensure that we don't stray to far from an orthonormal decomposition during learning. The while loop on line 10 takes a simplistic approach to the inductive matrix completion problem given a set of features; this subprocess could likely be improved by existing work Jain and Dhillon (2013); Zhang et al. (2018).

Algorithm 1 should be run offline to produce a set of Artificial Core Users. After which, one can use  $V_{ACU}S_{ACU}$  to make faster recommendations for out-of-sample users. One may also be interested in using  $R_{ACU}$  as a condensed version of their dataset.

This process resembles the way one would train a neural network and shares similarities with neural network models (Sedhain et al., 2015; Strub et al., 2016; Wu et al., 2016; He et al., 2017; Li et al., 2018). Neural network models generally use autoencoders. One layer Neural Collaborative Autoencoders with no output activation can be reformulated to parallel matrix factorization (Li et al., 2018). Unlike Algorithm 1, existing these models are not concerned with interpretability.

We evaluate our work using two metrics: item vectors testing error and sparse K-means error. To measure the item vectors testing error, we use Algorithm 2 on an independent set of testing users. Algorithm 2 is a simplistic recommender model that tries to predict missing user ratings. It decomposes the row blocks of  $R_{ACU}$  into orthogonal components using a singular value decomposition. Then, for a set of real testing users, Algorithm 2 optimizes a set of unit vectors to accompany the item vectors extracted from the decomposition of  $R_{ACU}$  - there is no new learning done in the item vectors. In this way, Algorithm 2 evaluates the generality of the item vectors extracted from  $R_{ACU}$  and the potential for high quality recommendations using a more elaborate learning model. It should be noted that, as a simplistic recommender model, Algorithm 2 produces far from state-of-the-art recommendation results, but is useful for our purposes of evaluating whether Artificial Core Users better capture the information in a recommendation dataset than real Core Users do. To compute the sparse K-means error of  $R_{ACU}$  we make a few modifications to Algorithm 3 using a sparse first input matrix; this results in Algorithm 4. Algorithm 4 estimates how well the Artificial Core Users represent an average collection of real users,  $R_{ACU}$ , by measuring how well the ACUs serve as centroids for our real testing users.

<sup>1</sup>We did choose to normalize the variance and mean center the rows of  $R$  before proceeding; any adjustments made here can be accounted for at recommendation time.

<sup>2</sup>As with any learning process, one can see a lot of improvement by selecting the right initialization. There is room to try supplementing Core User ratings to improve the initialization of  $R_{ACU}$ .

<sup>3</sup>Computing the singular value decomposition of a large matrix is known to be time consuming, so learning in blocks saves quite a bit of time.

**Algorithm 1** Generate\_ACUs( $R$ )

---

```

1: Let  $m$  be the number of users (row dimension of  $R$ ) and  $n$  be the items (column dimension of  $R$ ).
2: Initialize constants  $\alpha, \lambda, \beta, \gamma$ . Initialize matrix  $R_{ACU}$  with dimensions  $s \times n$ .
3: while not done do
4:   for  $b = 0$  to  $\zeta$  do
5:     Find orthonormal  $\left(\frac{s}{\zeta} \times k\right)$  matrix  $U_{ACU}^{(b)}$ , orthonormal  $(n \times k)$  matrix  $V_{ACU}^{(b)}$  and diagonal  $(k \times k)$  matrix
      $S_{ACU}^{(b)}$  such that  $U_{ACU}^{(b)} S_{ACU}^{(b)} \left(V_{ACU}^{(b)}\right)^\top \approx R_{ACU}^{(b)}$  and  $k \leq \min\left(\frac{s}{\zeta}, n\right)$ .
6:     Set  $V_{ACU}^{(b)} = V_{ACU}^{(b)} S_{ACU}^{(b)}$ .
7:     Select  $i$  randomly from training user batch index set  $I$ .
8:     Initialize  $U_i$ .
9:      $j = 0$ .
10:    while not done do
11:      Compute sparse  $E = R_{we} - U_{we} \left(V_{ACU}^{(b)}\right)^\top$  for the specified entries of  $R_i$ . Other entries of  $E$  remain
      zero.
12:      if not done then
13:        if  $j \bmod 2 = 0$  then
14:           $U_{we} \leftarrow (1 - \beta\gamma)U_{we} + \beta E V_{ACU}^{(b)}$ .
15:          Normalize the columns of  $U_i$ .
16:        else
17:          if  $j \bmod 4 = 1$  then
18:            Means_Update( $U_i, U_{ACU}^{(b)}$ ).
19:          else
20:             $V_{ACU}^{(b)} \leftarrow (1 - \alpha\lambda)V_{ACU}^{(b)} + \alpha E^\top U_i$ .
21:          end if
22:        end if
23:         $R_{ACU}^{(b)} = U_{ACU}^{(b)} \left(V_{ACU}^{(b)}\right)^\top$ .
24:        Find orthonormal matrices  $U_{ACU}^{(b)}$ ,  $V_{ACU}^{(b)}$  and diagonal matrix  $S_{ACU}^{(b)}$  such that
         $U_{ACU}^{(b)} S_{ACU}^{(b)} \left(V_{ACU}^{(b)}\right)^\top \approx R_{ACU}^{(b)}$ .
25:        Set  $V_{ACU}^{(b)} = V_{ACU}^{(b)} S_{ACU}^{(b)}$ .
26:         $j = j + 1$ .
27:      end if
28:    end while
29:  end for
30: end while

