**<http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/index.html>**

**Other good links**

**sigmon funk -**[**http://sifter.org/~simon/journal/20061211.html**](http://sifter.org/~simon/journal/20061211.html)

**SVD tutorial -** [**http://alias-i.com/lingpipe/demos/tutorial/svd/read-me.html**](http://alias-i.com/lingpipe/demos/tutorial/svd/read-me.html)[**http://www.miislita.com/information-retrieval-tutorial/svd-lsi-tutorial-1-understanding.html**](http://www.miislita.com/information-retrieval-tutorial/svd-lsi-tutorial-1-understanding.html)

**Introduction**

**Recommender Systems**

**Recommender systems** are a hot topic in this age of immense data and web marketing. Shopping online is ubiquitous, but online stores, while eminently searchable, lack the same browsing options as the brick-and-mortar variety.  Visiting a bookstore in person, a customer can wander over to the science fiction section and casually look around without a particular author or title in mind. Online stores often offer a browsing option, and even allow browsing by genre, but often the number of options available is still overwhelming.

Commercial sites try to counteract this overload by showing special deals, new options, and staff favorites, but the best marketing angle would be to recommend items that the user is likely to enjoy or need. Unless online stores want to hire psychics, they need a new technology. The field of data mining has a developing field of research in recommender systems, which fits the bill.

**Recommender systems are systems that, based on information about a user's past patterns and consumption patterns in general, recommend new items to the user.** Some systems incorporate information about the items in question, others are based only on usage patterns; the latter kind of system is known as a collaborative filtering system.  Instead of asking the user to explicitly pick filters for a search, [collaborative filtering](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/collaborativefiltering.html) uses information about the user's past behavior and similar users to make suggestions.

**Our project**

Applications of recommender systems can be found outside the online retail trade, although that is one of the most popular places to find them. The [comprehensive exercise ("comps")](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/about.html) assignment for [our group](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/about.html) was to build a collaborative filtering system to recommend courses for students at Carleton College. The end product would allow a current student to enter his or her transcript and - based on which classes had been taken and what grades had been earned - a list of classes in which the student would potentially do well would be returned. Clearly, there are some ethical issues at stake here, as the group would have to have access to old transcript data from real Carleton students in order to build a working recommender. Privacy both from the comps group and the end user was a serious challenge. After [exploring several anonymity-preserving algorithms](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/anonymization.html), the group expanded to include datasets of movie ratings provided by MovieLens and Netflix. Also available is an [account](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/datastructures.html) of the challenges we faced in terms of data storage and the solutions we used.

# General Collaborative Filtering Algorithm Ideas

Recommender systems can be present in all sorts of systems and situations, and thus can be implemented in many different ways. Here is an overview of the methods of implementation, which will help with understanding what we did for our comps project.

## Grand Underlying Assumption of Collaborative Filtering

There is one important assumption underlying all of collaborative filtering, which is this: **users who have similar preferences in the past are likely to have similar preferences in the future**. It is this assumption which allows us to take a user's history and extrapolate into their future and predict items which they might enjoy.

It's pretty obvious why this can be assumed - we can see that in human nature there are people who are like each other and people who are not. And the people who are like each other tend to, of course, be similar. For recommender systems, we just look at the similarity in their preferences on a particular item.

Like all assumptions (especially those relating to extrapolation), it is not necessarily true all the time. However, for the purposes of a recommender system, it is true often enough. The partial correctness of the assumption can be seen in our results - if the assumption was without merit, we could do no better than wild guessing, but we obviously get results better than random prediction.

## Explicit vs. Implicit Data Collection

In order to make any recommendations, the system has to collect data. The ultimate goal of collection the data is to get an idea of user preferences, which can later be used to make predictions on future user preferences.

There are two ways to gather the data. The first method is to ask for **explicit** ratings from a user, typically on a concrete rating scale (such as rating a movie from one to five stars). The second is to gather data **implicitly** as the user is in the domain of the system - that is, to log the actions of a user on the site.

Explicit data gathering is easy to work with. Assumedly, the ratings that a user provides can be directly interpreted as the user's preferences, making it easier to make extrapolations from data to predict future ratings. However, the drawback with explicit data is that it puts the responsibility of data collection on the user, who may not want to take time to enter ratings.

On the other hand, implicit data is easy to collect in large quantities without any extra effort on the part of the user. Unfortunately, it is much more difficult to work with. The goal is to convert user behavior into user preferences, but it requires getting over one hurdle: how exactly does one infer preference based on actions in a system? This can be a difficult question to answer.

Of course, these two methods of gathering data are not mutually exclusive. A combination of the two have the possibility for the best overall results - one could gain the advantages of explicit voting when the user chooses to rate items, and could still make recommendations when the user does not rate items by implicitly collecting data.

There are additional small difficulties related data gathering:

* Data collection can only record the actions of a user, and knows nothing of the actual person behind the computer - it is near impossible to properly a user who actually happens to be two people using the same computer.
* Explicit votes may still not be an accurate representation of the true preferences of a user if the person does not know him or herself well enough. Certain studies have shown that people can rate items differently for indiscernible reasons.
* Implicit data collection can involve some privacy issues; any system that would make recommendations must avoid overstepping its bounds.

## Passive vs. Active Filtering

Once you've gathered your data, there are two basic ways of filtering through it to make predictions. The most basic method is **passive filtering**, which simply uses data aggregates to make predictions (such as the average rating for an item). The more advanced method is to use **active filtering**, which uses patterns in user history to make predictions. An example of this would be finding similar users to the current user and using their history to predict a rating.

The distinction between these two methods is subtle, but basically comes down to whether or not the recommendations are user-specific or not. In passive filtering, every user will be given the same predictions for a particular item. In active filtering, the system takes into account your specific history in order to make a recommendation. To put this distinction in a solid example, the news site [Digg.com](http://digg.com) uses passive filtering, showing all users the articles which have received the most votes, whereas the online sales site [Amazon.com](http://www.amazon.com) uses active filtering, trying to recommend products based upon a user's specific actions.

Ultimately, active filtering is what most people mean when they talk about collaborative filtering. Though passive filtering has very useful and practical applications, a *personal* recommendation system can only be implemented using active filtering.

## User-centric vs. Item-centric Filtering

All recommender systems must decide whether or not it will attempt to see patterns between users or between items. A **user-centric system** will find similarities between users then use the similar users' preferences to predict ratings. An alternative to this is an **item-centric system** which will attempt to find the relationships between items, and make predictions then only based on a user's preferences and these relationships.

It is not necessary that a recommender system focus only on users or items, but most typically only find similarities between users or similarities between items and not both.

## Memory-Based vs. Model-Based Algorithms

One big distinction between algorithms is that of memory-based algorithms and model-based algorithms. The basic difference is that memory-based algorithms uses all the data all the time to make predictions, whereas model-based algorithms use the data to learn/train a model which can later be used to make predictions. This means that the memory-based algorithms generally should have all data in memory, whereas model-based can make fast predictions using less data than the original (once you build the model).

The individual advantages and disadvantages of each method will be discussed in their own sections.

# Anonymization

Our initial project was to develop a course recommender using historical transcript data provided by the registrar. When we received these data, they did not contain obvious personal identifiers such as name or student ID, but they did contain the complete schedules and grade history of several thousand Carleton students. The sensitivity of this information made privacy a critical issue. Even without a name or student ID, for instance, it would still be possible to identify a student using their course history, and it might be possible to make such an identification using only the student's major in some cases (i.e. special majors or triple majors). To address this issue we examined several different mechanisms to anonymize this dataset without rendering it useless for collaborative filtering.

