A new movie recommendation system based on low rank matrix factorization

GEORGE GIAPITZAKIS*, University of Waterloo, Canada
OLIVIER LALONDE†, University of Waterloo, Canada

This work concerns the well-studied task of recommending movies to users based on their previous ratings of movies. We suggest a number of small but significant improvements on the standard pipeline for this task based on low-rank matrix factorization. The first improvement is to renormalize the ratings, which is found to have a number of benefits: for this, a Bayesian inference approach to parameter estimation is proposed. We also suggest an improvements on the inference step, namely, to add a diversification term to the objective value that is optimized over the full set of available movies for recommendation purposes.

Additional Key Words and Phrases: Recommender systems, matrix factorization, ALS

1 Introduction

In an era where petabytes of user data are readily available, recommendation systems have emerged as powerful tools to maximize customer satisfaction by guiding them toward content relevant to their interests. Notable services where recommendation systems are used include e-commerce platforms (like Amazon or eBay), streaming services (like Netflix or Spotify), news agencies, and many more. In this work, we propose a new algorithm for content recommendation based on low-rank matrix factorization, which builds on the standard alternating least squares algorithm, the basic ideas of which we now review. Although, throughout, we think of ourselves as recommending movies to users for concreteness, obviously, the same ideas apply to any context in which users are rating something or other. The basic scheme on top of which we will be building draws inspiration from Nikhil Saxena's blog post [5] on the subject.

Suppose we have a collection $\mathcal{U}$ of users, a collection $\mathcal{M}$ of movies, and that we have a collection of ratings, modeled as a map $R : \Omega \mapsto \{1, 1.5, 2, ..., 5\}$, where $\Omega \subseteq \mathcal{U} \times \mathcal{M}$ represents the collection of pairs (u, m) such that user u assigned a rating to movie m. The idea here is to think of R as matrix of size $|\mathcal{U}| \times |\mathcal{M}|$ of which we actually know very few entries, and for some small rank k, we want to write

$$R \approx ST$$

where S is a $|\mathcal{U}| \times k$ matrix and T is a $k \times |\mathcal{U}|$ matrix. Another way to state the exact same thing is the following: we want to associate a feature vector v_u to every user $u \in \mathcal{U}$ (which correspond to the rows of S) and a feature vector w_m for every movie $m \in \mathcal{M}$ (which correspond to the columns of T) in such a way that the rating that user u would give to movie m were they to watch the movie and actually rate it is approximately given by the dot product $v_u \cdot w_m$. The performance of the factorization corresponding to a choice of matrices S and T is given by the loss function

$$\left(\sum_{(u,m)\in\Omega} (v_u \cdot w_m - R(u,m))^2 \right) + \lambda \sum_{u\in\mathcal{U}} \|v_u\|^2 + \lambda \sum_{u\in\mathcal{M}} \|w_m\|^2$$

where $\lambda > 0$ is a regularization hyperparameter (just like k) [3]. As is usual in machine learning, the game plan is to carry out this optimization on a training set for various choices of the hyperparameters and move forward with the pair (S, T) obtained thereby yielding the smallest loss on the test set. The problem of actually carrying out this optimization is well-known to be NP-hard (i.e. finding

Authors' Contact Information: George Giapitzakis, ggiapitz@uwaterloo.com, University of Waterloo, Waterloo, Ontario, Canada; Olivier Lalonde, olalonde@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada.

S and T such that the corresponding loss is within a given constant error of the optimum), but this was never going to stop machine learning practitioners, and it turns out that reasonable results are obtained in practice using the so-called *alternating least squares* algorithm (henceforth abbreviated as ALS), which belongs to the well-known family of alternating minimization algorithms [8]. This algorithm is based on the simple observation that if one were to fix either all the v_u or all the w_m in the above expression, the problem immediately decomposes into a series of independent least squares problems with L2 regularization which are hence very easy to solve in an embarrassingly parallel way. The strategy is then to set all the v_u to be random, and then to optimize over the w_m, and then to optimize over the v_u given this choice of w_m, and then to optimize over the w_m given this choice of v_u, etc., until convergence. The number of iterations required to get good results typically ranges between 10 and 30.

Once such an approximate factorization has been obtained, it is used for inference in the following way: when a completely new user comes in and gives ratings $r_1, ..., r_l$ to movies $m_1, ..., m_l$, we wish to associate a user vector v to this user which will reflect these ratings properly. This is done by solving the following regularized least squares problem for the same value of λ that was selected during training:

$$\min_v \left(\sum_{i=1}^{l} (v \cdot w_{m_i} - r_i)^2 \right) + \lambda \|v\|^2 \tag{1}$$

This can then be used for movie recommendation purposes in the obvious way: we use this fitted user vector v to predict how user u would rate a given movie by computing its dot product with the corresponding movie vector, and recommend movies for which this predicted rating is high.

We propose improvements to both parts of this standard pipeline. First off, we propose to renormalize all ratings with respect to movie z-scores. Concretely, for every movie m, we estimate the mean, μ_m, and σ_m, the standard deviation, of the distribution of ratings that were associated to that movie, and the rating r that the user u associated to movie m is then renormalized as $\frac{r-\mu_m}{\sigma_m}$. The advantages of doing this are manifold, including a slight improvement in performance as well as the fact that the dot product of two movie vectors now carries an operational significance. Since it is found that most movies in the database are found to have few ratings attached to them, it is not possible to obtain reliable estimates for μ_m and σ_m in the naive way, and we propose a simple Bayesian inference algorithm based on rejection sampling to alleviate this problem.