```

---

**Algorithm 2** Item\_Vectors\_Testing\_Error( $R, R_{ACU}, \zeta$ )

- 
- 1: Let  $m$  be the number of users (row dimension of  $R$ ),  $n$  be the items (column dimension of  $R$  and  $R_{ACU}$ ) and  $s$  be the number of ACUs (row dimension of  $R_{ACU}$ ).
  - 2: Initialize constants  $\beta, \gamma$ .
  - 3: **for**  $b = 0$  to  $\zeta$  **do**
  - 4: Find orthonormal  $\left(\frac{s}{\zeta} \times k\right)$  matrix  $U_{ACU}^{(b)}$ , orthonormal  $(n \times k)$  matrix  $V_{ACU}^{(b)}$  and diagonal  $(k \times k)$  matrix  $S_{ACU}^{(b)}$  such that  $U_{ACU}^{(b)} S_{ACU}^{(b)} \left(V_{ACU}^{(b)}\right)^\top \approx R_{ACU}^{(b)}$  and  $k \leq \min\left(\frac{s}{\zeta}, n\right)$ .
  - 5: **end for**
  - 6: Let  $V_{ACU} = \left[V_{ACU}^{(0)} \cdots V_{ACU}^{(\zeta)}\right]$ .
  - 7: Aggregate  $S_{ACU}^{(b)}$  for  $b = 0$  to  $\zeta$  along diagonal blocks to form the diagonal matrix  $S_{ACU}$ .
  - 8: Set  $V_{ACU}^{(b)} = V_{ACU}^{(b)} S_{ACU}^{(b)}$ .
  - 9: Randomly select 80% of the nonzero entries in  $R$  as training entries and let the other 20% be the probe entries.
  - 10: Define  $R^{(T)}$  as the sparse matrix made up of the training entries of  $R$ , and define  $R^{(P)}$  as the sparse matrix made up of the probe entries of  $R$  such that  $R = R^{(T)} + R^{(P)}$ .
  - 11: Initialize  $U$ .
  - 12: **while** not done **do**
  - 13: Compute sparse  $E = R^{(T)} - UV_{ACU}^\top$  for the specified entries of  $R^{(T)}$ . Other entries of  $E$  remain zero.
  - 14: **if** not done **then**
  - 15:  $U \leftarrow (1 - \beta\gamma)U + \beta EV_{ACU}$ .
  - 16: Normalize the columns of  $U$ .
  - 17: **end if**
  - 18: **end while**
  - 19: Compute sparse  $E = R^{(P)} - UV_{ACU}^\top$  for the specified entries of  $R^{(P)}$ .
  - 20: Return the average absolute value of an entry in the  $E$  for the specified entries of  $R^{(P)}$ .
- 