## k-Anonymity

In order to address the problem of anonymization, Dr. Latanya Sweeney proposed a framework called **k-anonymity** [4]. Suppose that data is stored in a table where each row is a user and each column is a class. Then the table is said to be k-anonymous if every combination of sensitive attributes is shared by at least k rows in the table. In other words, our table is k-anonymous if every combination of courses and grades is shared by at least k students.

There are two primary techniques for making a table k-anonymous [3]. Suppression involves deleting entries in the table until k-anonymity is achieved, while generalization consists of modifying entries so that they are less specific. For example, a zip code might be generalized by removing one of its digits. Meyerson and Williams showed that finding the fewest number of attributes to suppress in order to make a table k-anonymous is NP-hard [1], and most recent work has focused on developing efficient but sub-optimal algorithms for ensuring k-anonymity.

|  |  |  |  |
| --- | --- | --- | --- |
| **Gender** | **Dorm** | **Zip Code** | **Major** |
| M | Musser | 43983 | CS |
| F | Burton | 54293 | CS |
| F | Myers | 43923 | Math |
| M | Nourse | 40231 | CS |

Table 1: A table of data. Note that this table is not k-anonymous because each row is unique.

|  |  |  |  |
| --- | --- | --- | --- |
| **Gender** | **Dorm** | **Zip Code** | **Major** |
| M | \* | 4\*\*\*\* | CS |
| F | \* | \* | Science |
| F | \* | \* | Science |
| M | \* | 4\*\*\*\* | CS |

|  |  |  |  |
| --- | --- | --- | --- |
| **Gender** | **Dorm** | **Zip Code** | **Major** |
| \* | \* | 439\*\* | Science |
| \* | \* | \*\*\*\*\* | CS |
| \* | \* | 439\*\* | Science |
| \* | \* | \*\*\*\*\* | CS |

Table 2: Two distinct k-anonymizations of the above table.

Our experiments with k-anonymization revealed a limitation of current algorithms. All of the current research on k-anonymity has focused on anonymizing a single table in which each row represents an individual and each column is a distinct attribute. For transcript data, however, each user is identified by a set of courses rather a tuple of fixed length. Students can take a different number of courses, and this adds an extra dimension to the problem. The concept of a column is not well defined, and it is not clear which attributes to generalize or suppress.

One possible solution is to create a table with a column for every possible course, so that each column represents a distinct course and each cell, c(i,j), contains a binary value indicating whether user i has taken course j. After this transformation, every row will be the same length, and a traditional k-anonymization algorithm can be used to find an approximate solution. This approach works, but it has several major limitations. The first is that by greatly increasing the number of columns in the table, the performance decreases. The approximation algorithm proposed by Meyerson and Williams is already cubic in the number of rows [1], and by increasing the time it takes to process each row the problem can become infeasible for large datasets. Even more problematic than speed, however, is that it is impossible to generalize attributes when every value is binary. This is especially troubling for collaborative filtering algorithms, which require measuring the similarity between users. If one student has taken Databases (CS 347 at Carleton) and another has taken Data Mining (CS 377), we would like to be able to say that the two students are somewhat similar since they have both taken upper-level computer science courses, but to anonymize these two rows one would have to delete both attributes.

Solving this problem would require several changes to existing k-anonymity algorithms. Rather than putting each course in a separate column, a better approach would be to compare the sets of courses taken by users and determine which pairs of courses can be generalized to minimize the number of necessary changes. In this case, the courses described above might be generalized CS 300 to avoid deleting them. For large datasets it is not possible to do this exhaustively, but it might be possible to come up with a heuristic to determine pairs of students that are roughly the same and then compare those students exhaustively. This remains an interesting open research problem.

## Perturbation

While developing a k-anonymous collaborative filtering system is still an open problem, there are other techniques we can use to help protect the privacy of sensitive user data. One of the simplest methods is to randomly perturb the data. In their article "SVD-Based Collaborative Filtering with Privacy," Huseyin Polat and Wenliang Du perturbed their dataset by adding a random number chosen according to a Gaussian distribution to each rating [2]. By using a Gaussian distribution with mean 0, each individual rating is obscured but the average ratings for each user and movie should remain roughly constant. In fact, Polat and Du find only a small difference in error between the perturbed and non-perturbed data. Our own experiments have corroborated this. (show results).

Perturbation prevents a malicious party from determining the exact value of a specific rating, but it does not necessarily provide anonymity for the user. In the transcript dataset, for instance, we can easily perturb the grades, but it is not clear how to perturb what courses a user took, and thus they could be identified using their schedule. There is also a question of magnitude. Unless we chose a Gaussian distribution with a very high standard deviation, it would still be possible to distinguish between a student who got mostly As and a student who got mostly Fs, and with sensitive data this may not be acceptable. Perturbation provides some privacy, especially from non-malicious users, but it does not guarantee the strict anonymity of schemes like k-anonymity.

## References

[1] A. Meyerson and R. Williams. On the complexity of optimal k-anonymity. In Proceedings of the Symposium on Principles of Database Systems, pages 223-228, 2004.

[2] H. Polat and W. Du. SVD-based collaborative filtering with privacy. In Proceedings of the 2005 ACM symposium on Applied computing, pages 791-795, New York, NY, USA, 2005. ACM Press.

[3] L. Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. International Journal on Uncertainty, Fuziness, and Knowledge-based Systems, 10(5):571-588, 2002.

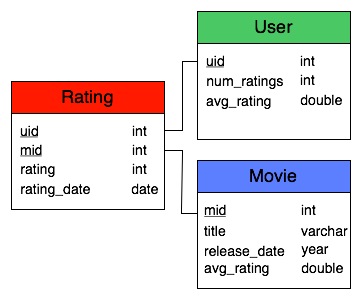
[4] L. Sweeney. k-anonymity: A model for protecting privacy. International Journal on Uncertainty, Fuziness, and Knowledge-based Systems, 10(5):557-570, 2002.

# Data Structures

One of the early challenges we faced when developing our recommender system was choosing data structures that would allow easy and efficient access to our large data sets. This problem became even more important when we started using the Netflix data. With over 100 million ratings, it was important to optimize for both speed and memory usage. Eventually we tried two different techniques for storing our data: a relational database and a hash-table based class called MemReader.

## Database

One of the most natural ways to store a large amount of data is a relational database management system (RBDMS). These software packages use sophisticated data structures and algorithms to access data stored on disk. Programs can access the database using the structured query language (SQL), which supports a wide variety of queries. For example, it is easy to find all of the users who have seen a certain movie or to find the movies that two users have both seen. This flexibility led us to create a MySQL database for the Netflix and Movielens data sets. The schema is shown below:



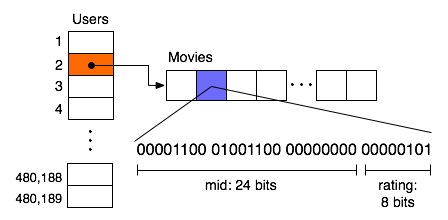
Note that we precomputed the average rating for each movie and user for performance reasons. Ultimately, performance considerations led us to abandon the database. While most database systems are highly optimized, they still store most of the data on hard disk, which is very slow to access. Additionally, we discovered that MySQL did not implement some of the high performance algorithms we needed for complex queries. Since recommendations using the Netflix data can require millions of database queries, we had to find a different approach

## MemReader

(This data structure was proposed on Netflix prize forum by user "voidanswer." It appears to be impossible to link directly to specific posts, but their code is available at <http://www.freelive.org/javaflix.zip>. Our code is significantly different than this version.)

The main reason that database access is so slow is that it accesses data from the hard disk. We can avoid this bottleneck by storing all of the data in memory. Though the Netflix data set is very large, it can be stored in less than two gigabytes if managed carefully. MemReader is designed to provide such access.

