From Tapestry to SVD

A Survey of the Algorithms That Power Recommender Systems

Abstract

This paper is a survey of the algorithms that power recommender systems. To start, the social and monetary relevance of recommender systems is outlined. Then we delve into the specifics of how the first recommender system, Tapestry, coined the idea of numerically defining customer similarity. Moving forward, we show how this central concept of similarity is re-hashed in present day recommender systems, namely that of Amazon.com. Specifically, we examine the complexity of a user-based approach in a large scale system such as Amazon's, identify its weaknesses, and see how these weaknesses are overcome using an item-based approach. The last component of this paper focuses on the Netflix Prize™ and investigates the single most important algorithm in the contest so far: an incremental approach to finding the singular value decomposition (SVD) of a mostly-blank matrix.

By Joseph Huttner
Under the direction of Professor Steven Lindell
Haverford College Department of Computer Science
8 May 2009
Dedication

I would like to dedicate this thesis to my parents, Paula and Eric, my brother, Jared, and my sister, Elizabeth. Your continued support in my collegiate endeavors, academic and otherwise, will never be forgotten. I love you all!

-Joseph
# Table of Contents

Introduction ............................................................................................................. 4  
Tapestry .................................................................................................................. 7  
User-based Approach .............................................................................................. 8  
A Deeper Look ......................................................................................................... 13  
Item-Based Approach .............................................................................................. 14  
The Netflix Prize ...................................................................................................... 15  
Singular Value Decomposition (SVD) .................................................................. 17  
Combining Netflix and SVD .................................................................................. 23  
Conclusion ............................................................................................................... 28  
Works Cited ............................................................................................................ 31
1. INTRODUCTION

From the beginning of human existence until the latter part of the 1700s, nearly all goods used and consumed by man were fruits of his own labor. Egyptian relics were hand-built by thousands of the Pharaohs’ men. Bronze canons on British naval vessels were hand-forged. And clothes were hand-weaved using threads that were hand-spun. Production's limiting factor was surely human labor.

The events leading up to the Industrial Revolution of the 1780s were instrumental in changing production forever. In 1776, James Watt invented a much improved steam engine and incorporated a company to produce and sell it. During the inaugural presentation of the steam engine, Watt’s business partner, Matthew Bolton, proclaimed, “I sell here, Sir, what all the world desires to have—POWER”1. Mechanical power, that is. Indeed, steam proved a far more efficient energy than the human variety, and its supply seemed endless.

The consequences of this bottomless labor supply were profound. As Wired magazine columnist George Gilder explains,

Suddenly you could do things you could not afford to do before. You could make a factory work twenty-four hours a day churning out products in a way that was just incomprehensible before the industrial era. It really did mean that physical force became virtually free in a sense. The whole economy had to reorganize itself to exploit this physical force. You had to ‘waste’ the power of the steam engine and its derivatives in order to prevail2.

And waste they did! Companies such as Ford Motor Corp. and John Deere became famous for developing fully-automated assembly lines capable of running perpetually, if so desired. Across many industries production levels grew quickly and machines’ role in production was indubitable.

Fast forward to the digital revolution of the 1980s and we see a similar trend with logic. Specifically, a markedly more efficient, cheap type of switch takes the digital world by storm. Vacuum tubes, a predecessor of the transistor, were expensive, handmade, and unreliable. They gave way to silicon transistors, which were reliable, machine-made, and inexpensive. Just as in the Industrial Revolution, the successful companies of the digital revolution owed a large part of their success to their knack for finding new ways to utilize this extra computing power. This idea still holds: Google, though known to the world as a search engine, "in many ways is foremost an effort to build a network of supercomputers, using the latest academic research, that can process more data - faster and cheaper - than its rivals"3. In the northwestern United States alone, Google, Microsoft, Amazon, and Yahoo are all competing to build some of the world’s largest and most advanced computer facilities, “packed with tens of thousands of servers that will propel the next generation of Internet applications. Call it the data center arms race”4. The more cheap computational power that Google can tap into, and the more novel things they can do with this power, the better positioned it is to outpace its rivals.

Today, this idea of “waste,” or more simply a more efficient utilization of resources, also applies to physical goods, especially in e-commerce. Unlike traditional brick and mortar stores, where variety is limited by the number of products a store can display in an aisle, online

---

1 Levere 1.
2 Kelly 2.
3 Hansell and Markoff 2.
4 Katz 1.
merchants like Amazon.com can exploit their almost insignificant storage and distribution costs. As diagram 1 shows, online merchants have a huge incentive to increase their catalog size by exploiting their limitless shelf space because the gross difference between marginal revenue and marginal cost is high. Contrastingly, for brick and mortar stores the difference is low, or even negative.

**Diagram 1.**

Common sense dictates that any profit-maximizing company will continue to expand so long as the marginal cost of expansion is less than the marginal revenue resulting from that expansion. Hence, while traditional stores must restrict themselves to selling only the most popular goods, the low expansion costs of online merchants allow them to realize significant profit from selling small quantities of hard-to-find items to a dramatically expanded customer base. Diagram 2 illustrates the huge volume of less popular items that many online merchants focus on. Collectively, these products are known as the “long tail.”
Given that the long tail extends as far out as the number of products in existence, it is tempting to think the amount of goods that long tail-oriented businesses sell is proportional to the variety of goods offered. But this hypothesis supposes that customer consumption is proportional to the amount of goods offered, which is clearly incorrect – Amazon’s virtual store may sell 1000 times more products than a local electronics store, but Amazon’s average customer does not buy 1000 times more electronics. Interestingly, the opposite train of thought is more correct. As the abundance of the long tail grows, people consume almost no more than before. There is “still a scarcity of human attention and hours in the day. In some sense, it’s still a fixed pie game”\(^5\).

While consumption levels stay flat, what does change as more goods become available is the time consumers must spend analyzing prospective purchases. Surely, figuring out which of the 1700 mp3 players available on Amazon one should buy is much more time-consuming than choosing from among the twenty available at Best Buy\(^6\). Therefore, it comes as no surprise that for long tail oriented businesses to remain competitive they must provide consumers with some guidance as to which products may interest them. Without this guidance, consumers are swarmed by an endless trough of products that is too cumbersome to fully analyze. As Clive

---

\(^5\) Anderson 1.

\(^6\)
Thompson of the New York Times writes, “How do you decide among an effectively infinite number of options?”

The systems that help consumers decide which products they should engage in (whether that engagement means buying, watching, listening, reading, etc.) are called recommender systems. This paper provides a historical overview of several algorithms that power recommender systems, examines their computational complexity, and explains why particular varieties of recommender systems are better suited to specific tasks.

2. TAPESTRY

Though current recommendation systems are synonymous with e-commerce sites like Amazon and Netflix, their history predates either company. The first recommender system, Tapestry, was developed at the Xerox Palo Alto Research Center and is described in a 1992 issue of the Communications of the ACM. The motivation for Tapestry was the increasing use of electronic mail, which resulted in “users being inundated by a huge stream of incoming documents.” One initial attempt to reduce the quantity of incoming mail was “to provide mailing lists, enabling users to subscribe only to those lists of interest to them.” However, in practice users’ interests rarely corresponded perfectly to preexisting mailing lists.