**Algorithm 3** Means\_Update( $A, B$ )

- 
- 1: Initialize lists  $L_i$  for  $we = 0$  to the row dimension of  $B$ .
  - 2: **for**  $r = 0$  to the row dimension of  $A$  **do**
  - 3:  $\min\_val = \infty, \min\_index = 0$ .
  - 4: **for**  $l = 0$  to the row dimension of  $B$  **do**
  - 5: **if** The distance between the  $r^{th}$  row of  $A$  and the  $l^{th}$  row of  $B$  is less than  $\min\_val$  **then**
  - 6:  $\min\_val =$  this distance.
  - 7:  $\min\_index = l$ .
  - 8: **end if**
  - 9: **end for**
  - 10:  $L_{\min\_index}.push(r)$ .
  - 11: **end for**
  - 12: **for**  $l = 0$  to the row dimension of  $B$  **do**
  - 13: **if** list  $L_l$  is non-empty **then**
  - 14: Set the  $l^{th}$  row of  $B$  to the weighted average of the rows of  $A$  with indices stored in list  $L_l$ .
  - 15: **end if**
  - 16: **end for**
-

**Algorithm 4** Sparse\_Means\_Error( $R, R_{ACU}$ )

---

```

1: avg_error = 0.
2: entry_count = 0.
3: for  $r = 0$  to the row dimension of  $R$  do
4:   min_val =  $\infty$ , min_index = 0.
5:   for  $l = 0$  to the row dimension of  $R_{ACU}$  do
6:     if The sparse distance between the  $r^{th}$  row of  $R$  and the  $l^{th}$  row of  $R_{ACU}$  is less than min_val for specified
       entries of  $R$  then
7:       min_val = this distance.
8:       min_index =  $l$ .
9:     end if
10:  end for
11:  for sparse entries in the  $r^{th}$  row of  $R$  do
12:    entry_count += 1.
13:    temp_err = absolute difference between entry of the  $r^{th}$  row of  $R$  and the corresponding entry in the
       min_index row of  $R_{ACU}$ .
14:    avg_error += (temp_err - avg_error) / entry_count.
15:  end for
16: end for
17: Return avg_error.

```

---

## 4 Experimental Results

Our main objective in our experiments is comparability across methods rather than state-of-the-art performance. To make the error on the probe entry sets comparable when testing with Algorithm 2, we aimed for similar errors on the training set; to accomplish this goal we stopped the algorithm early when necessary.

All experiments are run on the Bridges’ NVIDIA P100 GPUs through the Pittsburgh Supercomputing Center. We ran experiments using the MovieLens ml-20m dataset (Harper and Konstan, 2015). We normalized the variance and mean centered the rows of the dataset ratings matrix before proceeding.<sup>4</sup>

We will compare our work for generating ACUs to existing methods for selecting Core Users from (Zeng et al., 2014; Li et al., 2016; Cao and Kuang, 2016; Kuang et al., 2017). All of the methods for finding Core Users tested here are derived from the K-nearest neighbor algorithm. These methods use some metric to determine the pair-wise similarity between all existing users. Then, for each user, a list of the top-K most similar users is generated; from these combined lists the Core Users are selected. In our Core User experiments, we collect the top-50 most similar users for each user. Existing methods differ in their pair-wise similarity metric for users, and in their selection method within the compiled top-K most similar user lists.

Recall that the cosine similarity of two vectors  $A$  and  $B$  is  $(A \cdot B) / (||A|| \cdot ||B||)$ . In all of our Core User experiments, the pair-wise similarity between all existing users will be calculated in one of two ways: it will either be the cosine similarity of the users’ ratings vectors, or it will be the cosine similarity of their boolean vectors where their boolean vectors indicate only whether or not an item has been rated - not how well the user liked it.

Once the lists of the top-K most similar users to each user have been generated, one can either count the frequency of a given user’s appearance in the top-K lists and take the most frequent users as the Core Users, or one can weight a user’s appearance in a list by the inverse of the rank within the list that user appeared; the first way we will refer to as frequency-based and the second way we will refer to as rank-based.