In order to provide fast access, we store each rating in two hash tables, one indexed by uid and one indexed by mid. To store the ratings themselves, we observe that neither the uid nor the mid take more than 24 bits to store. Since an int in Java is 32 bits, we shift the uid or mid over to the upper 24 bits and store the rating in the lower bits. Since bitwise operations are very fast, this technique reduces memory usage without slowing down access very much. A diagram of the user-to-movie hash table is shown below. Additionally, we store a hash table containing the number of movies rated by each user and the sum of the ratings for each user (and similarly the number and sum of ratings for each movies). This allows for efficient computation of averages and the ability to add users to an existing MemReader object.



MemReader has proved very effective for the Netflix data set, and we have extended it to work for both the Movielens and transcript data. The MemReader interface does not offer the flexibility of SQL, but we have implemented our own algorithms for specific queries and have seen a significant speed improvement.

**Algorithms**

We experimented with a number of different types of algorithms to build recommender systems. To learn more about them, please click on a link below:

* [Memory-based algorithms](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/memorybased.html)
* [Model-based algorithms](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/modelbased.html)
  + [Item-based collaborative filtering](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/itembased.html)
  + [Personality Diagnosis](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/pd.html)
  + [SVD](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/svd.html)
* [Association Rules](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/association.html)

**Algorithms Criteria**

Regardless of the type of algorithm we tried, there are three important criteria that determine how useful an algorithm is.

**1. Quality of Predictions**

This is a pretty obvious one - of course we want our recommender to make good recommendations. More so, we want it to perform better than any "dumb" prediction algorithm which just uses global data, such as an average rating for items.

**2. Speed/Scalability**

Most recommender systems work in a commercial and/or online setting, and so it is important that they can start making recommendations for a user almost instantly. This means that the algorithm cannot take too long to make any predictions - it has to work, and work fast!

Directly related to speed is the scalability of the algorithm. Again, systems in a commercial and/or online setting can have a huge dataset. The algorithm must maintain its speed even if there are many billions of ratings.

**3. Easily Updated**

The datasets behind recommender systems are constantly being updated with new ratings from users. As such, the algorithm must handle this updated information quickly. If the algorithm required a model that needed several hours to build, it might miss out on its chance to make recommendations based on new information quickly.

**Secondary Criteria**

Naturally there are some other properties we would like our algorithms to have, but are not always obtainable. Here is a brief list of other desired qualities.

* Cold start ability - that is, the ability to start making good predictions to a new user.
* Sparse data handling - sometimes our datasets are very sparse, and but we still want to make good predictions.

# Memory-based algorithms

Memory-based algorithms approach the collaborative filtering problem by using the entire database. As described by Breese et. al [1], it tries to find users that are similar to the active user (i.e. the users we want to make predictions for), and uses their preferences to predict ratings for the active user.

This page will talk about the general ideas; for specific equations and implementations, consult the Breese et. all paper and/or our code.

## Similarity Measurements

In order to measure similarity, we want to find the correlation between two users. This gives us a value from -1 to 1 which determines who alike two users are. A value of 1 means that they both rate in the exactly the same manner, whereas a value of -1 means that they rate things exactly opposite (i.e. one high, the other low or vice versa).

There were two similarity measurements we used. The first was the Pearson correlation coefficient. It is the basic correlation algorithm for samples adapted for rating information. It tries to measure how much two users vary together from their normal votes - that is, the direction/magnitude of each's vote in comparison to their voting average. If they vary in the same way on the items they have rated in common, they will get a positive correlation; otherwise, they will get a negative correlation.

The other similarity measurement is called vector similarity. We can treat two users as vectors in n-dimensional space, where n is the number of items in the database. As with any two vectors, we can compare the angle between them. Intuitively, if the two vectors generally point in the same direction, they get a positive similarity; if they point in opposite directions, they get a negative similarity. To simulate this we just take the cosine the angle between these two vectors, which gives us a value from -1 to 1.

## Predicting Ratings

In order to predict a rating for an item for an active user, we need to find all weights between the active user and all other users. We then take all non-zero weights and have each other user "vote" on what they think the active user should rate the item. Those with higher weights will matter more in the voting process. Once these votes are tallied, we have a predicted vote.

Note that the voting is based on how far off from a user's average they rate a movie - that is, we want to say how far off from the active user's average the active user will rate the item. Thus, with a positive correlation, the active user agrees with however far off the other user voted on a particular item; and with a negative correlation, the active user disagrees (i.e. goes in the opposite direction) from the other user's vote.

## Enhancements

This works relatively well, but there are some enhancements we can make to the weighting techniques.

* **Default Voting:** It turns out that correlation, as a similarity measurement, does not work very well on sparse data sets. That is, when two users have few items in common, their weights tend to be over-emphasized. Default voting simply adds a number of imaginary items that both have rated in common in order to smooth the votes.
* **Inverse User Frequency:** There is an intuition that commonly enjoyed items are less important to weight than rarer items. That is, if everyone liked Star Wars, it doesn't help as much in determining weight as two people enjoying a rare independent film together. To take this into account, we just transform each vote when weighting two users such that commonly rated items are given less importance.
* **Case Amplification:** This is a simple one - we simply amplify each weight by an exponent so that higher weights get higher and lower ones get lower. It tends not to work very well, but you can see a tiny improvement.

## Advantages

* The quality of predictions are rather good.
* This is a relatively simple algorithm to implement for any situation.
* It is very easy to update the database, since it uses the entire database every time it makes a prediction.

## Disadvantages

* It uses the entire database every time it makes a prediction, so it needs to be in memory it is very, very slow.
* Even when in memory, it uses the entire database every time it makes a prediction, so it is very slow.
* It can sometimes not make a prediction for certain active users/items. This can occur if the active user has no items in common with all people who have rated the target item.
* Overfits the data. It takes all random variability in people's ratings as causation, which can be a real problem. In other words, memory-based algorithms do not generalize the data at all.

### References

[1] J.S. Breese, D.Heckerman, and C.Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In Proceedings of the Fourteenth Conference on Uncertainty in Artifical Intelligence, 1998

# Model-based recommendation systems

[Memory-based recommendation systems](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/memorybased.html) are not always as fast and scalable as we would like them to be, especially in the context of actual systems that generate real-time recommendations on the basis of very large datasets. To achieve these goals, **model-based recommendation systems** are used.

**Model-based recommendation systems** involve building a model based on the dataset of ratings. In other words, we extract some information from the dataset, and use that as a "model" to make recommendations without having to use the complete dataset every time. This approach potentially offers the benefits of both speed and scalability.

Although the basic idea behind model-based recommendation systems is the same, there are a number of approaches that we can take to actually build the model and use it. Some examples are:

* **Probability problem**: From this perspective, the problem of predicting a rating for a user-item pair is seen as the problem of predicting the probability of the rating being a particular value. **Bayesian networks** and **clustering** (for example, see [1]) often use this idea. For an algorithm that we implemented and uses this idea, please see [Personality Diagnosis](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/pd.html).
* **Enhancement to memory-based algorithms**: The main idea behind [memory-based recommendation systems](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/memorybased.html) is to calculate and use the similarities between users and/or items and use them as "weights" to predict a rating for a user and an item. The same idea can be used in model-based algorithms: the similarities between users and/or items can be calculated and then stored as a model, and then we can use the stored similarity values to predict ratings. These models can also be built using similarities between items rather than users and in fact, sometimes it is more desirable to do so. For example, the [NetflixPrize](http://www.netflixprize.com) data contains slightly fewer than 5,00,000 users, but only a little over 17,000 movies. This makes it likely that the resulting model over will be items will be smaller than that for users.