Our second improvement concerns the way in which movies are recommended to users. Specifically, we consider how to implement an analog of Youtube's *new to you* feature [2] in the context of low-rank matrix factorization. A feature of the recommendation system we just described is that its recommendations won't tend to take a user out of their comfort zone, i.e. it will recommend movies to them that are similar to movies we know they like (as we can then be fairly confident that they will like them as well). In case this is not something the user wants, our simple solution to prevent this behavior is the following. After normalizing the movie vectors corresponding to movies the user watched so they have unit norm, we model this collection as a continuous distribution on the sphere, the partial density function f of which is then fitted using kernel density estimation. The method to enforce the diversification of search results is then simply to filter out the movies for which the corresponding value of f is above a certain threshold. Unlike kernel density estimation in real-dimensional space which has been well-studied and for which well-performing choices of families of kernels are known, much less work has been done when the underlying space is the sphere in several dimensions. The strategy we selected for choosing the kernel is an empirical one, where the kernel is very simply learned among an extremely general class of functions using convex optimization in the style of machine learning. We note that our approach is predicated on the rating renormalization that was previously discussed.

1.1 Datasets used

We perform all our experiments using the MovieLens datasets[1]. There are two versions of this dataset: the *small* one containing 100K ratings for ~9K movies from ~600 users and the *full* one containing 33M ratings for ~86K movies from ~330K users. For every experiment, we report which one was used. The code for the experiments is available at https://github.com/giorgosgiapis/NormalizedALS/tree/main.

2 Preliminaries

We will make use of the following standard fact:

Theorem 2.1 (The singular value decomposition). *Let A be a $m \times n$ matrix with $m < n$. There exists a $m \times m$ orthogonal matrix U, a $n \times m$ orthogonal matrix V and a diagonal matrix Σ such that*

$$A = U\Sigma V^T$$

The diagonal elements of Σ, which are unique, are called the singular values of A.

The area of the sphere in $\mathbb{R}^n$, which we will denote from now on as C_n, is known to be equal to:

$$C_n = \frac{2\pi^{n/2}}{\Gamma(n/2)}$$

where Γ denotes the gamma function.

3 Improving on ALS: renormalizing the ratings

We now propose a simple improvement on basic ALS, which consists of renormalizing the ratings associated with a given movie with respect to their z-score. Concretely, for every movie $m \in \mathcal{M}$, we imagine that the collection of user ratings of that movie is distributed according to some distribution p_r. Given the mean μ_m and the standard deviation σ_m of this distribution, we then renormalize every rating r that was attributed to this movie as

$$r \longrightarrow \frac{r - \mu_m}{\sigma_m}$$

The point of doing this is twofold. How we estimate μ_m and σ_m in practice is explained in Section 3.1.

The first point is purely based on prediction accuracy: without renormalization, given a movie vector v_m, since ratings are no smaller than 1, the constraint that $v_u \cdot v_m$ be close to the rating that user u would give to movie m means that we're restricting all user vectors to be such that $v_u \cdot v_m$ is positive (or at most slightly negative). This means that we are strongly restricting the user vectors to a small subset of the space, which harms the expressivity of the model. This is reflected in the singular vectors of the matrix whose columns are the user vectors, which are strongly skewed. With renormalization, the expectation value of $v_u \cdot v_m$ is zero, as it should be if v_u were allowed to roam freely in the space, so to say. In this way, we are giving the model much greater freedom.

The second point (which is not entirely unrelated to the first) is that, with z-score renormalization, the dot product of two movie vectors takes on an operational significance, and in fact, becomes a

[1]https://grouplens.org/datasets/movielens/

measure of how correlated their ratings are. We recall that given two random variables A and B with the same support, their correlation is defined as

$$\mathrm{corr}(A, B) = \frac{\mathrm{cov}(A, B)}{\sigma_A \sigma_B}$$

This is a number strictly between -1 and 1 which measures the degree of linear correlation between two movies. Now, given two movies m_1 and m_2, writing S to be the matrix whose columns are the user vectors, the model's estimation of their correlation is:

$$\mathrm{corr}(m_1, m_2) \approx \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} (v_{m_1} \cdot w_u)(v_{m_2} \cdot w_u) = \frac{1}{|\mathcal{U}|} v_{m_1}^T S S^T v_{m_2}$$

Writing the singular value decomposition of S as

$$S = U \Sigma V^T$$

We see that

$$\frac{1}{|\mathcal{U}|} S S^T = \frac{1}{|\mathcal{U}|} U \Sigma^2 U^T$$

As mentioned previously, due to renormalization, the movie vectors fill out the whole space in a reasonably isotropic way, which implies that the singular values of S are fairly uniform. In the case of the small database, it is found that the ratio between the largest and the smallest singular value is smaller than 2 with renormalization and is about 5 without it. We therefore have that Σ^2 is roughly equivalent to the identity matrix times some constant. This means that the correlation between m_1 and m_2 is roughly proportional to the dot product of their movie vectors. Since this correlation takes values in $[-1, 1]$, provided that the norms of w_{m_1} and w_{m_2} are fairly similar, this means that:

$$\mathrm{corr}(m_1, m_2) \approx \frac{w_{m_1} \cdot w_{m_2}}{\|w_{m_1}\| \|w_{m_2}\|}$$

This is indeed something that is seen experimentally on the average, although there are some outliers.