A second, better approach was for each user to define a personal filter, thereby receiving only those e-mails that were caught by this filter. More specifically, the system would allow for collaborative filtering, which in this case meant that “people collaborated to help people perform filtering by recording their reactions to the documents they read.” Tapestry’s formal name for these reactions was “annotations.” In today’s language, “annotations” are akin to tagging - “That article made me laugh, so I’ll tag it as ‘funny.’” A typical filter for some user, say Catherine, in Tapestry might have been “documents tagged as ‘funny’ by Amy, Annie, or Allison.” Then, whenever Amy, Annie, or Allison tags something as “funny,” whatever document receiving that tag would be delivered to Catherine’s inbox. Generally, the idea here is that the active user is in some way similar to a many other users. Upon examining these similar users’ most-liked items, it is possible to get an idea of what items the active user may like.

As seen with Catherine’s filter, determining to whom to send documents cannot be computed by simply examining a document when it enters the e-mail system. Rather, the nature of filters requires “(potentially) repeatedly issuing queries over the entire database of previously received documents.” This is because a user may react to a document sometime after it arrives, and then tag it. Upon tagging a document, Tapestry then cued up all the filters

7 Thompson 2.
8 Barry Schwartz, in his now-famous book The Paradox of Choice, goes one step further. He argues that too many options actually decrease human happiness. Clearly, a lot of psychological and social factors are at play here, and they fall outside the scope of this thesis. However, if we buy into Mr. Schwartz’s idea, which he supports empirically, we can make the case that recommender systems actually help to increase human happiness by filtering out superfluous options.
9 Goldberg, Nichols, Oki, and Terry.
10 Ibid. 61.
11 Ibid. 61.
12 Ibid. 61.
13 Ibid. 61.
14 Ibid. 61.
that triggered on that particular tag from that particular user, and then sent out the document to the users whose filters were triggered.

Despite recognizing the importance of tagging when creating a successful recommender system, Tapestry had its shortcomings. Namely, it was impossible to analyze whether users were receiving the content they actually wanted to read. To re-use our example: Are Amy, Annie and Allison’s annotations really the best indicators of what Catherine will find funny? To answer this question, Tapestry would need to have accumulated user experience data to better analyze how well filters really worked. The importance of measuring the accuracy of recommender systems becomes a key theme in future implementations.

3. USER-BASED APPROACH

When Amazon introduced a system for generating item recommendations for users, the company faced even more challenges than Tapestry. As outlined in an internal Amazon paper on the subject, e-commerce recommender systems operate in a challenging environment for several reasons:\footnote{Linden, Smith, and York 76.}

\begin{enumerate}
\item A large retailer might have huge amounts of data, tens of millions of customers and millions of distinct catalog items.
\item Many applications require the set to be returned in real-time, in no more than half a second, while still producing high-quality recommendations.
\item New customers typically have extremely limited information, based on only a few purchases or product ratings.
\item Older customers can have a glut of information based on thousands of purchases and ratings.
\item Customer data is volatile: Each interaction provides valuable customer data, and the algorithm must respond immediately to new information.
\end{enumerate}

Following Tapestry’s lead, Amazon's first thought in constructing a recommender system was that based on the active user's buying history, the recommender system would find other users with similar buying patterns, and then recommend those other users’ highly-rated items. The notion of similarity is key here. Recall that in Tapestry, users were matched up by the users themselves; users selected who they were similar to by tagging other users. The Amazon system, however, would have to perform this matching automatically. Formally, this is called user-to-user collaborative filtering - “user-to-user” because users are matched to one another and “collaborative filtering” because the algorithm relies on historical purchasing data to determine which users are similar.

In general, user-based systems compute the top-$A$ recommended items for a particular user by following a three-step approach. “In the first step, they identify the $K$ users in the database that are the most similar to the active user. During the second step, they compute the union of the items purchased by these users and associate a weight with each item based on its importance in the set. In the third and final step, from this union they select and recommend $A$ items that have the highest weight and have not already been purchased by the active user.”\footnote{Deshpande and Karypis 146-147.}
To get a better flavor for how such a system works, let’s examine how exactly each of these three steps is executed.

Step 1 - Identify the K users in the database that are most similar to the active user, \( u \).

Since there are many users in the system at any one time, it makes sense to construct a similar users matrix for all user pairs, and later generate the recommendations for the active user “on the fly” - i.e. online. Step 1’s task is solely to generate this similar users matrix. A common method to calculate similarity is using the cosine similarity score. Using this technique, similarity scores between users will range from negative one, meaning the two users are exactly opposite, to positive one, meaning exactly the same, with zero indicating independence, and in-between values indicating intermediate similarity or dissimilarity.

To perform the similarity calculation with respect to each user, each customer is first represented as an \( N \)-dimensional vector of items, where \( N \) is the number of distinct catalog items. The values of each customer's vector are the customer’s rating for every item purchased. If a customer has not purchased an item, the corresponding index in the vector is left blank. The similarity between two users is found by taking the cosine of each user's \( N \)-dimensional item vector:

\[
\text{similarity}(\vec{U}, \vec{V}) = \cos(\vec{U}, \vec{V}) = \frac{\vec{U} \cdot \vec{V}}{\|\vec{U}\| \cdot \|\vec{V}\|}
\]

The “\( \cdot \)” denotes the vector dot-product operation – the \( j \)th value of \( u \)’s item vector, \( U \), is multiplied by the \( j \)th value of \( v \)’s, \( V \), for all mutually non-blank values. Then these values are summed. The \( \|\vec{U}\| \) operation indicates \( \|\vec{U}\| \)’s magnitude – the square root of the sum of the square of all the values in \( \vec{U} \).

For almost all customers the item vector is extremely sparse (i.e. there are a lot of blanks) since nearly all customers buy very few of the total available items. Consequently, the time required to compute similarity with one of these item vectors is \( O(1) \). To construct all of these similarities is then \( M \times O(1) = O(M) \) since there are \( M \) total users. However, a few customers, like editors, have actually rated many products in the catalog. For them, the magnitude of their customer vector is close to the total number of distinct catalog items \( N \), as shown in diagram 3. The time required to compute a single one of these similarity vectors is bounded by \( O(N) \). Since so few customers actually have an item vector this big, the time to compute these similarities is still \( O(N) \). Summing these two complexities gives the total complexity for the first step of the algorithm, making the similarity vector: \( O(M + N) \).
As we proceed to step 2, we must note that it is inefficient to remember all the “similar” users when those towards the end of the list are actually very dissimilar to the corresponding active user, \( u \). For this reason, only the \( K \) most similar users are selected and used as we proceed onward.