A model-based system such as this also often allows **trimming of the model** to make the system more scalable. In particular, we can limit the number of similar entities (users or items) that we store for each entity; in other words we store only **k most similar entities**. Researchers (e.g., [3]) have found (as we also did with [Personality Diagnosis](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/pd.html) that storing a limited number of similar entities often has little effect on the accuracy of predictions.

For an example that we implemented, see [Item-based collaborative filtering](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/itembased.html).

* **Linear algebra problem**: The problem of making recommendations is sometimes viewed by considering the matrices of users and ratings available to us and performing linear algebra operations on them. For an example that we implemented, please see [Singular Value Decomposition](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/svd.html).

## Advantages

* **Scalability**: Most models resulting from model-based algorithms are much smaller than the actual dataset, so that even for very large datasets, the model ends up being small enough to be used efficiently. This imparts **scalability** to the overall system.
* **Prediction speed**: Model-based systems are also likely to be **faster**, at least in comparison to [memory-based systems](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/memorybased.html) because, the time required to query the model (as opposed to the whole dataset) is usually much smaller than that required to query the whole dataset.
* **Avoidance of overfitting**: If the dataset over which we build our model is representative enough of real-world data, it is easier to try to avoid overfitting with model-based systems.

## Disadvantages

* **Inflexibility**: Because building a model is often a time- and resource-consuming process, it is usually more difficult to add data to model-based systems, making them inflexible.
* **Quality of predictions**: The fact that we are not using all the information (the whole dataset) available to us, it is possible that with model-based systems, we don't get predictions as accurate as with model-based systems. **It should be noted**, however, that the quality of predictions depends on the way the model is built. In fact, as can be seen from the [results](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/results.html) page, a model-based system performed the best among all the algorithms we tried.

## References

[1] J.S. Breese, D.Heckerman, and C.Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In Proceedings of the Fourteenth Conference on Uncertainty in Artifical Intelligence, 1998.

[2] M.Deshpande and G. Karypis. Item-based top-n recommendation algorithms. ACM Trans. Inf. Syst., 22(1):143-177, 2004.

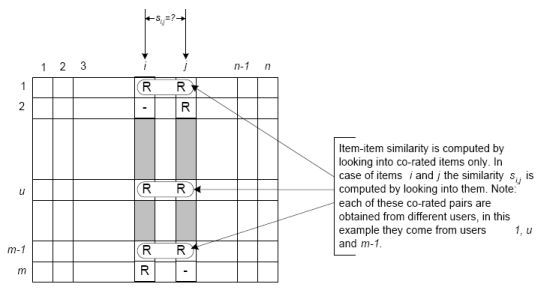
[3] B.M. Sarwar, G. Karypis, J.A. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th International World Wide Web Conference, pages 285-295, 2001.

# Item-based collaborative filtering

**Item-based collaborative filtering** is a [model-based](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/modelbased.html) algorithm for making recommendations. In the algorithm, the similarities between different items in the dataset are calculated by using one of a number of similarity measures, and then these similarity values are used to predict ratings for user-item pairs not present in the dataset.

## Similarities between items

The similarity values between items are measured by observing **all the users who have rated both the items**. As shown in the diagram below, the similarity between two items is dependent upon the ratings given to the items by users who have rated both of them:

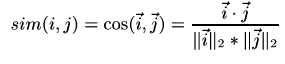


## Similarity measures

There are a number of different mathematical formulations that can be used to calculate the similarity between two items. As can be seen in the formulae below, each formula includes terms summed over the set of common users U.

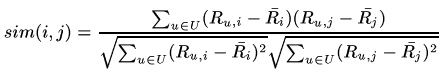
### Cosine-based similarity

Also known as **vector-based similarity**, this formulation views two items and their ratings **as vectors**, and defines the similarity between them as the angle between these vectors:



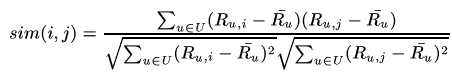
### Pearson (correlation)-based similarity

This similarity measure is based on how much the ratings by common users for a pair of items deviate from average ratings for those items:



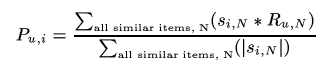
### Adjusted cosine similarity

This similarity measurement is a modified form of vector-based similarity where we take into the fact that different users have different ratings schemes; in other words, some users might rate items highly in general, and others might give items lower ratings as a preference. To remove this drawback from vector-based similarity, we subtract average ratings for each user from each user's rating for the pair of items in question:



## From model to predictions

Once we make a model using one of the similarity measures described above, we can predict the rating for any user-item pair by using the idea of **weighted sum**. First we take all the items similar to our target item, and from those similar items, we pick items which the active user has rated. We weight the user's rating for each of these items by the similarity between that and the target item. Finally, we scale the prediction by the sum of similarities to get a reasonable value for the predicted rating:



## Our implementation

We implemented item-based collaborative filtering using these parameters:

* Adjusted cosine-based similarity
* Minimum number of users for each item-item pair: 5 (see below for explanation)
* Number of similar items stored: 50

## Challenges

We tried item-based collaborative filtering on the Movielens dataset, but as the [Results](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/results.html) page shows, it didn't perform very well in testing. In particular, we isolated two main problems, which were mainly due to the sparsity of the data:

* The first problem manifested itself during adjusted-cosine similarity measurement calculation, in the case when there was **only one common user between movies**. Since we subtract the average rating for the user, the adjusted-cosine similarity for items with only one common user is 1, which is the highest possible value. As a result, for such items, which are common in the Movielens database, the most similar items end up being only these items with one common user. The solution we implemented was to specify a minimum number of users (in this case, 5) that two movies needed to have in common before they could be called similar.
* The second challenge arose when we used weighted sum to calulate the rating for test user-movie pairs. Since we were storing only 50 similar movies for each movie, and for each target movie, we only consider the similar movies that the active user has seen, it was often the case with the Movielens dataset that **there weren't many such movies for many of the users**. This resulted in bad predictions overall for large test sets. Because this was due to the sparsity of the dataset itself, we couldn't come up with a straightforward solution to this problem.

## References

[1] M.Deshpande and G. Karypis. Item-based top-n recommendation algorithms. ACM Trans. Inf. Syst., 22(1):143-177, 2004.

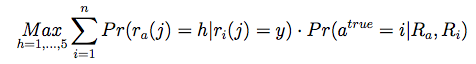
[2] B.M. Sarwar, G. Karypis, J.A. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th International World Wide Web Conference, pages 285-295, 2001.

# Personality Diagnosis

## Basic Algorithm

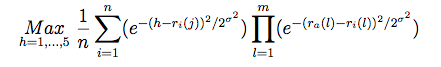
Personality diagnosis (PD) is a probability-based, [model](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/modelbased.html) and [memory](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/memorybased.html) hybrid algorithm. The original PD algorithm was put forth by D. Pennock et al in 2000 [1]. Personality diagnosis works on theassumption that the active user has a hidden variable, known as a "true personality," that can accurately predict the ratings for the user on all items.

For each user in the dataset, calculate the probability that the active user is this user, given their respective rating vectors. Multiply that probability by the probability that the active user will rate the item under consideration as one of the available ratings, given what the comparison user rated the item. Sum that together over all users, and take the rating with the highest probability as the predicted rating for the active user on the item.



Where *h* is a possible rating, *n* is the number of users, *ra(j)* is the rating of the active user on item *j*, and *Ra* is the rating vector for the active user.

This is implemented with the following equation, where the above notation holds, with the additional caveats that movies common to two users are numbered from *l* to *m*, and little sigma is a parameter.  Sigma was chosen to be 2.0 in our trials, adjusted from the experimental results found in the paper [1] to fit the different rating scale of our data.