3.1 Parameter estimation using Bayesian inference

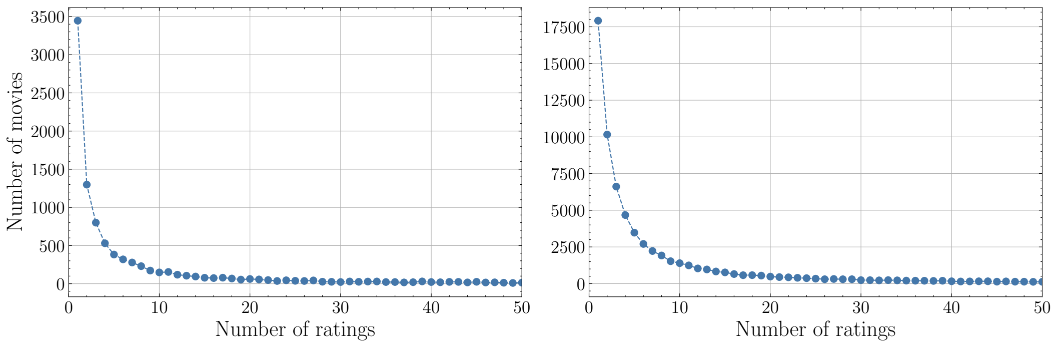

Fig. 1. Histogram of the number of movies with every number of ratings for the small (left) and full (right) dataset

As mentioned earlier, for a given movie m, we wish to estimate the mean μ_m and the standard deviation σ_m of the true rating distribution p_r^m of this movie's ratings. To this end, the obvious thing to do would be to simply compute the empirical mean and standard deviation of the ratings that were given to it and take these as estimates. As can be seen from Figure 1, however, most movies in the database have far too few ratings attached to them for this naïve approach to work. We describe an approach based by Bayesian inference which provides more accurate estimates.

The basic idea of our approach is the following. In the spirit of Bayesian statistics, we treat the true rating distribution p_r^m as itself being a random variable drawn from some probability distribution $f(p_r)$. Letting D stand for the data that was observed about m (i.e. the number of ratings in every category), our estimates for μ_m and σ_m will then be the expected values of the corresponding parameters when p_r is drawn according to the distribution $f(p_r|D)$. In our case, $f(p_r)$ is approximated as the distribution resulting from sampling uniformly from the set Q of *trusted empirical distributions*: this is the set formed by selecting the movies which received a number of ratings greater than some threshold (which we more or less arbitrarily picked to be 120) and computing the empirical distribution of the ratings of each of them. The problem of estimating μ_m and σ_m therefore reduces to the problem of sampling distributions from the distribution $f(p_r|D)$.

This is done in the following way. Bayes' theorem gives:

$$f(p_r|D) = \frac{f(D|p_r)f(p_r)}{f(D)}$$

Letting M be any upper bound on $\max_{p_r} f(D|p_r)$, the following rejection sampling algorithm will therefore generate values of p_r that are distributed according to the distribution $f(p_r|D)$:

```
• Set $P$ to be the empty list
• Repeat:
    – Sample $p_r$ uniformly at random from $Q$, and pick $u$ uniformly over the interval $[0, 1]$
    – If it holds that $\frac{f(D|p_r)}{M} > u$, add $p_r$ to $P$
```

All that remains to do is determine numerically the value of $\frac{f(D|p_r)}{M}$ above. Fixing a movie, for every possible rating r, let N_r be the number of times that rating r was awarded to the movie, let N be the sum of all the N_r, and let $\hat{p}_r = \frac{N_r}{N}$ be the corresponding empirical probability distribution. We see that:

$$f(N_r|p_r) = \prod_r (p_r)^{N_r}$$

Gibb's inequality from information theory says that this expression is maximized by the empirical distribution. It follows that we can take:

$$M = \prod_r (\hat{p}_r)^{N_r}$$

Taking logarithms for reasons of numerical stability yields:

$$\log\left(\frac{f(N_r|p_r)}{M}\right) = \sum_r N_r \log\left(\frac{p_r}{\hat{p}_r}\right) = -N D_{\mathrm{KL}}(\hat{p}_r, p_r)$$

where D_{KL} is the Kullback-Leibler divergence of distributions. The full algorithm is the following:

The rejection sampling algorithm for parameter estimation

```
(1) Set $M, \Sigma$ to be empty lists
```

(2) Repeat as long as the empirical errors on the means of M and Σ are above some fixed threshold:
   (a) Sample q_r uniformly at random from Q, and generate a uniformly random variable u in $[0, 1]$
   (b) If it holds that:
$$\exp(-D_{\mathrm{KL}}(\hat{p}_r, q_r)N) > u$$
   Add q_r's mean to M and its standard deviation to Σ

(3) Output the means of M and Σ as the estimates for the mean and standard deviation of the true distribution p_r