The final variant is whether or not to consider ‘hidden’ ratings in the cosine similarity of users. Without considering hidden ratings, two users with no overlapping rated items would have zero similarity, but if the users have rated items that are similar to one another this metric seems insufficient. Cao and Kuang (2016); Kuang et al. (2017) consider semantic relationships between items to determine item similarity, here we will use the cosine similarity of the item vectors where the item vectors are the rating data from all of the users for the given item.<sup>5</sup> After computing the item similarity, a missing rating may be substituted for with a weighted average of similar rated items, where the item similarity can be used as the weight.

<sup>4</sup>Any adjustments made here can be accounted for at recommendation time.

<sup>5</sup>For each item, a list of the top-K most similar items is generated.

#### 4.1 MovieLens

The MovieLens ml-20m contains ratings for 138493 users on a set of 27278 movies (Harper and Konstan, 2015). This section will discuss the performance of ACUs compared to existing Core User methods using the MovieLens ml-20m dataset.

Table 1: Item Vectors Testing Error of 13000 Core Users collected with previously existing methods using the ml-20m dataset (Harper and Konstan, 2015). We averaged the results of Algorithm 2 over 75 runs where each run was given 200 independent testing users and 2600 of the Core User Item Vectors, or  $\approx 50\%$  of the singular value mass. In each run, we stopped Algorithm 2 when we reached a training error of 0.2.

Item Similarity Used	Ratings Used	Frequency-Based	Probe Entry Set Error
yes	yes	yes	1.41
yes	yes	no	<b>1.38</b>
yes	no	yes	1.71
yes	no	no	1.67
no	yes	yes	1.45
no	yes	no	1.39
no	no	yes	1.73
no	no	no	1.70

Table 1 shows the performance of the various previously existing Core User selection methods when tested using the item vectors testing error; each method was used to collect 13000 Core Users or a little under 10% of the original MovieLens ml-20m dataset size. This is half the number of Core Users necessary to maintain recommendation accuracy as claimed in (Zeng et al., 2014), but our aim is comparing the viability of these methods. To test each method we used 13 row blocks, or 1000 users per block. We found that the performance is sensitive to the number of item vectors retained as latent factors in the line 4 of Algorithm 2; we’ll discuss reasons for this a bit further down. We chose to use 20% of the singular values, for a total of 2600 latent factors across all the blocks. To make the error on the probe entry sets comparable across methods, we stopped running Algorithm 2 when the error on the training set had reached 0.2. The average absolute value of an entry in the ratings matrix, after centering, is  $\approx 0.8$ ; in other words, the error when always predicting that a missing rating will be the mean, zero, is  $\approx 0.8$ . Therefore, we ran Algorithm 2 for various Core User methods until the error on the training set was  $\approx 75\%$  better than simply always guessing the mean. The first column in Table 1 indicates whether or not hidden ratings taken from the item similarities were incorporated into the calculation of the user similarities. The second column indicates whether the cosine similarity of the user vectors is taken using the actual item ratings or just the item booleans. The third column indicates whether we used a frequency-based or a rank-based approach to select the Core Users from the aggregated lists of the top-50 most similar users to each other user. These results support previous literature suggesting that the best method for selecting Core Users considers item similarity, compares ratings rather than booleans, and uses a rank-based selection approach.

For all methods used in Table 1, Algorithm 4 returns an error of  $\approx 0.715$ ; this error is 12% better than the error when using only one centroid with the mean at the origin.

Figures 1 and 2 help to explain why the performance of Algorithm 2 is sensitive to the number of item vectors retained as latent factors in the line 4. They compare the singular values of the 13000 Core User ratings matrix where the Core Users are selected using the most competitive selection method: the method used in the second row of Table 1 to the singular values of an equivalently sparse matrix with random ratings as the non-zero values. Singular values lay out the behavior of a matrix as a mapping between spaces, and the Marchenko-Pastur theorem (Marchenko and Pastur, 1967), describes the asymptotic behavior of the singular values of large random matrices. Figures 1 and 2 show that the singular values of the Core User ratings matrix only significantly differ from those of a random matrix in the lowest order singular values, which are, by convention, the largest singular values. In fact, beyond the first 1% of singular values, the singular values of the Core User ratings matrix differ by at most 3.6 from those of a random matrix, with larger order singular values contributing less influence over the behavior of the matrix. So, while the larger order singular values of the Core User ratings matrix are non-zero, which is generally how we measure relevance, this relationship suggests that the larger order singular vectors may not be informative and may actually make learning more difficult by adding noise.