## Modification

The above approach uses the existing users as models for the active user, but in doing so iterates through all the known users, resulting in the complexity found in memory-based algorithms. To leverage the advantages of model-based algorithms, we can choose to iterate over only a select portion of the existing users. We incorporated the idea of a similarity table from the [item-similarity algorithm](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/itembased.html), adapting it to store similar users. Using the adjusted cosine similarity measurement and similarity tables of size k=50 (see [the page on item-similarity](http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/itembased.html)), we contrasted the results with a memory-based implementation of PD and found a significant speed-up with little loss to accuracy.

|  |  |  |
| --- | --- | --- |
|  | **MovieLens UA test** | **MovieLens UB test** |
| All users | RMSE = 1.11235, time = 388 seconds | RMSE = 1.12708, time = 367 seconds |
| 50 most similar users | RMSE = 1.19959, time = 27 seconds | RMSE = 1.20229, time = 28 seconds |

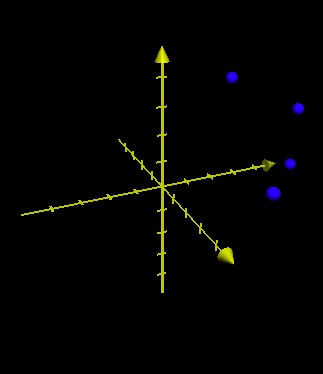
## References

[1] D.Pennock, E.Horvitz, S.Lawrence, and C.L. Giles. Collaborative filtering by personality diagnosis: A hybrid memory- and model-based approach.  In Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence, UAI 2000, pages 473-480, Standford, CA, 2000.

# Dimensionality Reduction and the Singular Value Decomposition

## Dimensionality Reduction

One common way to represent datasets is as vectors in a feature space. For example, if we let each dimension be a movie, then we can represent users as points. Though we cannot visualize this in more than three dimensions, the idea works for any number of dimensions.



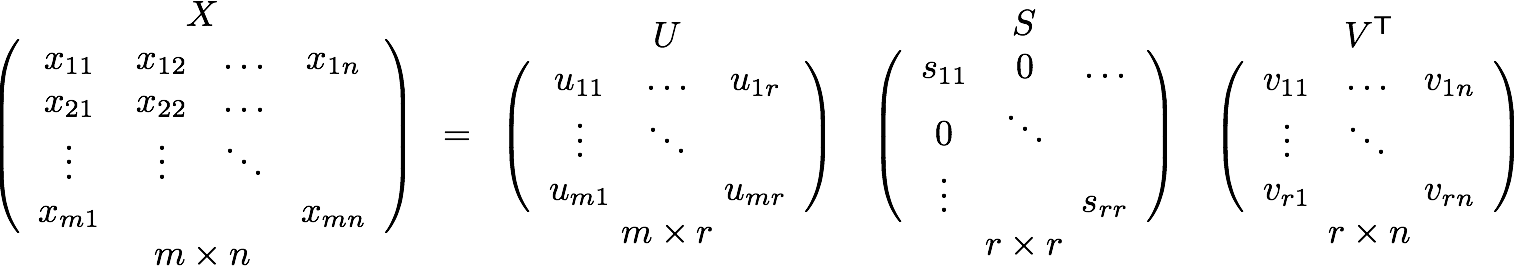
One natural question to ask in this setting is whether or not it is possible to reduce the number of dimensions we need to represent the data. For example, if every user who likes The Matrix also likes Star Wars, then we can group them together to form an agglomerative movie or feature. We can then compare two users by looking at their ratings for different features rather than for individual movies.

There are several reasons we might want to do this. The first is scalability. If we have a dataset with 17,000 movies, than each user is a vector of 17,000 coordinates, and this makes storing and comparing users relatively slow and memory-intensive. It turns out, however, that using a smaller number of dimensions can actually improve prediction accuracy. For example, suppose we have two users who both like science fiction movies. If one user has rated Star Wars highly and the other has rated Empire Strikes Back highly, then it makes sense to say the users are similar. If we compare the users based on individual movies, however, only those movies that both users have rated will affect their similarity. This is an extreme example, but one can certainly imagine that there are various classes of movies that should be compared.

## The Singular Value Decomposition

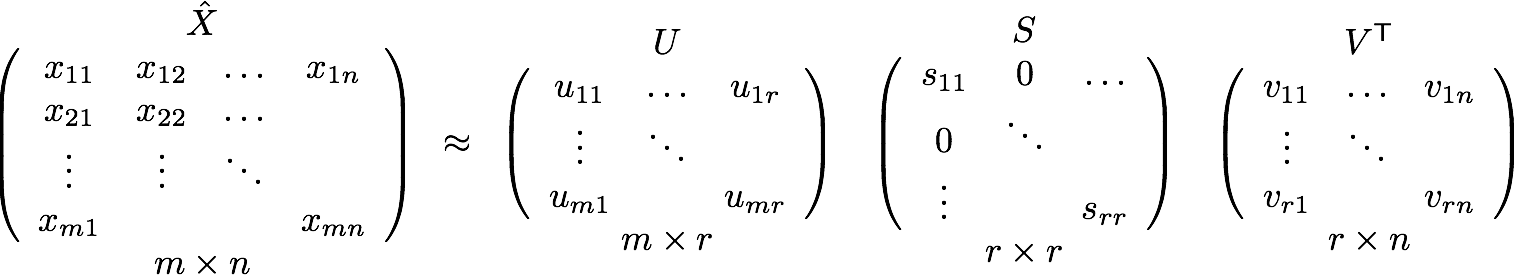
In an important paper, Deerwester et al. examined the dimensionality reduction problem in the context of information retrieval [2]. They were trying to compare documents using the words they contained, and they proposed the idea of creating features representing multiple words and then comparing those. To accomplish this, they made use of a mathematical technique known as Singular Value Decomposition. More recently, Sarwar et al. made use of this technique for recommender systems [3].

The Singular Value Decomposition (SVD) is a well known matrix factorization technique that factors an m by n matrix **X** into three matrices as follows:



The matrix **S** is a diagonal matrix containing the singular values of the matrix **X**. There are exactly r singular values, where r is the rank of **X**. The rank of a matrix is the number of linearly independent rows or columns in the matrix. Recall that two vectors are linearly independent if they can not be written as the sum or scalar multiple of any other vectors in the space. Observe that linear independence somehow captures the notion of a feature or agglomerative item that we are trying to get at. To return to our previous example, if every user who liked Star Wars also liked The Matrix, the two movie vectors would be linearly dependent and would only contribute one to the rank.

We can do more, however. We would really like to compare movies if most users who like one also like the other. To accomplish we can simply keep the first k singular values in **S**, where k<r. This will give us the best rank-k approximation to **X**, and thus has effectively reduced the dimensionality of our original space. Thus we have



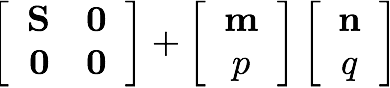
## Recommendations with the SVD

Given that the SVD somehow reduces the dimensionality of our dataset and captures the "features" that we can use to compare users, how do we actually predict ratings? The first step is to represent the data set as a matrix where the users are rows, movies are columns, and the individual entries are specific ratings. In order to provide a baseline, we fill in all of the empty cells with the average rating for that movie and then compute the svd. Once we reduce the SVD to get X\_hat, we can predict a rating by simply looking up the entry for the appropriate user/movie pair in the matrix X\_hat. Further details can be found in [2,3].