If N is small, we see that most samples will pass the test 2.b, so that the distribution $f(p_r|D)$ is quite close to the distribution $f(p_r)$. The algorithm is essentially dismissing the observed data as unreliable. When N gets large, however, the algorithm becomes increasingly selective and only distributions such that $D_{\mathrm{KL}}(\hat{p}_r, q_r)$ is small (and hence such that q_r and $\hat{p}_r$ are close, by Pinsker's inequality) are likely to be accepted, so that the outputted estimates converge to the estimates one would obtain directly from $\hat{p}_r$ directly. One can think of the algorithm as a principled way of interpolating between these two regimes. This means that a threshold should be set on the value of N such that if N is big enough, we should simply directly estimate μ and σ from $\hat{p}_r$, as running the rejection sampling algorithm is both pointless, as it would output values close to these anyway, and also an enormous waste of time, as the number of iterations required grows quickly with N.

To test how well the algorithm performs in practice, we began by generating the set Q of trusted distributions on the large dataset in the manner described previously. We then simulated the case of a movie with few ratings simply by picking a distribution q_r from Q uniformly at random and by generating N samples from it, for $N \in \{5, 10, 15, 20, 25\}$. Figure 2 illustrates the mean squared error of both the naive estimate using the sample mean and the estimate resulting from rejection sampling (using the actual mean of q_r as a reference point). We can see that the rejection sampling procedure consistently performs a fair bit better than the naive estimate when N is small but that this advantage vanishes when N gets large, which is consistent with the description of the algorithm's behavior that was given previously. Figure 3 shows the average number of iterations needed for the rejection sampling algorithm to converge on both datasets. Convergence is achieved when the sample variance of both the mean and the standard deviation of the accepted samples so far is at most α times the number of already accepted samples, where α is a cutoff threshold (we chose $\alpha = 0.1$), or when the maximum number of iterations is reached (we chose 100K). As a fail-safe, we always force the algorithm to generate at least 100 accepted samples. As mentioned earlier, with increasing number of samples N, the algorithm becomes more selective, and hence more iterations are needed to achieve convergence.

We compare our method against a baseline ALS implementation on the MovieLens small dataset[2]. For the baseline ALS implementation, we set the regularization parameter to 0.1 while for our method we set it to 0.2. We fix the maximum number of iterations to 25 for both methods. For the rejection sampling step in normalized ALS, we take all movies that have more than 110 ratings to build the set of trusted distributions. For movies with more than 45 ratings, we employ sample mean estimation instead of rejection sampling[3]. In Figure 4 we present the test losses across various

[2]This choice of the small version was made primarily because of time restriction and the current state of the Datasci cluster.
[3]The parameters for the Baseline ALS were selected using a full grid search. For normalized ALS, due to time restrictions, we picked the parameters through trial and error. With a full grid search we could have possibly achieved even better loss values.

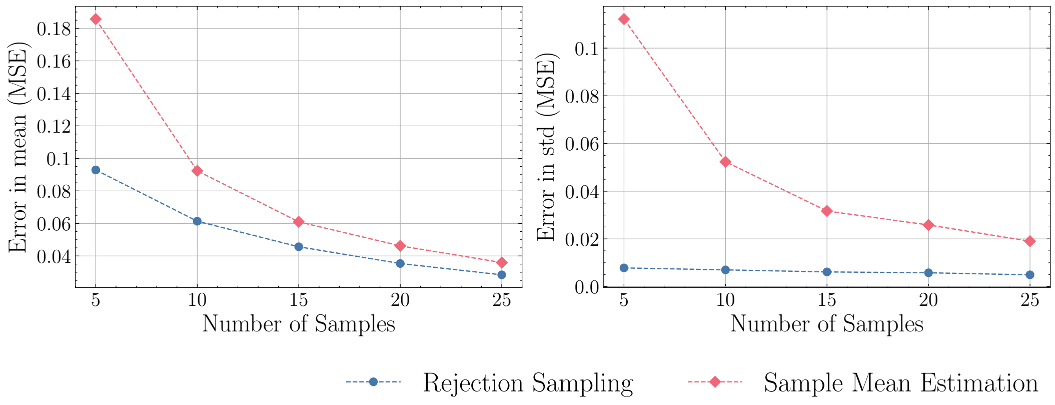

Fig. 2. Illustration of the performance of rejection sampling for mean (left) and standard deviation (right) estimation on the full dataset

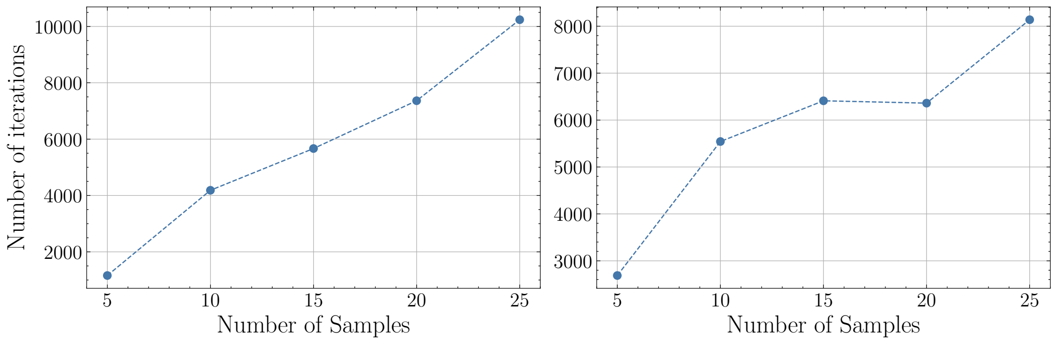

Fig. 3. Number of rejection sampling iterations as a function of the number of samples N for the small (left) and full (right) datasets.

ranks of the factor matrices. Overall, normalized ALS consistently outperforms the baseline for every rank tested (~5% improvement in MSE loss).