Step 2 - Compute the union of the items rated by the most similar users, \( K \), and associate a weight with each item based on its importance in the set. More formally, suppose that \( u, v \) are users and \( i, j \) are items. In order to estimate the unknown rating \( r_{ui} \) (user \( u \)'s rating of item \( i \)), we resort to a set of users \( K(u; i) \) that tend to rate similarly to user \( u \) (found in step 1) and actually rated item \( i \). I.e. \( r_{vi} \) is known for every \( v \in K(u; i) \). Then, the estimated value of \( r_{ui} \) is taken as a weighted average of these similar users' ratings, as shown in equation 1:

**Equation 1.**

\[
R_{ui} = \frac{\sum_{v \in K(u; i)} S_{uv} \cdot r_{vi}}{\sum_{v \in K(u; i)} S_{uv}}
\]

The similarity of two users \( u \) and \( v \) is denoted \( S_{uv} \). Here, predicted ratings (weights) are determined by multiplying similarity and rating, though there are other equations that accomplish this weighting differently. Figure 1 on page twelve illustrates this computation.

The complexity for generating the predicted ratings for a single customer \( u \) is \( O(K) \) because \( S_{uv} \) and \( r_{vi} \) are already stored, and so their lookups are just \( O(1) \). Since \( K \) contains only the most similar customers to \( u \), and each \( v \) in \( K \) has purchased relatively few items, the
complexity is dominated by the size of $K$. So the total complexity here is $O(K)$ for each user.

So the total complexity of the first two steps is simply their sum: $O(M + N) + O(K)$ where $K << M < N$, or just $O(M + N)$.

Step 3 - Select and recommend the $A$ items that have the highest weight and have not already been purchased by the active user. Here, to recommend items for user $u$, we simply look at the predicted ratings from step 2 and select the $A$ items with the highest rating. The complexity is then $O(A)$ per user. Note that since this step uses the current user’s predicted ratings as input, and returns the just those products with the highest predicted ratings as output, we need not analyze this step’s complexity in relation to the two previous steps. Rather, we just compute the complexity separately: $O(A)$ per user.
Step 2 of the user-to-user collaborative filtering algorithm:
Creating music album recommendations for Deborah.\(^{17}\)

<table>
<thead>
<tr>
<th>User</th>
<th>Similarity (S)</th>
<th>Kanye</th>
<th>Jay-Z</th>
<th>Nos</th>
<th>S * Kanye</th>
<th>S * Jay-Z</th>
<th>S * Nos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maria</td>
<td>0.99</td>
<td>3.0</td>
<td>2.5</td>
<td>-</td>
<td>2.97</td>
<td>2.48</td>
<td>-</td>
</tr>
<tr>
<td>Eduardo</td>
<td>0.38</td>
<td>3.0</td>
<td>3.0</td>
<td>1.5</td>
<td>1.14</td>
<td>1.14</td>
<td>0.57</td>
</tr>
<tr>
<td>Javier</td>
<td>0.89</td>
<td>4.5</td>
<td>-</td>
<td>3.0</td>
<td>4.02</td>
<td>-</td>
<td>2.68</td>
</tr>
<tr>
<td>Florencia</td>
<td>0.92</td>
<td>3.0</td>
<td>3.0</td>
<td>2.0</td>
<td>2.77</td>
<td>2.77</td>
<td>1.85</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>10.9</td>
<td>6.39</td>
<td>5.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.sum</td>
<td></td>
<td>3.18</td>
<td>2.29</td>
<td>2.19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total / S.sum = Predicted rating for Deborah 3.43 2.70 2.33

Look at the data for the “Jay-Z” column. Maria rated the Jay-Z album 2.5 and has a similarity score of 0.99 with Deborah. Multiplying Maria’s rating by this similarity score equals 2.48. This operation is repeated for all three users who have rated the Jay-Z album (Javier did not rate it). The sum of these three values (6.39) represents the top half of equation 1 on page ten. Then, sum just the similarity scores of all the users who rated the Jay-Z album. In this case, that is Maria, Eduardo, and Florencia. This sum (2.29) represents the bottom half of equation 1. Finally, divide “Total” (6.39) by “S.sum” (2.29) giving the Deborah’s predicted rating for the Jay-Z album (2.70).

\(^{17}\) Based on the table in Programming Collective Intelligence by Toby Segaran, page 15.
4. A DEEPER LOOK

At first, our thought may be that the complexity for steps one and two (constructing the similar users table and predicting the ratings) is relatively fast because $O(M+N)$ is linear. And while true that running a linear time algorithm on a huge dataset is definitely feasible, there are some underlying issues that make this computation painstakingly more cumbersome than it initially appears. First is that this user-based algorithm is linear for each user. Consequently, we need to add another $M$ to the front of $O(M+N)$, making the complexity quadratic instead of linear.

Another barrier to implementing the user-based approach is more subtle. Recall that customers are frequently buying and rating new products. Since most customers have purchased relatively few items, and thus have sparse item vectors, user similarity scores are very sensitive to new purchase data. Intuitively, this makes sense. If you only ordered five things ever on Amazon, and then purchased a sixth item that you rated very highly, your new most-similar users vector would definitely reflect your liking of this sixth item. It is unlikely that the twenty users you were most similar to before rating the sixth item are anywhere near the top after rating it. Moreover, any similarity score that does not reflect this new data is worthless because it is not indicative of your present taste. In some cases, providing a bad or incorrect recommendation is worse than providing no recommendation at all. (Imagine you waste $15 on a crappy album. Now you’re angry with Amazon. You would have been happier just browsing artists on your own.) So to truly have constant utility, user similarity vectors must be re-calculated after every single item purchase.

This begs the question of how many new similarity scores need to be generated in a given amount of time so that the recommender system can give accurate recommendations. In the worst case, every customer's similar users vector changes with any purchase by any user because that latest purchase may be the one that makes two previously dissimilar users similar, and two previously similar users dissimilar. So a new similarity score needs to be calculated between the active user and every other user upon every purchase. That’s 50 million new similarity scores every time an item is rated. Per day, about 300,000 items are purchased and rated on Amazon. That means over 10 billion new similarity scores must be generated per second!

Such an extreme quantity of online computations encounters severe performance and scaling issues, even on Amazon’s robust servers. Choking on these time-sensitive computations means outdated, lower-quality recommendations, which in turn reduces sales and ultimately profit. Although one way to reduce the complexity of the most-similar users computation is to cluster the users and work off the nearest cluster only, this tends to decrease the quality of the recommendation. So despite the somewhat attractive linear complexity of a user-to-user collaborative filtering algorithm, the frequency with which user similarity scores must be recalculated causes a huge complexity hit and forces all computation to be done online. As a result, the user-based approach is impractical for a large-scale system.

18 “Amazon’s Multiple Personalities.” Using this data, which cites internal Amazon factsheets, Amazon does roughly 400,000 orders a day. That number is probably a little high, because Amazon’s other service offerings are included in their gross sales numbers. It's probably closer to 250-300k orders a day.