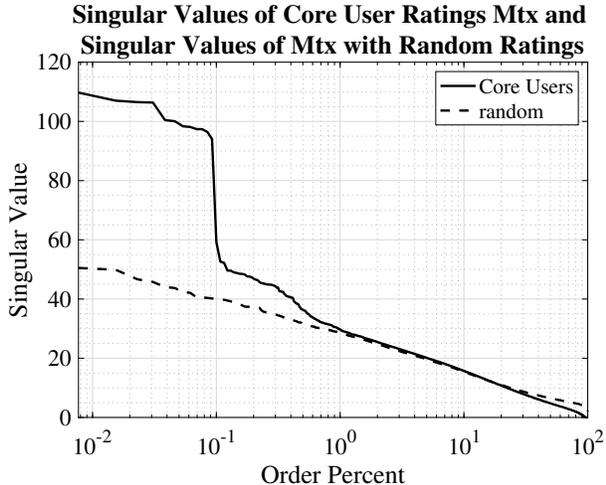


Figure 1: Singular Values of 13000 Core User Ratings Matrix where Core Users are selected from the ml-20m dataset (Harper and Konstan, 2015) using the most competitive selection method: the method used in the second row of Table 1. Singular Values of an equally sparse matrix with random ratings as the non-zero values. There are 13000 singular values, where by convention lower order singular values have larger value. The x-axis is labeled as the order percent, so the  $i^{th}$  singular value would have  $x$ -tick value  $i/130$ .

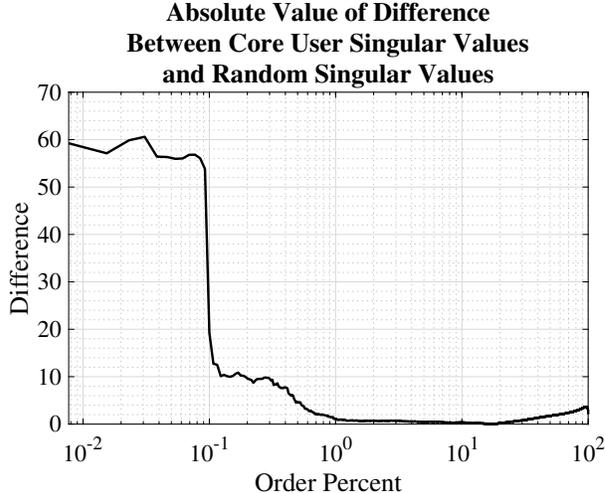


Figure 2: Absolute Value of the difference between the curves in Figure 1.

We now move to discussing the results of learning ACUs using Algorithm 1. We initialized our ACUs as Core Users selected with the most competitive selection method: the method used in the second row of Table 1.

Figure 3 shows the item vectors testing error of 13000 ACUs over iterations, where an by an iteration of Algorithm 1 we are referring to one loop beginning on line 3. For each test, we averaged the results of Algorithm 2 over 20 runs where each run was given 200 independent testing users and either 2600 or 650 ACU item vectors. As in our Core User tests, in each run we stopped Algorithm 2 when we reached a training error of 0.2. In both testing and training we used 13 row blocks, or 1000 ACUs per block. Whether we use 2600 or 650 ACU item vectors we can see clear improvement compared to the real Core User item vectors testing error, which is simply the  $y$ -intercept of these graphs. The best item vectors testing error using 650, or 5% of the 13000 ACU item vectors is better than the item vectors testing error of 27000 Core Users (20% of the users in the complete dataset) when using using 650 Core User item vectors. The best item vectors testing error using 650 of the 13000 ACU item vectors is 0.987, while the item vectors testing error of 27000 Core Users when using using 650 Core User item vectors is 1.03.