4 Improving inference with recommendation diversification

As discussed in the introduction, we consider a way to implement recommendation diversification in the context of low-rank matrix factorization-based movie-recommendation systems. Concretely, what this means is that for a given user who has awarded an sufficiently large number of ratings, we strive to bias the system's recommendations away from what is typical for that user. As per the results of the previous section, thanks to rating normalization, it is the case that, given two movies m_1 and m_2, the normalized cosine distance

$$\frac{w_{m_1} \cdot w_{m_2}}{||w_{m_1}||||w_{m_2}||}$$

is a good indicator of how similar two movies are, i.e. what the correlation between their ratings is. In view of this, our strategy is the following: given the set S of normalized movie vectors corresponding to all movies which the user rated, we will view S as samples from a probability distribution f on the sphere in real-dimensional space which encodes the user's viewing habits.

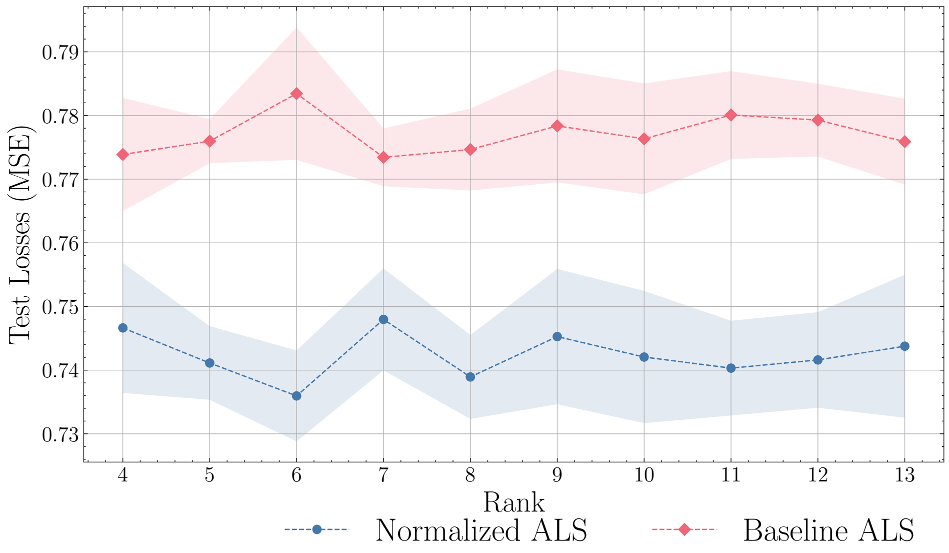

Fig. 4. Comparison of test MSE losses for baseline ALS (red line) and normalized ALS (blue line) for various ranks of the factor matrices. We report the mean loss across 10 runs. The shaded area represents the standard deviation.

How this is done is the subject of section 4.1. Given the user vector v_u that we inferred for the user under consideration using regularized least squares, we will then maximize the objective:

$$v_u \cdot w_m - B \log(\epsilon + f(w_m/||w_m||))$$

over all movies m, for some parameter $B > 0$ and for some small $\epsilon > 0$, and recommend to the user the first movies for which this value is largest. It is simple to see that the smaller $f(w_m/||w_m||)$ is, the larger the additional term is. Whether this is the right way to enforce recommendation diversification can be debated, but this is a reasonable choice considering that it is proportional to the *information content* [7], which is the go-to way to quantify how 'surprising' an event is in information theory. The ϵ term is there so that the term is stabilized for movies with a very small (or zero) corresponding value of p_m. We note that we did not try to find out what reasonable choices for B and ϵ could be because we do not have a baseline to compare to: in a real world setting, an actual pool of users would be needed for this.
This section is organized as follows: the first subsection describes how we use kernel density estimation to estimate the probability density function f, and the second subsection explains how it can be implemented in a distributed computing setting using locality sensitive hashing.

4.1 Kernel density estimation on the sphere in n dimensions

Given a user u, let T be the collection of normalized feature vectors corresponding to the movies that the user rated. As described in the introduction, for diversification, we want to think of T as being sampled from a probability distribution f on the sphere in $\mathbb{R}^k$, which encodes the likelihood of a user watching a hypothetical movie corresponding to the given point. It can be argued that this doesn't make complete sense because the set of points on the sphere which correspond to movies is finite, but since k is small, the number of movies is enormous and the corresponding feature vectors are fairly isotropic, for modeling purposes, it isn't unreasonable to think of every point on the sphere as corresponding to a movie. Given the vectors in T, therefore, we would like to come up with a reasonable estimate for f. We show how we try to do this using kernel density estimation, henceforth abbreviated as KDE, which works in the following way.