19 Linden, Smith, and York 77.

20 Ibid. 77.
5. THE ITEM-BASED APPROACH

As previously stated, the underlying problem with the user-based approach is that user similarity scores are inherently very sensitive to new purchase data. In thinking about similar items, however, we see an opposite scenario. If we think about two similar items, we can imagine how when a new customer buys and rates either of these two items, it is not that likely that the similarity between these items changes much. There are two reasons for this. First, every item is purchased many times, so one additional rating only affects the final similarity score a little bit. The second reason is more subtle: similar products are in fact similar because they share relatively few key qualities. Two first-person shooter games might both have great graphics, an awesome dungeon level, and good multiplayer capabilities. Beyond that, there is not much room for differentiation - they are just video games. A product is just a product. And for most products, consumers only care about four or five key qualities. Consequently, once two items are deemed “similar,” it is unlikely that they will ever become dissimilar. The steadiness of item similarity scores means they can be calculated offline with relative infrequency because the accuracy of a particular recommendation changes only very slightly with each new customer rating. Therefore, it comes as little surprise that Amazon uses an item-based collaborative filtering algorithm for its recommender system. Following is a breakdown of a standard item-to-item collaborative filtering algorithm.

The primary insight behind an item-based algorithm is the fact that a customer is more likely to purchase items that are similar to the items that he has already purchased in the past. By analyzing historical purchasing information, these sets of similar items can be identified and used for generating recommendations. In general, item-based systems compute the top-$A$ recommended items for a particular user in two steps. “The first component builds a model that captures the relations between the different items,” and the second component “applies this precomputed model to derive the top-$A$ recommendations for an active user”\textsuperscript{21}. The first component is online and the second component is offline. Here is how a detailed explanation of the how they are performed:

Step 1 - Build a model that captures the relations between the different items. More simply, for each item in the catalog we want to compute a list of items most similar to that item. While our first instinct might be to create a huge product-to-product matrix and fill it in with a similarity score for every product pair, “many product pairs have no common customers, and thus the approach is inefficient in terms of processing time and memory usage”\textsuperscript{22}. A loop-based approach is much more efficient:

\textsuperscript{21} Deshpande and Karypis 147.
\textsuperscript{22} Linden, Smith and York 77.
1. For each item $I$ in product catalog: \[ O(N) \]

2. For each customer $C$ who purchased $I$: \[ O(M) ** \]

3. For each item $J$ purchased by customer $C$: \[ O(N) \]

4. Record that a customer purchased $I$ and $J$: \[ O(1) \]

5. For each such item $J$: \[ O(N) ** \]

6. Compute the similarity between $I$ and $J$: \[ O(M) \]

where “similarity” is identical to our earlier equation, but with items in place of users:

\[
similarity(I, J) = \cos\left(\frac{I \cdot J}{\|I\| \cdot \|J\|}\right)
\]

Similarity for two items is only computed if some customer has purchased both items. In the worst case, every customer has purchased $I_1$, and every customer has also purchased every other item in the catalog. Therefore, we would iterate through the entire product catalog at line 1 (this is constant), iterate through the entire user vector at line 2, and then re-iterate through the entire product catalog at line 3. The upper bound on the complexity of this loop is $O(N^2M)$ as we need to compute $N(N-1)$ similarities, each potentially requiring $M$ operations. In practice, however the complexity is closer to $O(NM)$ because most users have purchased few items. This affects the number of calls to lines 3 and 5 (indicated with “**”), which now become very small.

As we proceed to step 2, we must note that it is inefficient to remember all the “similar” items when those towards the end of the list are actually not all that similar to the corresponding active item. For this reason, only the $K$ most similar items are selected and used as we proceed forward.

Step 2 - Apply this pre-computed model to derive the top-$A$ recommendations for an active user. The time required to compute the top-$A$ recommendations for an active user that has is given by $O(AK)$ because we need to look at the $K$ most similar items for each one of the items that the user has already purchased and identify the overall $A$ most similar items.

6. THE NETFLIX PRIZE

In October, 2006, Netflix, the online DVD rental service company, announced a competition to improve the accuracy of its proprietary movie recommendation system, Cinematch, by ten percent. Dubbed “The Netflix Prize,” the competition has garnered much media attention in large part because of the $1$ million grand prize it carries. Anyone can enter the contest so long as they agree to license their potentially winning solution to Netflix. To date,
4439 teams from 182 different countries have submitted 38422 entries, with the best solution hovering at a 9.65% improvement over Cinematch23.

To take a crack at beating Cinematch, one starts by downloading a dataset of 100 million known movie ratings from real Netflix users. Each datum is a triplet of the form (user ID, movie ID, rating). The range of values for the rating field is 1-5, with higher numbers indicating greater preference. So one such triplet might be (99064, 13236, 3) indicating that the user with ID 99064 rated the movie with ID 13236 a 3. The challenge is to go make sense of these 100 million entries. Particularly, for some (user ID, Movie ID, unknown) not in the database, calculate the value of the unknown rating.

Visualizing the dataset as a huge, sparsely-filled matrix helps clarify the problem at hand. In it are users across the top and movies down the side, with each cell containing either an observed rating (1-5) or a blank, indicating that that user has not rated that movie. Since there are 17,000 movies and 500,000 users, the matrix will have about 8.5 billion entries. Also, since the dataset contains only 100M ratings, 8.4 billion entries are blank - i.e., only one in eighty-five cells contains a rating.

<table>
<thead>
<tr>
<th>Users (500,000)</th>
<th>User ID 01</th>
<th>User ID 02</th>
<th>User ID 03</th>
<th>User ID 04</th>
<th>User ID 05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movie ID 01</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Movie ID 02</td>
<td>?</td>
<td>3</td>
<td>?</td>
<td>?</td>
<td>2</td>
</tr>
<tr>
<td>Movie ID 03</td>
<td>5</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Movie ID 05</td>
<td>?</td>
<td>5</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

To determine how well a proposed algorithm predicts ratings, all those question marks (“blanks”) must be replaced by a predicted rating. This “filled” matrix is sent off to Netflix whereupon it is checked against a less sparse version of the dataset released for public testing. If the training set is 84/85ths blank, the control set is perhaps 81/85ths blank. The pairs that are blank in the public dataset but non-blank in the control dataset are called the control pairs. How accurate the submitted algorithm predicts the ratings of the control pairs determines the its accuracy. Accuracy is measured using mean squared error (MSE)24, meaning if the algorithm guesses a 1.5 and the actual rating is 2, you are docked (2 - 1.5)² points, or 0.25. These errors are computed for all control pairs, and then their average is computed. The ratio of this average (the MSE) to Cinematch’s MSE gives the algorithm's improvement score.

---


24 Technically, accuracy is measured using root MSE, which means you’d be docked 0.25¹/² in this example. But since MSE and RMSE are monotonically related, I have opted to use MSE because it is simpler conceptually.
Although many early attempts to beat Cinematch used item or user-based algorithms, any improvements they made were quite small. The first major breakthrough in the competition used a wholly new approach based on singular value decomposition (SVD) to vault from nowhere to number four on the leader board in “one fell swoop”\textsuperscript{25}. To understand how SVD can generate recommendations, we must first understand SVD in its mathematical context. Then we will see how it applies to the specific problem of generating recommendations.

7. SINGULAR VALUE DECOMPOSITION (SVD)

At its heart, SVD is a “way of factoring matrices into a series of linear approximations that expose the underlying structure of the matrix”\textsuperscript{26}. Two important properties of the SVD are that this linear factoring is exact and optimal. “Exact” in the sense that adding together the series of linear factors yields the original matrix, and “optimal” because at each step in the series, the standard means for measuring matrix similarity, mean squared error, is minimized. To showcase how SVD works on a complete set of data, we will borrow an example from Charles Paulson which looks at the golf scores for three “hypothetical” players, Phil, Tiger, and Vijay.