## Updating the SVD

One of the challenges of using an SVD-based algorithm for recommender systems is the high cost of finding the singular value decomposition. Though it can be computed offline, finding the svd can still be computationally intractable for very large databases. To address this problem, a number of researchers have examined incremental techniques to update an existing svd without recomputing it from scratch [1,4]. We looked at a method proposed by Matthew Brand for adding and modifying users in an existing SVD [1]. Brand focuses on so-called rank 1 updates, where a single column is modified or added to the orignal matrix. Formally, given the singular value decomposition of a matrix **X**, we want to find the singular value decomposition of the matrix **X+ab**T, where **a** and **b** are column vectors.

The full derivation of Brand's method is beyond the scope of this document, but we will provide a brief discussion of the algorithm. Given the SVD **X=USV**T, let **m=U**T**a**, **p=a-Um**, p=sqrt(**p**Tp) and **P**=**p**/p. Similarly let **n=V**T**b**, **p=b-Vn**, q=sqrt(**q**Tq) and **Q**=**q**/q. Then we first find the singular value decomposition **U'S'V'**T of the matrix



Then the svd of our new matrix is given by

Since we can use the low rank approximations of **U**, **S**, and **V**, this algorithm is quite fast, and Brand shows that the entire SVD can be built in this manner in O(mnk) time, where m and n are the dimensions of the matrix and k is the reduced rank of the approximation.

Singular Value Decomposition

## Extensions

The idea of reducing the dimensionality of a dataset is not limited to the singular value decomposition. While SVDs provide one of the most theoretically grounded techniques for finding features, there are a number of approximation algorithms that can be used on very large datasets. We did not explore this area in great depth, but we did use a method proposed on the Netflix forums by [simonfunk](http://sifter.org/%7Esimon/journal/20061211.html) and implemented in C by [timelydevelopment](http://www.timelydevelopment.com/demos/NetflixPrize.htm). We ported this code to Java and tried it with both the Netflix and Movielens datasets. The fact that this algorithm performed the best among all of those we tried suggests that dimensionality reduction is a powerful idea that would be worth exploring in the future.

## References

[1] M. Brand. Fast online svd revisions for lightweight recommender systems. In Proceedings of the 3rd SIAM International Conference on Data Mining, 2003.

[2] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman. Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6), 1990.

[3] B.M. Sarwar, G. Karypis, J.A. Konstan, and J.Reidl. Application of dimensionality reduction in recommender system - a case study. In ACM WebKDD 2000 Web Mining for E-Commerce Workshop, 2000.

[4] B.M. Sarwar, G.Karypis, J.A. Konstan, and J. Reidl. Incremental singular value deocmposition algorithms for highly scalable recommender systems. In Proceedings of the Fifth International Conference on Computer and Information Technology (ICCIT), 2002.

**SIMON FUNK**

[Followup to [this](http://sifter.org/%7Esimon/journal/20061102.1.html)]

Ok, so here's where I tell all about how I (now we) got to be tied for third place on the [netflix prize](http://netflixprize.com). And I don't mean a sordid tale of computing in the jungle, but rather the actual math and methods. So yes, after reading this post, you too should be able to rank in the top ten or so.

Ur... yesterday's top ten anyway.

My first disclaimer is that our last submission which tied for third place was only actually good enough for ninth place or so. It landed where it did because, just for giggles and grins, we blended results (50/50) with [Jetrays](http://www.findmorefives.com/) who had a similar score to us at the time.

Second, my friend Vincent has been manning the runs on his desktop machines, diligently fine tuning and squeezing out every last bit of performance possible with whatever controls I could give him (not to mention learning python so he could write scripts to blend submissions and whatnot). In short, almost all my progress since my last post has been due to other people. In the meantime I've implemented a handful of failed attempts at improving the performance, plus one or two minorly successful ones which I'll get to.

Now for the method to the mathness, begining with a review of the problem:

Netflix provided 100M ratings (from 1 to 5) of 17K movies by 500K users. These essentially arrive in the form of a triplet of numbers: (User,Movie,Rating). E.g., one such entry might be (105932,14002,3). Times 100 million. Now go make sense of it. In particular, for (User,Movie,?) not in the database, tell me what the Rating would be--that is, predict how the given User would rate the given Movie.

I'm tempted to get all philosophical on my soap box here and go into ways of thinking about this stuff and modeling vs function mapping approaches, yadda yadda, but I know you all are just here for the math, so I'll save that for the next hapless hosteler who asks me what I do for a living.

For visualizing the problem, it makes sense to think of the data as a big sparsely filled matrix, with users across the top and movies down the side (or vice versa if you feel like transposing everything I say henceforth), and each cell in the matrix either contains an observed rating (1-5) for that movie (row) by that user (column), or is blank meaning you don't know. To quantify "big", sticking with the round numbers, this matrix would have about 8.5 billion entries (number of users times number of movies). Note also that this means you are only given values for one in eighty five of the cells. The rest are all blank.

Netflix has then posed a "quiz" which consists of a bunch of question marks plopped into previously blank slots, and your job is to fill in best-guess ratings in their place. They have chosen mean squared error as the measure of accuracy, which means if you guess 1.5 and the actual rating was 2, you get docked for (2-1.5)^2 points, or 0.25. (In fact they specify root mean squared error, affectionately referred to as rmse, but since they're monotonically related it's all the same and thus it will simply hurt your head less if you ignore the square root at the end.)

One additional tidbit is they also provide a date for both the ratings and the question marks, which aside from giving you more rocks to try to squeeze blood from also implies that any cell in the matrix can potentially have more than one rating in it. This would seem to violate the whole matrix analogy but as it happens that's mere foreplay compared to what follows.

Imagine for a moment that we have the whole shebang--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. For instance, any given movie can, to a rough degree of approximation, be described in terms of some basic attributes such as overall quality, whether it's an action movie or a comedy, what stars are in it, and so on. And every user's preferences can likewise be roughly described in terms of whether they tend to rate high or low, whether they prefer action movies or comedies, what stars they like, and so on. And if those basic assumptions are true, then a lot of the 8.5 billion ratings ought to be explainable by a lot less than 8.5 billion numbers, since, 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.

A fun property of machine learning is that this reasoning works in reverse too: 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.

In practice this means defining a model of how the data is put together from a smaller number of parameters, and then deriving a method of automatically inferring from the data what those parameters should actually be. In today's foray, that model is called singular value decomposition, which is just a fancy way of saying what I've already eluded to above: We'll assume that a user's rating of a movie is composed of a sum of preferences about the various aspects of that movie.

For example, imagine that we limit it to forty aspects, such that each movie is described only by forty values saying how much that movie exemplifies each aspect, and correspondingly each user is described by forty values saying how much they prefer each aspect. To combine these all together into 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. E.g., Terminator might be (action=1.2,chickflick=-1,...), and user Joe might be (action=3,chickflick=-1,...), and when you combine the two you get Joe likes Terminator with 3\*1.2 + -1\*-1 + ... = 4.6+... . Note here that Terminator is tagged as an anti-chickflick, and Joe likewise as someone with an aversion to chickflicks, so Terminator actively scores positive points with Joe for being decidedly un-chickflicky. (Point being: negative numbers are ok.) Anyway, all told that model requires 40\*(17K+500K) values, or about 20M -- 400 times less than the original 8.5B.

ratingsMatrix[user][movie] = sum (userFeature[f][user] \* movieFeature[f][movie]) for f from 1 to 40

In matrix terms, the original matrix has been decomposed into two very oblong matrices: the 17,000 x 40 movie aspect matrix, and the 500,000 x 40 user preference matrix. Multiplying those together just performs the products and sums described above, resulting in our approximation to the 17,000 x 500,000 original rating matrix. Singular value decomposition is just a mathematical trick for finding those two smaller matrices which minimize the resulting approximation error--specifically the mean squared error (rather convenient!).