Given a space $\mathcal{S}$, a collection of samples $\{x_1, ..., x_m\}$ in it coming from an unknown distribution and given a kernel function $K(x, y) : S \times S \mapsto \mathbb{R}^+$ with the property that for any $x \in \mathcal{S}$, we have that

$$\iint\limits_S K(x, y)dy = 1$$

KDE estimates the density of the original distribution as

$$f_{\text{KDE}}(x) = \frac{1}{m}\sum_{i=1}^{m} K(x_i, x) \tag{2}$$

It is easy to see that f_{KDE} is a valid probability density function over S from the conditions we imposed on K. When S is taken to be the whole of $\mathbb{R}^n$, it is traditional to take K to be of the form $K(x, y) = g(||x - y||)$, and a great deal is known about the theoretically right choice of g (which is most commonly taken to be a Gaussian function with mean 0 and whose variance is determined from sophisticated theoretical considerations). When S is the sphere in $\mathbb{R}^n$, however, a great deal less appears to be known. Inspired by the above, we will look at kernel functions of the form $K(x, y) = g(x\cdot y)$, where $g : [-1, 1] \mapsto \mathbb{R}^+$ is an arbitrary continuous function, which common sense dictates should be monotonically increasing. Instead of the aforementioned traditional approach which consists of selecting g from a restricted set of functions based on theoretical considerations, we will fit g ourselves among a very general set of functions using convex optimization. This optimization will be performed in the following way. We will begin by selecting a number of prolific users in terms of the number of ratings: let $S_1, ..., S_m$ be the corresponding collections of movie vectors. For every user $i \in \{1, ..., m\}$, we partition S_i into sets S_i^1 and S_i^2, with the first being about four times as large as the second (to get the classic 80%-20% split in machine learning). The vectors in S_i^1 will be used for approximating the probability density function f_i of user i's movie distribution, and the vectors in S_i^2 will be used for testing how good the approximation is. Specifically, given a vector y, from Equation (2), the model's predicted value for $f_i(y)$ is given by:

$$\frac{1}{|S_i^1|}\sum_{x\in S_i^1} K(g(x \cdot y))$$

We use the negative log-likelihood loss from machine learning to quantify the quality of this approximation using the vectors in S_i^2. The total loss of the model over the training set is then given by:

$$\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{|S_i^2|}\sum_{y\in S_i^2} - \log\left(\frac{1}{|S_i^1|}\sum_{x\in S_i^1} K(g(x \cdot y))\right)\right)$$

Given that our parametrization of $g(t)$, which we describe in the next section, will be a linear combination of real-valued parameters for any fixed t, this objective will be convex, so the whole problem will be efficiently solvable provided that the constraints in the model are also convex, as they will be. We now proceed to explain these things in more detail.

4.1.1 The space from which g will be selected. Given a cutoff $t_0 \in [-1, 1)$ (which is a hyperparameter), we will be considering kernel functions g of the form

$$g(t) = I[t \geq t_0] \sum_{k=0}^{N} a_k T_k(t)$$

where the T_n are the standard Chebyshev polynomials of the first kind, which are the go-to method for approximation schemes in real analysis, where $I[t \geq t_0]$ denotes the indicator function, i.e. it equals one if $t \geq t_0$ and 0 otherwise, and where the a_k are the free parameters of the model, i.e. those are the parameters that we are optimizing over. It can be shown that any function on $[-1, 1]$ which is exactly zero for all $t \leq t_0$ can be approximated arbitrarily well using functions of this form, and that N can be taken to be fairly small unless the function in question is very wiggly. The point of introducing a hard cutoff t_0 (which will generally take values around 0.5) is that it is experimentally observed that the kernel $K(x, y) = g(x \cdot y)$ associated with a given x should be strongly localized around x, i.e. be zero for points that are far away from it.

We now turn to discussing which constraints are to be imposed on the parameters a_k so that the resulting function $g(t)$ has the properties that we want, i.e. that the corresponding kernel forms a probability density function on the sphere. The simplest one is the requirement that g be nonnegative: in fact, we will require that $g(x)$ be monotonically non-decreasing everywhere. From requiring that g also be continuous, we get the following linear constraints on the a_k:

$$g(t_0) = \sum_{k=0}^{N} a_k T_k(t_0) = 0$$

and:

$$g'(t) = \sum_{k=0}^{N} a_k T_k'(t) \geq 0, \quad \forall t \in [t_0, 1]$$

This last set of constraints is infinite in size: we simply handle it by generating one such constraint for every point in a fine-grained partition of $[t_0, 1]$. This wouldn't work if N were big, but since we will be working with small values of N, the results that we observe are reasonable. The other requirement is that the corresponding kernel function integrates to one over the sphere, i.e.

$$\iint_S g(x \cdot y) dy = 1$$

Because of rotational symmetry, the above holds for all x if and only if it holds for an arbitrary x. It can be checked (by assuming without loss of generality that $x = (1, 0, 0, ..., 0)$ so that $g(x \cdot y) = g(y_1)$, and splitting the integral) that this amounts to the following normalization condition:

$$\int_{-1}^{1} g(t)(1 - t^2)^{\frac{n-3}{2}} dt = \frac{1}{C_{n-1}}$$

where Γ is the gamma function. The corresponding constraint on the a_k is:

$$\int_{-1}^{1} g(t)(1 - t^2)^{\frac{n-3}{2}} dt = \sum_{k=0}^{N} \left(\int_{t_0}^{1} T_k(t)(1 - t^2)^{\frac{n-3}{2}} dt \right) a_k = \frac{1}{C_{n-1}}$$