Since the Core Users are stored in the same memory format as the original complete dataset, retaining only 20% of the users as Core Users results in approximately a 80% memory reduction. The memory reduction of the Artificial Core Users depends on the number of latent factors one decides to store. If one chooses to store only 5% of the resulting latent factors, both user and item vectors, then this results in a 36% reduction in memory compared to the original data set. If one chooses to store only 5% of the the item vectors then this results in a 57% reduction in memory compared to the original data set. Additionally, the improvement in the item vectors testing error when using only 5% of the the item vectors in Algorithm 2 compared to using 20% of the the item vectors as shown in Figures 3 suggests that these extra vector may be more noisy than informative.

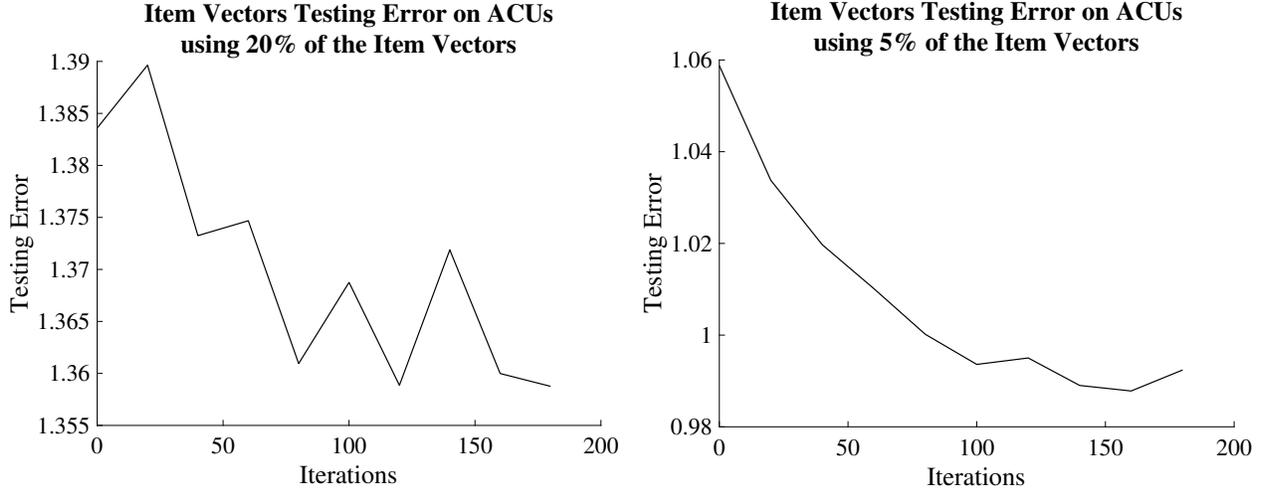


Figure 3: Item Vectors Testing Error of 13000 ACUs over iterations using the ml-20m dataset (Harper and Konstan, 2015). For each test, we averaged the results of Algorithm 2 over 20 runs where each run was given 200 independent testing users and (Left) 2600/(Right) 650 ACU item vectors. In each run, we stopped Algorithm 2 when we reached a training error of 0.2.

Admittedly, Algorithm 1 learns relatively slowly. One loop beginning on line 3 can take up to 4 minutes to complete. We stopped running Algorithm 1 after 180 iterations at which point we reached an item vectors testing error of 1.36 with 2600 ACU item vectors and 0.99 with 650 ACU item vectors, which outperforms the most competitive real Core User methods. We also improved the sparse mean error *slightly*, so our ACUs are a bit better centroids for the testing users than 13000 real Core Users are. Figure 4 shows the sparse mean error of our ACUs over iterations calculated with Algorithm 4. The second improvement is marginal, but demonstrates that our ACUs still resemble an average group real users.