So, in other words, if we take the rank-40 singular value decomposition of the 8.5B matrix, we have the best (least error) approximation we can within the limits of our user-movie-rating model. I.e., the SVD has found our "best" generalizations for us. Pretty neat, eh?

Only problem is, we don't have 8.5B entries, we have 100M entries and 8.4B empty cells. Ok, there's another problem too, which is that computing the SVD of ginormous matrices is... well, no fun. Unless you're into that sort of thing.

But, just because there are five hundred really complicated ways of computing singular value decompositions in the literature doesn't mean there isn't a really simple way too: Just take the derivative of the approximation error and follow it. This has the added bonus that we can choose to simply ignore the unknown error on the 8.4B empty slots.

So, yeah, you mathy guys are rolling your eyes right now as it dawns on you how short the path was.

If you write out the equations for the error between the SVD-like model and the original data--just the given values, not the empties--and then take the derivative with respect to the parameters we're trying to infer, you get a rather simple result which I'll give here in C code to save myself the trouble of formatting the math:

userValue[user] += lrate \* err \* movieValue[movie];

movieValue[movie] += lrate \* err \* userValue[user];

This is kind of like the scene in the Wizard of Oz where Toto pulls back the curtain, isn't it. But wait... let me fluff it up some and make it sound more impressive.

The above code is evaluated for each rating in the training database. Lrate is the learning rate, a rather arbitrary number which I fortuitously set to 0.001 on day one and regretted it every time I tried anything else after that. Err is the residual error from the current prediction. So, the whole routine to train one sample might look like this:

/\*

\* Where:

\* real \*userValue = userFeature[featureBeingTrained];

\* real \*movieValue = movieFeature[featureBeingTrained];

\* real lrate = 0.001;

\*/

static inline

void train(int user, int movie, real rating)

{

real err = lrate \* (rating - predictRating(movie, user));

userValue[user] += err \* movieValue[movie];

movieValue[movie] += err \* userValue[user];

}

Note that predictRating() here would also use userValue and movieValue to do its work, so there's a tight feedback loop in play.

I mention the "static inline" and cram the lrate into err just to make the point that: this is the inside of the inner loop, and every clock cycle counts. My wee laptop is able to do a training pass through the entire data set of 100 million ratings in about seven and a half seconds.

Slightly uglier but more correct, unless you're using an atemporal programming language you will want to do this:

uv = userValue[user];

userValue[user] += err \* movieValue[movie];

movieValue[movie] += err \* uv;

Anyway, this will train one feature (aspect), and in particular will find the most prominent feature remaining (the one that will most reduce the error that's left over after previously trained features have done their best). When it's as good as it's going to get, shift it onto the pile of done features, and start a new one. For efficiency's sake, cache the residuals (all 100 million of them) so when you're training feature 72 you don't have to wait for predictRating() to re-compute the contributions of the previous 71 features. You will need 2 Gig of ram, a C compiler, and good programming habits to do this.

There remains the question of what to initialize a new feature to. Unlike backprop and many other gradient descent algorithms, this one isn't really subject to local minima that I'm aware of, which means it doesn't really matter. I initialize both vectors to 0.1, 0.1, 0.1, 0.1, .... Profound, no? (How it's initialized actually does matter a bit later, but not yet...)

The end result, it's worth noting, is exactly an SVD if the training set perfectly covers the matrix. Call it what you will when it doesn't. (If you're wondering where the diagonal scaling matrix is, it gets arbitrarily rolled in to the two side matrices, but could be trivially extracted if needed.)

Before I decide to re-title this entry Much Ado About Nothing, let me pick up the pace now with a host of refinements:

Prior to even starting with the SVD, one can get a good head start by noting the average rating for every movie, as well as the average offset between a user's rating and the movie's average rating, for every user. I.e., the prediction method for this baseline model is:

static inline

real predictRating\_Baseline(int movie, int user)

{

return averageRating[movie] + averageOffset[user];

}

So, that's the return value of predictRating before the first SVD feature even starts training.

However, even this isn't quite as simple as it appears. You would think the average rating for a movie would just be... its average rating! Alas, Occam's razor was a little rusty that day. Trouble is, to use an extreme example, what if there's a movie which only appears in the training set once, say with a rating of 1. Does it have an average rating of 1? Probably not! In fact you can view that single observation as a draw from a true probability distribution who's average you want... and you can view that true average itself as having been drawn from a probability distribution of averages--the histogram of average movie ratings essentially. If we assume both distributions are Gaussian, then according to my shoddy math the actual best-guess mean should be a linear blend between the observed mean and the apriori mean, with a blending ratio equal to the ratio of variances. That is: If Ra and Va are the mean and variance (squared standard deviation) of all of the movies' average ratings (which defines your prior expectation for a new movie's average rating before you've observed any actual ratings) and Vb is the average variance of individual movie ratings (which tells you how indicative each new observation is of the true mean--e.g,. if the average variance is low, then ratings tend to be near the movie's true mean, whereas if the average variance is high, then ratings tend to be more random and less indicative), then:

BogusMean = sum(ObservedRatings)/count(ObservedRatings)

K = Vb/Va

BetterMean = [GlobalAverage\*K + sum(ObservedRatings)] / [K + count(ObservedRatings)]

But in fact K=25 seems to work well so I used that instead. :)

The same principle applies to computing the user offsets. The point here is simply that any time you're averaging a small number of examples, the true average is most likely nearer the apriori average than the sparsely observed average. Note if the number of observed ratings for a particular movie is zero, the BetterMean (best guess) above defaults to the global average movie rating as one would expect.

Moving on:

20 million free parameters is still rather a lot for a training set with only 100 million examples. While it seems like a neat idea to just ignore all those blank spaces in the implicit ratings matrix, the truth is we have some expectations about what's in them, and we can use that to our advantage. As-is, this modified SVD algorithm tends to make a mess of sparsely observed movies or users. To give an example, imagine you have a user who has only rated one movie, say American Beauty. Let's say they give it a 2 while the average is (just making something up) 4.5, and further that their offset is only -1, so we would, prior to even employing the SVD, expect them to rate it 3.5. So the error given to the SVD is -1.5 (the true rating is 1.5 less than we expect). Now imagine that the current movie-side feature, based on broader context, is training up to measure the amount of Action, and let's say that's a paltry 0.01 for American Beauty (meaning it's just slightly more than average). The SVD, recall, is trying to optimize our predictions, which it can do by eventually setting our user's preference for Action to a huge -150.0. I.e., the algorithm naively looks at the one and only example it has of this user's preferences, in the context of the one and only feature it knows about so far (Action), and determines that our user *so* hates action movies that even the tiniest bit of action in American Beauty makes it suck a lot more than it otherwise might. This is not a problem for users we have lots of observations for because those random apparent correlations average out and the true trends dominate.

So, once again, we need to account for priors. As with the average movie ratings, it would be nice to be able to blend our sparse observations in with some sort of prior, but it's a little less clear how to do that with this incremental algorithm. But if you look at where the incremental algorithm theoretically converges, you get something like:

userValue[user] = [sum residual[user,movie]\*movieValue[movie]] / [sum (movieValue[movie]^2)]

The numerator there will fall in a roughly zero-mean Gaussian distribution when charted over all users, which through various gyrations I won't bore you with leads to:

userValue[user] = [sum residual[user,movie]\*movieValue[movie]] / [sum (movieValue[movie]^2 + K)]

And finally back to:

userValue[user] += lrate \* (err \* movieValue[movie] - K \* userValue[user]);

movieValue[movie] += lrate \* (err \* userValue[user] - K \* movieValue[movie]);

This is essentially equivalent to penalizing the magnitude of the features, and so is probably related to [Tikhonov regularization](http://en.wikipedia.org/wiki/Ridge_regression). The point here is to try to cut down on over fitting, ultimately allowing us to use more features. Last I recall, Vincent liked K=0.02 or so, with well over 100 features (singular vector pairs--if you can still call them that).