It follows that the normalization constraint is simply a linear constraint on the a_k. All that remains to do is to evaluate the integrals corresponding to the coefficients of the a_k numerically: this is quite straightforward to do and many approaches are possible. In our code, we employed a variation on Clenshaw-Curtis quadrature [1] based on the fact that, for any $k \in \mathbb{N}$ and any $s \geq 0$, the integral

$$\int_{-1}^{1} t^k (1 - t^2)^s dt$$

admits an analytic expression in terms of the beta function, namely, it is 0 if k is odd and $B\left(\frac{k+1}{2}, s+1\right)$ if k is even, as can be seen from the substitution $u = x^2$.

4.1.2 Numerical results. We now report the results of the method we have been describing so far in this section. We implemented our algorithm in the Julia language, in part because it is far easier to do mathematics in it than in Scala. The corresponding code is in the file `fitkernel.jl`. The corresponding convex optimization problem was solved using the SCS solver [4]. For our ends, we took N to be equal to 9, as we found that increasing its value did not change the result of the algorithm very much. We considered the 100 most prolific users in the small dataset, which all have hundreds of ratings. These were randomly split into 80 training users, which were used for training the model, and 20 users, which were used for testing the model, i.e. we evaluated the average loss on every one of them. We set $N = 5$ across the board and considered cutoff values in $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$. Each run took about 10 minutes. The resulting kernels are displayed in figure 5 and the loss on the training and test sets are displayed in figure 6. In all cases, as a sanity check, we numerically integrated the resulting kernel over the sphere to ensure that it is indeed normalized, i.e. integrates to one.

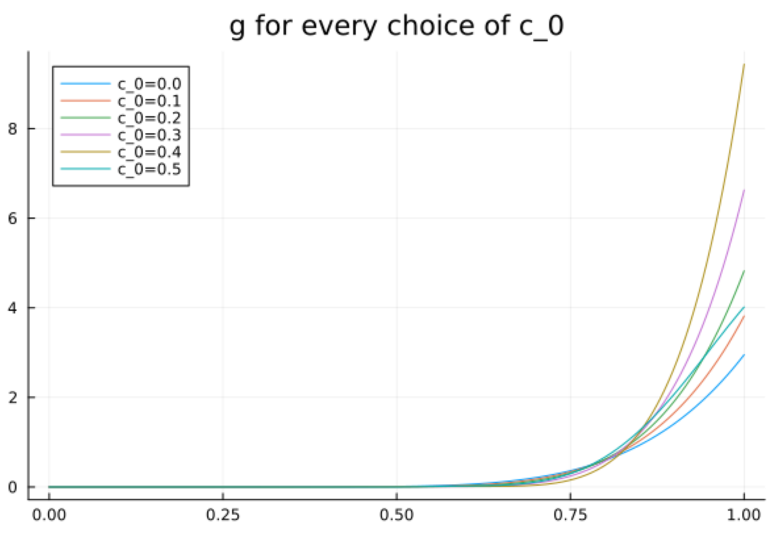

Fig. 5. Kernel selected by the model for every choice of cutoff

It may be surprising at first that the error decreases as the cutoff increases: if N were taken to be very large, this wouldn't happen, as increasing the cutoff restricts the space of functions that the model is able to choose from. But in this case, since N is fairly small, increasing the cutoff enables the model to better approximate the best possible kernel, which is very close to zero for values much smaller than one. We note that the slight discrepancy in the second plot is due to numerical instability reasons (the solver failed to converge): as the cutoff gets larger, the coefficients

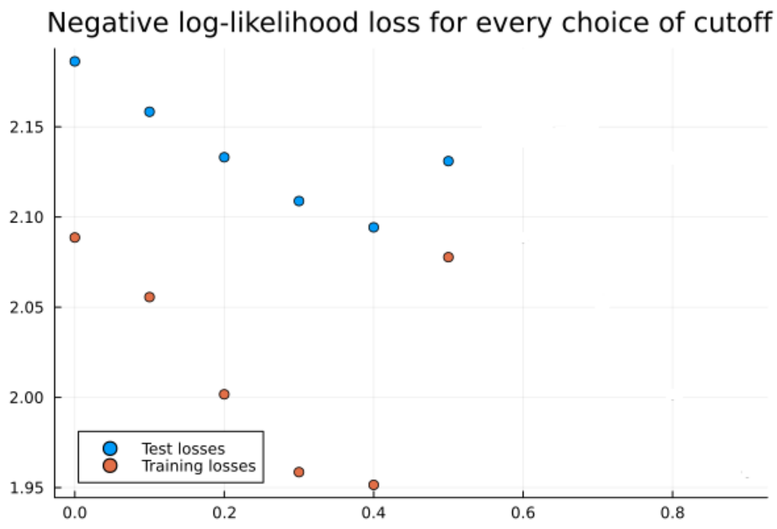

Fig. 6. Training and test losses for every cutoff

of the optimal polynomial blow up (as can be seen from the polynomial below), which makes the optimization problem harder. Had we have more time, we would've tried to circumvent this issue, as solving the optimization problem for a larger cutoff would both give better performance and also make the distributed algorithm run faster. From figure 5, it is reasonable to think that the cutoff could be taken to be as large as 0.6 without degrading the loss significantly.