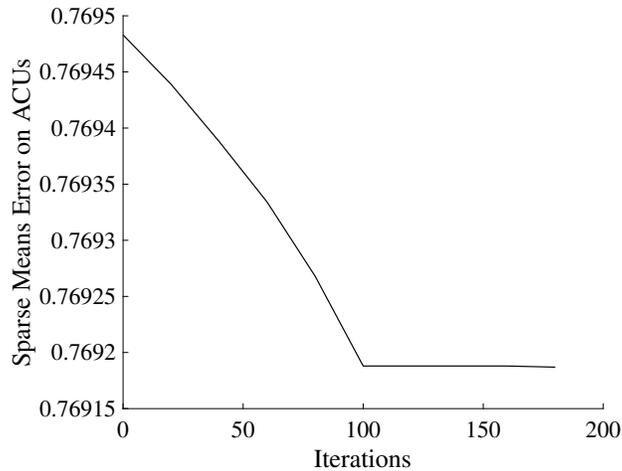


Figure 4: Sparse Mean Error of 13000 ACUs using the ml-20m dataset (Harper and Konstan, 2015).

We were able to condense the information of 27000 sparse Core Users into 13000 ACUs or approximately half the number of users. Since ACUs carry dense information, storing 5% of the 13000 ACUs latent factors takes up about three times as much memory as 27000 sparse Core Users do, but 5% of the 13000 ACUs latent factors achieves a smaller item vectors testing error than the same number of Core User latent vectors when using 27000 sparse Core Users. We can see from Figure 5 that the largest singular values of the ACU ratings matrix, that had matched the largest singular values of the ratings matrix with 13000 Core Users at initialization, has shifted toward the the largest singular values of the ratings matrix with 27000 Core Users after 180 training iterations.

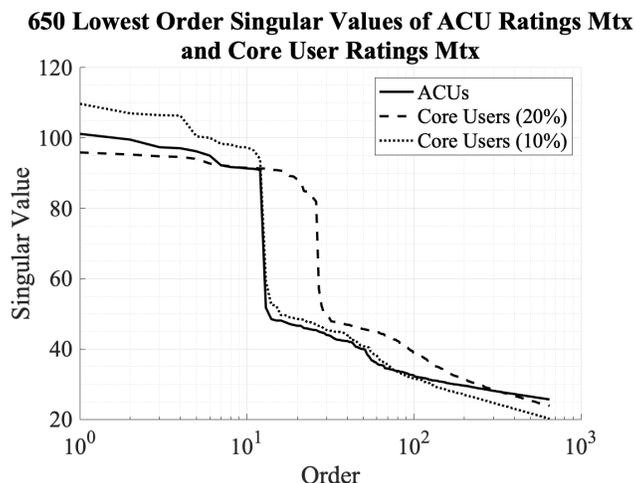


Figure 5: 650 largest singular values of 13000 trained ACU ratings matrix using the ml-20m dataset (Harper and Konstan, 2015), compared to the largest singular values of the ratings matrix of 27000 Core Users (20% of the users in the complete dataset) and the largest singular values of the ratings matrix of 13000 Core Users (10% of the users in the complete dataset).

## 5 Conclusion

We have shown that our Artificial Core Users improve the recommendation accuracy of real Core Users while mimicking real user data. Because they act as good centroids for the complete dataset, they can be considered good representatives for real user clusters. They can be stored efficiently, yet they have dense ratings information which is more immediately interpretable than sparse data. We have removed the representation limits of Core Users and shown that an iterative training process can improve the recommendation accuracy of Core Users producing data that continues to resemble that of real users, conserve memory and improve recommendation efficiency.

## References

- Charu C. Aggarwal. *Recommender Systems*. Springer International Publishing Switzerland, 2016.
- Leo Breiman. Arcing classifier (with discussion and a rejoinder by the author). *Ann. Stat.*, 26:801–849, 1998.
- Gaofeng Cao and Li Kuang. Identifying core users based on trust relationships and interest similarity in recommender system. *IEEE International Conference on Web Services*, 2016.
- Konstantina Christakopoulou and Arindam Banerjee. Adversarial recommendation: Attack of the learned fake users. *arXiv*, 2018. URL [arXiv:1809.08336](https://arxiv.org/abs/1809.08336).
- F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4, 2015.
- Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. *WWW*, 2017.
- Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. *ICDM*, 2008.
- Prateek Jain and Inderjit S. Dhillon. Provable inductive matrix completion. *arXiv*, 2013. URL [arXiv:1306.0626](https://arxiv.org/abs/1306.0626).
- Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. *KDD*, 2008.
- Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.
- Li Kuang, Gaofeng Cao, and Liang Chen. Extracting core users based on features of users and their relationships in recommender systems. *International Journal of Web Services Research*, 14, 2017.
- Shyong K Lam and John Riedl. Shilling recommender systems for fun and profit. *WWW*, pages 393–402, 2004.
- Qibing Li, Xiaolin Zheng, and Xinyue Wu. Neural collaborative autoencoder. *arXiv*, 2018. URL [arXiv:1712.09043](https://arxiv.org/abs/1712.09043).
- S. Li, J. Kawale, and Y. Fu. Deep collaborative filtering via marginalized denoising auto-encoder. *CIKM*, 2015.