Moving on:

As I mentioned a few entries ago, linear models are pretty limiting. Fortunately, we've bastardized the whole matrix analogy so much by now that we aren't really restricted to linear models any more: We can add non-linear outputs such that instead of predicting with:

sum (userFeature[f][user] \* movieFeature[f][movie]) for f from 1 to 40

We can use:

sum G(userFeature[f][user] \* movieFeature[f][movie]) for f from 1 to 40

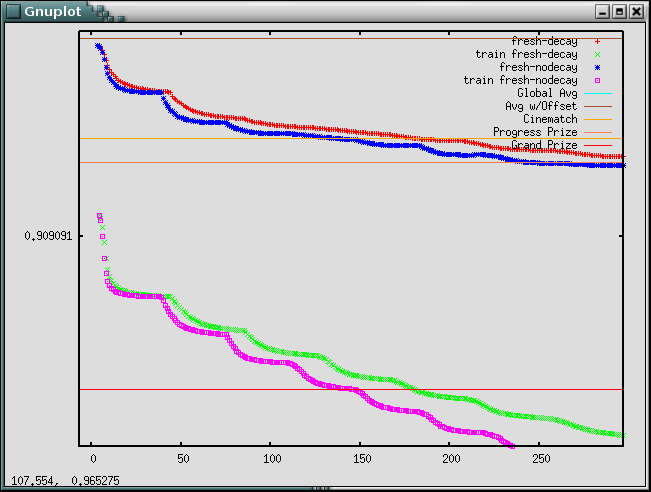
Two choices for G proved useful. One is to simply clip the prediction to the range 1-5 after each component is added in. That is, each feature is limited to only swaying the rating within the valid range, and any excess beyond that is lost rather than carried over. So, if the first feature suggests +10 on a scale of 1-5, and the second feature suggests -1, then instead of getting a 5 for the final clipped score, it gets a 4 because the score was clipped after each stage. The intuitive rationale here is that we tend to reserve the top of our scale for the perfect movie, and the bottom for one with no redeeming qualities whatsoever, and so there's a sort of measuring back from the edges that we do with each aspect independently. More pragmatically, since the target range has a known limit, clipping is guaranteed to improve our performance, and having trained a stage with clipping on we should use it with clipping on. However, I did not really play with this extensively enough to determine there wasn't a better strategy.

A second choice for G is to introduce some functional non-linearity such as a sigmoid. I.e., G(x) = sigmoid(x). Even if G is fixed, this requires modifying the learning rule slightly to include the slope of G, but that's straightforward. The next question is how to adapt G to the data. I tried a couple of options, including an adaptive sigmoid, but the most general and the one that worked the best was to simply fit a piecewise linear approximation to the true output/output curve. That is, if you plot the true output of a given stage vs the average target output, the linear model assumes this is a nice 45 degree line. But in truth, for the first feature for instance, you end up with a kink around the origin such that the impact of negative values is greater than the impact of positive ones. That is, for two groups of users with opposite preferences, each side tends to penalize more strongly than the other side rewards for the same quality. Or put another way, below-average quality (subjective) hurts more than above-average quality helps. There is also a bit of a sigmoid to the natural data beyond just what is accounted for by the clipping. The linear model can't account for these, so it just finds a middle compromise; but even at this compromise, the inherent non-linearity shows through in an actual-output vs. average-target-output plot, and if G is then simply set to fit this, the model can further adapt with this new performance edge, which leads to potentially more beneficial non-linearity and so on... This introduces new free parameters and again encourages over fitting especially for the later features which tend to represent fairly small groups. We found it beneficial to use this non-linearity only for the first twenty or so features and to disable it after that.

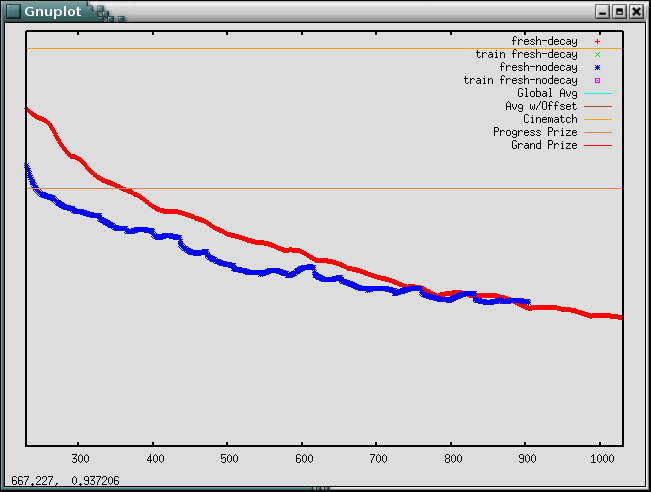
Moving on:

Despite the regularization term in the final incremental law above, over fitting remains a problem. Plotting the progress over time, the probe rmse eventually turns upward and starts getting worse (even though the training error is still inching down). We found that simply choosing a fixed number of training epochs appropriate to the learning rate and regularization constant resulted in the best overall performance. I think for the numbers mentioned above it was about 120 epochs per feature, at which point the feature was considered done and we moved on to the next before it started over fitting. Note that now it does matter how you initialize the vectors: Since we're stopping the path before it gets to the (common) end, where we started will affect where we are at that point. I do wonder if a better regularization method couldn't eliminate overfitting altogether, something like Dirichlet priors in an EM approach--but I tried that and a few others and none worked as well as the above.

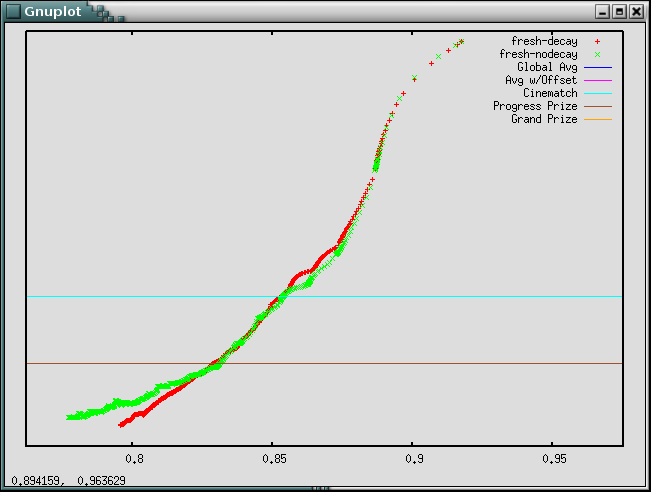
Here is the probe and training rmse for the first few features with and without the regularization term ("decay") enabled.



Same thing, just the probe set rmse, further along where you can see the regularized version pulling ahead:



Same plot again, but this time showing probe rmse (vertical) against train rmse (horizontal). Note how the regularized version has better probe performance relative to the training performance:



Anyway, that's about it. I've tried a few other ideas over the last couple of weeks, including a couple of ways of using the date information, and while many of them have worked well up front, none held their advantage long enough to actually improve the final result.

If you notice any obvious errors or have reasonably quick suggestions for better notation or whatnot to make this explanation more clear, let me know. And of course, I'd love to hear what y'all are doing and how well it's working, whether it's improvements to the above or something completely different. Whatever you're willing to share,

-Simon