We should briefly mention that the losses which we obtained, which are around 2, are significantly smaller than that which would be obtained using the baseline of the uniform distribution on the sphere, which would yield a loss of $\log(C_6) \approx 3.43$. This means that the kernel density estimation is doing a decent enough job, especially considering that the negative log-likelihood is lower bounded in expectation value by the entropy of the true distribution which we are trying to model, which is probably reasonably large, so the loss we obtained is likely not very far off from being smallest possible. For completeness, we write down the polynomial we obtained for $c_0 = 0.4$, which is:

$$g(t) \approx -17.647 + 171.865t - 664.272t^2 + 1273.414t^3 - 1210.543t^4 + 456.619t^5$$

4.2 Implementation of recommendation diversification in a distributed setting

We end this subsection to mentioning how KDE for recommendation diversification can be implemented in a distributed setting. Specifically, we are given a list of users $u_1, \dots, u_n$, and given the collection of movies $\mathcal{M}_i$ which was watched by user i with corresponding movie vectors S_i, we want to approximate the probability density function f_i which we infer from S_i using the techniques of this subsection for every movie in $\mathcal{M}_i^c$. From the following simple relationship between distances and dot products, which holds for any two vectors u and v:

$$\begin{aligned}\|u - v\|^2 &= (u - v) \cdot (u - v) \\ &= \|u\|^2 + \|v\|^2 - 2(u \cdot v)\end{aligned}$$

it holds that, for any i and for any vector v, since we are working exclusively with unit vectors, the KDE estimate for f_i satisfies:

$$f_i(x) \approx \frac{1}{|S_i|} \sum_{\substack{y \in S_i \\ \|x-y\| \le d_0}} g(x \cdot y)$$

where

$$d_0 = \sqrt{2 - 2c_0}$$

This naturally suggests that locality sensitive hashing could be used to significantly improve on the naive approach of computing the dot product of x with every vector in S_i from a performance standpoint, without harming the precision with which f_i is calculated too much. Specifically, we begin by computing

$$\bar{\mathcal{M}} = \bigcup_{i=1}^{n} \mathcal{M}_i$$

This is simple to do using Spark: we take the ratings dataframe, filter out ratings corresponding to users not in the list $u_1, ..., u_n$, drop all columns but the one corresponding to movie ids, and run `.distinct` on the result. We then perform an inner join with the result and the dataframe containing the movie vectors; we then perform an approximate join on the resulting dataframe with the dataframe containing the movie vectors using Spark's `BucketedRandomProjectionLSH` feature [6], with d_0 being specified as the distance threshold. For every row of the result, we compute the dot product of the two vectors and drop the two columns corresponding to the vectors. We now have a dataframe with three columns: two movieId columns and one dot product column, with the only moviesIDs in the first column corresponding to movies in $\bar{\mathcal{M}}$. From here, routine Spark operations can be used to approximate $f_i(m)$ for every user index i and every movie m.

The full distributed algorithm for recommending movies can be found in the file `RunInference.scala`.

5 Conclusion and future work

In this work, we proposed two modifications to the baseline ALS algorithm for collaborative filtering and empirically showed that our method performs better on the MovieLens small dataset. We also described a distributed pipeline for recommendation diversification. In terms of experiments, we leave as future work a more thorough hyperparameter tuning for our method as well as experimentation with various other available datasets, beyond movie recommendation.

References

[1] C. W. Clenshaw and A. R. Curtis. 1960. A method for numerical integration on an automatic computer. *Numerische Mathematik 2, 197* (1960).

[2] James Hale. 2021. YouTube Rolls Out Personalized Video Recommendation Feed 'New To You'. https://www.tubefilter.com/2021/07/08/youtube-new-to-you-video-recommendation-feed/ Accessed: 2024-12-08.

[3] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *2008 Eighth IEEE International Conference on Data Mining*. 263–272. https://doi.org/10.1109/ICDM.2008.22

[4] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. 2016. Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding. *Journal of Optimization Theory and Applications* 169, 3 (June 2016), 1042–1068. http://stanford.edu/~boyd/papers/scs.html

[5] Nikhil Saxena. [n. d.]. Collaborative Filtering Recommendation System with Apache Spark using Scala. https://medium.com/@nihitextra/collaborative-filtering-recommendation-system-with-apache-spark-using-scala-9a68e02e814d Accessed: 2024-12-09.

[6] Spark 3.5.3 ScalaDoc. [n. d.]. BucketedRandomProjectionLSH. https://spark.apache.org/docs/3.5.3/api/scala/org/apache/spark/ml/feature/BucketedRandomProjectionLSH.html Accessed: 2024-12-09.

[7] Wikipedia. [n. d.]. Information content. https://en.wikipedia.org/wiki/Information_content Accessed: 2024-12-09.

[8] Yaoliang Yu. 2022. Alternating Minimization. https://cs.uwaterloo.ca/~y328yu/mycourses/794/794-note-alt.pdf Accessed: 2024-12-08.