Suppose this trio plays nine holes and they each unbelievably make par on every hole. Then their scorecard, which can also be visualized as a 9x3 hole-player matrix, might look like the following bolded box:

<table>
<thead>
<tr>
<th>Hole</th>
<th>Phil</th>
<th>Tiger</th>
<th>Vijay</th>
<th>Par</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

If we wanted to predict what each player might shoot on same nine holes should they play them again, we might realize that one way to do so is to assign each hole a HoleDifficulty factor, and each player a PlayerAbility factor, and find the predicted score by somehow relating these two terms. Just by examining the matrix, we see that if we make HoleDifficulty equal to par for every hole, and PlayerAbility equal to one for all players, then we can produce the entire scorecard by simply multiplying HoleDifficulty by PlayerAbility:

\textsuperscript{25} Thompson 3.
\textsuperscript{26} Paulson 1.
Note that this factorization exactly predicts all the scores in the scorecard. In fact, this one-dimensional (1-D) factorization is the SVD of the scorecard matrix.

While it is easy to see that for this hypothetical scorecard the 1-D factorization exactly reproduces the original matrix, many matrices cannot be decomposed by mere eyeballing. One option, then, is to make use of the many powerful, pre-existing algorithms that can generate the SVD of any matrix, no matter how complex it is, and simply have it compute the SVD. The problem with these algorithms is that they apply traditional linear algebra operations to complete matrices. Consequently, they do not help with conceptually understanding what is going on when computing the SVD incrementally. Trying to wrap our heads around such algorithms makes no sense for our application. Our time is better served by going through an example in which we can observe the general process of computing the SVD using an incremental method. Here our dataset here is the real scorecard from the 2007 PGA Championship. The scores from that event are:

<table>
<thead>
<tr>
<th>Hole</th>
<th>Phil</th>
<th>Tiger</th>
<th>Vijay</th>
<th>Par</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

---

27 Strang 352.
Let's think about the 1-D factorization of this matrix for a moment. Just by observing the original scorecard, we should note that whatever the 1-D factorization is, it cannot be exact. This is because on hole 2, Phil beats Vijay by a stroke, but on hole 3 Vijay beats Phil by a stroke. Since multiplying a HoleDifficulty value by a single PlayerAbility value generates the values in the 1-D product matrix, it is impossible that

\[
\text{PlayerAbility}_{\text{Phil}} \cdot \text{HoleDifficulty}_{\text{Hole2}} < \text{PlayerAbility}_{\text{Vijay}} \cdot \text{HoleDifficulty}_{\text{Hole2}}
\]

and

\[
\text{PlayerAbility}_{\text{Phil}} \cdot \text{HoleDifficulty}_{\text{Hole3}} > \text{PlayerAbility}_{\text{Vijay}} \cdot \text{HoleDifficulty}_{\text{Hole3}}
\]

And also, a bit of housekeeping: by convention, we will call this one-dimensional attempt at finding the SVD the “rank-1 SVD,” instead of the “1-D factorization.” This simply indicates that this factorization is one-dimensional (rank-1), and optimal (by mean squared error), but the product of the factors does not necessarily exactly reproduce the original matrix. Using mathematical techniques not relevant to this paper, we get the rank-1 SVD of our matrix:

\[
\text{Rank-1 SVD} = \begin{array}{l}
\text{HoleDifficulty} = \begin{array}{cccc}
3.95 & 4.27 & 2.42 & 3.64 & 3.69 & 3.33 & 3.08 & 4.55 \\
4.27 & 5.02 & 2.85 & 4.67 & 4.33 & 3.92 & 3.63 & 5.35 \\
2.42 & 2.85 & 2.66 & 4.00 & 4.05 & 3.66 & 3.39 & 5.00 \\
3.64 & 4.67 & 4.36 & 4.00 & 4.05 & 3.66 & 3.39 & 5.00 \\
3.69 & 4.28 & 4.00 & 4.00 & 4.05 & 3.66 & 3.39 & 5.00 \\
3.33 & 4.33 & 4.05 & 4.05 & 4.05 & 3.66 & 3.39 & 5.00 \\
4.55 & 5.35 & 5.00 & 5.00 & 5.00 & 5.00 & 5.00 & 5.00
\end{array}
\end{array}
\]

As previously stated, the rank-1 SVD does not exactly reproduce the original matrix – Tiger actually shot a 4 on the first hole, not a 3.95. So how can we deal with this limit of the rank-1 SVD? The answer is that the process of SVD continues refining these HoleDifficulty and PlayerAbility components in a cycle. Remember that in the rank-1 approximation we started with some HoleDifficulty values and then figured out PlayerAbility values according to
HoleDifficulty. Once these PlayerAbility values are determined, we get a rank-1 approximation as previously shown. With this approximation in place, we can now compare the HoleDifficulty x PlayerAbility matrix to the actual matrix, and record the error values in a new matrix:

<table>
<thead>
<tr>
<th>Actual Scores</th>
<th>Rank-1 SVD</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phil Tiger Vijay</td>
<td>Phil Tiger Vijay</td>
<td>Phil Tiger Vijay</td>
</tr>
<tr>
<td>4 4 5</td>
<td>3.95 4.64 4.34</td>
<td>0.05 -0.64 0.66</td>
</tr>
<tr>
<td>4 5 5</td>
<td>4.27 5.02 4.69</td>
<td>-0.28 -0.02 0.31</td>
</tr>
<tr>
<td>3 3 2</td>
<td>2.42 2.85 2.66</td>
<td>0.58 0.15 -0.66</td>
</tr>
<tr>
<td>4 5 4</td>
<td>3.97 4.67 4.36</td>
<td>0.03 0.33 -0.36</td>
</tr>
<tr>
<td>4 4 4</td>
<td>3.64 4.28 4.00</td>
<td>0.36 -0.28 0.00</td>
</tr>
<tr>
<td>3 5 4</td>
<td>3.69 4.33 4.05</td>
<td>-0.69 0.67 -0.05</td>
</tr>
<tr>
<td>4 4 3</td>
<td>3.33 3.92 3.66</td>
<td>0.67 0.08 -0.66</td>
</tr>
<tr>
<td>2 4 4</td>
<td>3.08 3.63 3.39</td>
<td>-1.08 0.37 0.61</td>
</tr>
<tr>
<td>5 5 5</td>
<td>4.55 5.35 5.00</td>
<td>0.45 -0.35 0.00</td>
</tr>
</tbody>
</table>