- Zhang Li, Yu Lei, and Cao Shuyan. Constructing the core user set for collaborative recommendation based on samples selection idea. *International Journal of u- and e- Service, Science and Technology*, 9:27–34, 2016.
- V. A. Marchenko and L. Pastur. Distribution of eigenvalues for some sets of random matrices. *Mathematics of the USSR-Sbornik*, 1(4):457–483, 1967.
- Michael P OMahony, Neil J Hurley, and Guenole CM Silvestre. Promoting recommendations: An attack on collaborative filtering. *International Conference on Database and Expert Systems Applications*, pages 494–503, 2002.
- Al Mamunur Rashid, Istvan Albert, Dan Cosley, Shyong K. Lam, Sean M. McNee, Joseph A. Konstan, and John Riedl. Getting to know you: Learning new user preferences in recommender systems. *IUI*, 2002.
- Al Mamunur Rashid, George Karypis, and John Riedl. Learning preferences of new users in recommender systems: An information theoretic approach. *KDD*, 10:90–100, 2008.
- Badrul M. Sarwar, Joseph A. Konstan, Al Borchers, Jon Herlocker, Brad Miller, and John Riedl. Using filtering agents to improve prediction quality in the grouplens research collaborative filtering system. *Computer Supported Cooperative Work*, pages 345–354, 1998.
- Badrul Sarwar†, George Karypis, Joseph Konstan†, and John Riedl. Incremental singular value decomposition algorithms for highly scalable recommender systems. *ICIT*, 2002.
- Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–227, 1990.
- D. Sculley. Web-scale k-means clustering. *WWW*, pages 1177–1178, 2010.
- S. Sedhain, A. K. Menon, S. Sanner, and L. Xie. Autorec: Autoencoders meet collaborative filtering. *WWW*, 2015.
- Yanir Seroussi, Fabian Bohnert, and Ingrid Zukerman. Personalised rating prediction for new users using latent factor models. *ACM conference on Hypertext and hypermedia*, pages 47–56, 2011.
- Y. Shi, M. Larson, and A. Hanjalic. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *CSUR*, 2014.
- F. Strub, J. Mary, and R. Gaudel. Hybrid collaborative filtering with autoencoders. *arXiv*, 2016. URL [arXiv:1603.00806](https://arxiv.org/abs/1603.00806).
- H. Wang, N. Wang, and D.-Y. Yeung. Collaborative deep learning for recommender systems. *KDD*, 2015.
- H. Wang, S. Xingjian, and D.-Y. Yeung. Collaborative recurrent autoencoder: Recommend while learning to fill in the blanks. *NIPS*, 2016.
- Y. Wu, C. DuBois, A. X. Zheng, and M. Ester. Collaborative denoising auto-encoders for top-n recommender systems. *WSDM*, 2016.
- Wei Zeng, An Zeng, Hao Liu, Ming-Sheng Shang, and Tao Zhou. Uncovering the information core in recommender systems. *Scientific Reports*, pages 6140–6152, 2014.
- F. Zhang, N. J. Yuan, D. Lian, X. Xie, and W.-Y. Ma. Collaborative knowledge base embedding for recommender systems. *KDD*, 2016.
- S. Zhang, L. Yao, and A. Sun. Deep learning based recommender system: A survey and new perspectives. *arXiv*, 2017. URL [arXiv:1707.07435](https://arxiv.org/abs/1707.07435).
- Xiao Zhang, Simon S. Du, and Quanquan Gu. Fast and sample efficient inductive matrix completion via multi-phase procrustes flow. *ICML*, 2018.