To correct for these errors in the rank-1 SVD, we construct a rank-2 SVD. The new dimensions in this two-dimensional factorization serve only to address the errors in the one-dimensional factorization, and thereby create a factorization that is closer to regenerating the original matrix. Intuitively, it makes sense that by adding values to the matrix factorization (i.e. adding another dimension) the better we can explain, or recreate, the original matrix. So, adding this dimension should help get us closer to the original matrix. To compute the rank-2 SVD, we start by creating a new variable, HoleDifficulty2, whose goal is to tweak the errors of the original HoleDifficulty variable, now affectionately referred to as HoleDifficulty1. Then, with HoleDifficulty2 in place, we create PlayerAbility2 to tweak the errors of PlayerAbility1. We end up with the following:
To join these newly formed matrices with those from the rank-1 SVD, we stack the width-one HoleDifficulty (HD) matrices side-by-side, and the height-one PlayerAbility (PA) matrices on top of each other, and multiply them to get the rank-2 SVD:

\[
\begin{bmatrix}
0.05 & -0.64 & 0.66 \\
-0.28 & -0.02 & 0.31 \\
0.58 & 0.15 & -0.66 \\
0.03 & 0.33 & -0.36 \\
0.36 & -0.28 & 0.00 \\
-0.69 & 0.67 & -0.05 \\
0.67 & 0.08 & -0.66 \\
-1.08 & 0.37 & 0.61 \\
0.45 & -0.35 & 0.00
\end{bmatrix}
\begin{bmatrix}
0.82 & -0.20 & -0.53
\end{bmatrix}
\]
We continue this process of adding another dimension to reduce the mean squared error \( n \) dimensions in total, until the residual error of the rank-\( n \) SVD equals zero. This means the rank-\( n \) SVD exactly recreates the original matrix. For the golf example, the rank-3 SVD is the lowest dimension factorization that is exact. Its form, therefore, is the following:

\[
\begin{bmatrix}
\text{Original \_ Matrix : (9x3)}
\end{bmatrix}
= \begin{bmatrix}
\text{HoleDiff}_1 \ (9x1) & \text{HoleDiff}_2 \ (9x1) & \text{HoleDiff}_3 \ (9x1)
\end{bmatrix}
\begin{bmatrix}
\text{PlayerAbility}_1 \ (1x3) \\
\text{PlayerAbility}_2 \ (1x3) \\
\text{PlayerAbility}_3 \ (1x3)
\end{bmatrix}
\]

And the values follow:

<table>
<thead>
<tr>
<th>Actual Scores = rank-3 SVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phil</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HoleDifficulty_1-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.34</td>
</tr>
<tr>
<td>4.69</td>
</tr>
<tr>
<td>2.66</td>
</tr>
<tr>
<td>4.36</td>
</tr>
<tr>
<td>4.00</td>
</tr>
<tr>
<td>4.05</td>
</tr>
<tr>
<td>3.66</td>
</tr>
<tr>
<td>3.39</td>
</tr>
<tr>
<td>5.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PlayerAbility_1-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phil</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0.91</td>
</tr>
<tr>
<td>0.82</td>
</tr>
<tr>
<td>-0.21</td>
</tr>
</tbody>
</table>
8. COMBINING NETFLIX AND SVD

Now that we have an understanding of both the Netflix Prize and SVD, let’s figure out how one can help solve the other. As Simon Funk, the pioneer of using SVD on the Netflix dataset, states, “Imagine for a moment that we have the whole shebang [of the Netflix users x movies matrix]--8.5 billion ratings and a lot of weary users. Presumably there are some generalities to be found in there, something more concise and descriptive than 8.5 billion completely independent and unrelated ratings”\(^{28}\). For example, any movie can likely be described in terms of some number of attributes, such as “comedy”, “action”, leading actors, and so on. Likewise, every user’s preferences can be thought of in terms of certain similar qualities – how much they like comedy, action, particular actors, as well as whether they tend to rate high or low. If we believe that the dataset can be explained by such attributes, then we should be able to explain the 8.5 billion ratings in much fewer than 8.5 billion numbers because there are bound to be some generalities that speak across a large portion of the dataset. For instance, “a single number specifying how much action a particular movie has may help explain why a few million action-buffs like that movie”\(^{29}\).

The problem is that we have no idea what the most telling preferences are for the users and movies in the dataset, so there is no way to summarize the dataset based on these preferences. However, the reverse way of thinking does have true utility: if meaningful generalities can help to summarize the dataset, then summarizing the dataset may help us extract meaningful generalities. So the challenge now becomes how to represent the Netflix dataset using a smaller number of parameters, and then inferring from this compressed data what these parameters actually are. “In today's foray, that model is called singular value decomposition”\(^{30}\). In other words, in the process of computing the SVD, the most meaningful preferences that affect users’ ratings will automatically show themselves as nameless parameters signifying an unknown feature.

Suppose for a moment that we limit the dataset to forty aspects, such that each rating is described by forty values illustrating how much each movie embodies this aspect, and also that each user is described by forty values indicating how much he cares for each aspect. To generate a rating, “we just multiply each user preference by the corresponding movie aspect, and then add those forty leanings up into a final opinion of how much that user likes that movie”\(^{31}\):

\[
\text{for } f \text{ from 1 to 40:} \\
\quad \text{rating [user][movie] += Feature [f][user] * Feature [f][movie]}
\]

For example, *Fast & Furious* might be (action = 3, chickflick = 1, humor = -2) and user Tim might be (action = 2, chickflick = -1, humor = -1). Combine these terms and you get \(3*2 + 1*(-1) + 2*(-1) = 6 - 1 - 2 = 3\). Note here that the values of these aspects can be negative -- *Fast & Furious* is tagged as not funny, and Tim likewise is averse to funny movies, so *Fast & Furious* actually scores positive points with Tim for being decidedly not funny.

Anyhow, the decomposition of the original matrix is two very oblong matrices – the 17,000 x 40 movie aspect matrix, and the 500,000 x 40 user preference matrix. All said and

---

\(^{28}\) Funk, “Netflix Update: Try This at Home,” 2.


done, this model requires \((17,000 + 500,000)\times 40\), or about 20 million values – four hundred times less than the 8.5 billion for the original matrix. When we multiply these two matrices together, all we are really doing is taking the dot-product for every combination of user and movie and filling out the giant \(users \times movies\) matrix with the result\(^{32}\).

Unfortunately, there are some serious hurdles to calculating the SVD of the matrix representation of the Netflix dataset, \(R\). First, \(R\) is very big, and taking the SVD of a really big matrix requires tremendous computational power. Second, \(R\) is sparse in the sense that a lot of entries are blank (not zero, but actually without a value), and traditional SVD does not work on this type of matrix. The algorithm that Simon Funk uses to compute the SVD of the Netflix dataset is based on pre-existing algorithms for finding the incremental SVD of data with missing values, sometimes referred to as a modified Generalized Hebbian Algorithm\(^{33,34}\). Rather than using the classical methods of linear algebra to compute SVD, this approach is an iterative method based on calculus.

The Incremental SVD Algorithm, as I refer to it, is a gradient descent algorithm to compute the best approximation of the complete \(users \times movies\) matrix using only the known entries of \(R\). In mathematics, gradient descent is a general framework for solving optimization problems where we want to minimize functions of continuous (differentiable) parameters\(^{35}\). In traditional calculus, if we want to find the minimum of some function, \(y = f(x)\), we take the derivative of \(f(x)\) with respect to \(x\), \(f'(x)\), set the value of \(f'(x)\) to zero, and solve for \(x\). In the case of the Netflix problem, the function whose minimum we want to locate, \(e\), is that which plots the error of the rank-\(n\) SVD (where \(n\) is the iteration we are on) as measured against the known values in the actual matrix. (Remember that we have limited the approximation to the forty most prevalent features.) Since solving \(e' = 0\) is really hard (that would actually compute the SVD outright), standard calculus does not help here. Still, one simple way to find the minimum of a function is by tracing along its graph until the derivative changes sign. A neat property of gradient descent is that these traces can be executed very quickly by intelligently guessing the points closest to the minimum. Quickly, the point of minimization can be found.

The Incremental SVD Algorithm exploits this property of quick convergence on minima via gradient descent. To calculate the gradient, we take the derivative of the error function, \(e\), with respect to the user preference vector and also with respect to the movie feature vector. We should be left with two partial derivatives. Here is the basic derivation\(^{36}\):

Let \(R_{ij}\) be the known rating for movie \(i\) by user \(j\), and let \(P_{ij}\) be the predicted rating for that movie and user, based on the one-dimensional approximation given by \(MU\), where \(M\) is a movie column vector of parameters \(m_i\) and \(U\) a user row vector of parameters \(u_j\). The standard SVD formulation says that user \(j\)’s prediction of movie \(i\) is given by:

\[
P_{ij} = m_i \times u_j
\]

Then the error in the prediction is clearly the Frobenius norm of the matrix formed by:

---


\(^{33}\) Brand 1.

\(^{34}\) Gorrell and Webb 1.

\(^{35}\) Venkataraman 1.

\(^{36}\) Wagman 1.
\[ E_{ij} = R_{ij} - P_{ij} = R_{ij} - m_i \times u_j \]

For gradient descent, we want to take the derivative of the squared error with respect to each parameter - i.e. with respect to each user vector value and movie vector value. Below we show how to compute it with respect to a particular user parameter \( u_j \):

\[ \frac{\partial E^2}{\partial u_j} = 2E \times \frac{\partial E}{\partial u_j} \]

Since \( E = R - P \), and \( R \) is a constant, \( \frac{\partial E}{\partial u_j} \) is just \(-m_i\). Putting it all together:

\[ \frac{\partial E^2}{\partial u_j} = -2E_{ij} \times m_i \]

When using gradient descent, we make use of a rather arbitrary parameter \( l \), known as the learning rate, to use as a multiplier controlling the rate of descent:

\[ u_j := u_j - l \times \frac{\partial E^2}{\partial u_j} \]

Now just substitute the original derivative we wanted to find (note the sign change):

\[ u_j := u_j + 2 \times l \times E_{ij} \times m_i \quad \text{for every } i \]

This correction needs to be done for each \( m_i \), which is equivalent to

\[ u_j \leftarrow u_j + 2l \sum_i E_{ij} \times m_i \]

The gradient with respect to movies is derived similarly:

\[ m_i := m_i + 2 \times l \times E_{ij} \times u_j \quad \text{for every } j \]

This correction needs to be done for each \( u_j \), which is equivalent to

\[ m_i \leftarrow m_i + 2l \sum_j E_{ij} \times u_j \]

The learning rate is empirically found to perform best when equal to 0.001, which suggests that this value allows for fast descent without overshooting (on the next page, the factor of 2 is just swallowed up in the learning rate). All said and done, the complete algorithm has the following structure:
define GLOBAL VARIABLES
public_dataset = the Netflix dataset in matrix form with movies down the side and users across the top
n = the number of features to be trained. We use 40.
l = learning rate which controls the rate of descent. We use 0.001.
movie_matrix = a zeroed matrix w/ dimensions # of movies x n
user_matrix = a zeroed matrix w/ dimensions n x # of users

define train_all():
    R = public_dataset
    for k in range 1 to n:
        R = train_feature(R, k)  //calculate residuals
    return user_matrix * movie_matrix  //the rank-num_features SVD of
          //the public dataset.

define train_feature(R, k):
    M = movie column vector //length equal to number of movies = |M|
    U = user row vector  //length equal to number of users = |U|
    G_c = 0 //column gradient vector. |G_c| = |M|
    G_r = 0 //row gradient vector. |G_r| = |U|
    repeat:
        for j = 1 to |U|:
            for i = 1 to |M|:
                G_r[j] += E[i,j] * M[i]
                U = U + l * G_r
        for i = 1 to |M|:
            for j = 1 to |U|:
                G_c[i] += E[i,j] * U[j]
                M = M + l * G_c
        /*
        Even though M and U are actually vectors, we treat them as 1-d matrices here and multiply them together to form a |M| x |U|
        matrix.
        */
        if R - M*U stops decreasing:
            then break loop
    /*
    Stick these 1-d matrices (M and U) onto their corresponding components of the decomposition.
    */
    movie_matrix[k] = M  //update the kth feature column
    user_matrix[k] = U  //update the kth feature row
    return R - M*U  //Residual error
Visually, the series of calls looks like this:

The entire red curve plots the error of the model compared to the dataset. Each red hump represents the training of one feature. For our algorithm, we would have forty such humps. Each red dot represents a reduction of the model’s error, which is produced via the adjustment performed by gradient descent for that particular vector value. Just like in the golf SVD example, the rank-\(n\) approximation serves to minimize the errors of the rank-\(n-1\) approximation.

To summarize, we are given a sampling of the actual values in the observed matrix, \(R\). From there, we follow the gradient of the free parameters (the two oblong matrices) to minimize the error for those actual values in \(R\). It is the nature of the model (I.e. in combining the oblong user and movies matrices into a single users x movies matrix) that by attempting to fill in any of the target values (the actual values), the user and movie models must be independently developed. Since both the user and movie models are full matrices (no empty values), we can take the dot product for every combination of user and movie to fill out all the previously blank values of \(R\).

Thinking in terms of complexity, it is clear that for Big-O, the time complexity is determined by the number of ratings in the dataset. The more ratings the algorithm has to train on, the longer each vector update takes. Although code snippet 1 seems to suggest that every rating is looked at multiple times, this is actually not the case because the error for the \(k\)th approximation approaches its minimum without having to inspect anymore than a constant factor
$N$ pairs. Using our earlier notation, complexity is $O(N)$ where $N$ is the number of ratings. It is important to note that there is a constant scalar in front of this $N$ which depends on the variable nature of the dataset. For some datasets, the error from the gradient descent algorithm may be more easily represented by the model, so we can guess pretty accurately where the point of convergence will be for the gradient descent, so the learning rate can be set to a larger value. Consequently, the algorithm will run faster. For datasets in which the model does not easily represent the data, guessing the point of convergence is tricky, so the learning rate must be set to a smaller value. The algorithm will not run as fast.

7. CONCLUSION

With each passing day, the ubiquity of recommender systems on the Internet grows. In just the last six months of working on this thesis, I have noticed a deluge of recommendations flooding my day-to-day browsing. YouTube now recommends videos for me, Facebook suggests friends for me, and Gmail suggests more recipients for my e-mails. While recommender systems have clearly already established their place in e-commerce, they now appear crucial to the continued prosperity of our digital leisure and social networks. What’s next?

This paper has outlined three important algorithms capable of powering recommender systems. Tapesty, the first recommender system, used a user-to-user collaborative filtering algorithm to recommend articles to its readers. While effective in its heyday, Tapesty’s long-term utility was modest. Amazon stress-tested Tapesty’s user-based approach, and quickly found it computationally unfeasible for a large-scale system. This infeasibility stems in large part from the volatility of human taste. Naturally, Amazon’s next course of action was to find something more unwavering than human taste on which to base a recommender system. Items seemed like a good option. Working off the fact that two items are in fact similar because they share relatively few key qualities, Amazon was able to design a recommender system based on item similarity that did not need round the clock updating to continually provide recommendations indicative of its active users’ current taste. Today, Amazon’s item-based system scales well to the number of users and items, and returns accurate recommendations in real time.

But as John Donne said, “Never send to know for whom the bell tolls; it tolls for thee.” When Netflix announced a competition to increase the accuracy of its proprietary movie recommendation system by ten percent, many eager prize-seekers tried an item-based approach. But their “solutions” fell short. Clearly, an item-based algorithm was not going to win the Netflix Prize. On April 23, 2007, six months after the competition’s start date, a man by the name of Simon Funk (actually an alias for his real name, Brandyn Webb) vaulted out of nowhere to number three on the leader board literally overnight.

In the spirit of a true academic, he freely published his code and ideas on his personal blog, hoping that others would piggyback off his progress. His algorithm approached the Netflix dataset in a novel way. Presumably, Simon thought, there are some generalities among the 100 million rating in the dataset. If we accept this assumption, then there ought to be some way of representing these 100 million ratings in a lot less than 100 million numbers. “[F]or instance, a single number specifying how much action a particular movie has may help explain why a few
million action-buffs like that movie”\textsuperscript{37}. The trouble is that it is impossible to simply look at the dataset and pick out these generalities. Luckily, the reverse thought process has true utility:

If meaningful generalities can help you represent your data with fewer numbers, finding a way to represent your data in fewer numbers can often help you find meaningful generalities. Compression is akin to understanding and all that\textsuperscript{38}.

The challenge, according to Simon, was to find some way to compress the given ratings in such a way that those meaningful generalities could be teased out. Then, by nature of the model, the meaningful generalities could be used to predict ratings for the blank user, movie pairs. The model that performs this compression is known as singular value decomposition (SVD).

In our treatment of the SVD, we started by observing the incremental nature of the algorithm using a golf scorecard as our start matrix. At each subsequent approximation, we showed that the error in the matrix, as measured using mean squared error, can be minimized. With each successive approximation, more data (rows and columns) was added to the approximation that served to correct for the error still present in the model. Transitioning to the Netflix Prize, two oblong matrices, one representing user preferences, the other movie attributes, were akin to the PlayerAbility and HoleDifficulty components of the SVD for the golf example. Using gradient descent techniques and a dash of calculus, we arrived at the heart of the code that powers Simon’s Incremental SVD Algorithm.

As a teaser to the blog post outlining his algorithm, Simon listed the top few movies from the first thirteen categories inferred by his algorithm\textsuperscript{39}. Needless to say, attempting to explain with words what qualities are shared by the movies in these categories is, for all intensive purposes, impossible for us humans. This is both troubling and cool. For many decades, computers simply did whatever humans programmed them to do. They never understood the concept of addition; they just did it. But here, the outcome of applying the SVD to the Netflix dataset yields relationships understood only in the deep circuitry of the computer, but completely illegible to human brain. In a sense, the tables have turned.

Will we ever be able to infer what are those specific qualities making up the top categories from the Incremental SVD algorithm? Gary Lynch and Richard Granger, who are neuroscientists at the University of California, Irvine, and Dartmouth, respectively, wager that we likely will not. In their book “Big Brain: The Origins and Future of Human Intelligence,” they talk about humans’ little-known prehistoric ancestor called the Boskops, named after the South African town where they were discovered. The skulls of this species are some thirty percent bigger than ours, and they may well have been smarter than we are. But, as Lynch and Granger write, “it may have been their very intelligence that did them in. One hypothesis is that they were so thoughtful and peaceable that when we came along — homo sapiens, that is, with our smaller brains, inconveniently wired through the limbic system, making us warlike and aggressive — we simply wiped them out”\textsuperscript{40}. The moral of this story is that though the Boskops’ brain may have been superior to ours (could they understand the SVD categories?), “[w]e are pretty much stuck with our brains”\textsuperscript{41}.

\textsuperscript{37}Ibid. 2. 
\textsuperscript{38}Ibid. 3. 
\textsuperscript{39}Funk, “Netflix Challenge,” 1. 
\textsuperscript{40}McGrath 1. 
\textsuperscript{41}Ibid. 1.
But if we were going to start over, and re-build ourselves, what would the brain look like? As Charles McGrath of the New York Times suggests, “You would want to design the brain differently, more nearly along the lines of a computer, and you would want to give it more power.” Interestingly enough, parts of the brain have been simulated on the computer, and the results are fascinating. The same Lynch and Granger from “Big Brain” together built a simulation of one layer of neurons, called layer II, from rats’ pyriform, or olfactory, cortex - an area of the brain positively identified as linked to the sorting and storage of smells. In one experiment “the researchers gave the computer a number of odors whose common properties would have stimulated the same 60 percent of cells in a rat's olfactory bulb”. But after the computer encoded these scents, the fraction of shared neurons rose to 85 percent. Categorizations were taking place that were not evident in earlier testing of actual mice. The mice were capable of more powerful categorizations that the scientists originally thought. On subsequent virtual sniffs, the fraction of shared neurons dropped to almost twenty percent, indicating that while mice initially group similar odors together, subsequent sniffs serve to differentiate odors.

Even more akin to the SVD, this experiment “illustrates how the brain uses a specific rhythm of electrical activity to orchestrate and synchronize the many biological events that must come together in learning”. Specifically, the experimenters found that the optimal frequency for sniffing that strengthens connections, or synapses, between nerve cells is the rate actually used by mice. So what we see here is that the brain has evolved to take advantage of the optimal learning rate, if you will, of its chemical circuitry.

Human discourse clearly shows that we have a tendency to equate our brains with the structures around us. A Google search for scientific fill-ins for the phrase “The brain is like a…” yields answers diverse as a car, a Swiss Army knife, a computer, a tree, a conference call, a switchboard, and a social network where neurons are people. But it seems today that the most accurate description for the brain is that it’s like singular value decomposition. Human taste cannot be described with prose, as shown by Simon Funk’s algorithm’s results. But while we may not be able to describe preference with words (as much as we may think we can), there is some component of the brain that is aware of how categorization is performed. This explains why rats sniff at the rate that they do. They are maximizing the power of the mechanism that categorizes taste.

The possibility that computers may one day help perform the tasks that humans define as distinctly “human,” namely finding love and friendship, seems more plausible by the minute. But this supposition is much less scary when we recognize that SVD may not only be a way of factoring matrices into a series of linear approximations, but may also be the mathematical model of how the human brain actually performs categorization and preference.

---

42 Ibid. 1.
43 Weisburd 2.
Works Cited


