Matrix factorization for the Netflix Prize

May 10, 2012

Abstract

I compare two common techniques to compute matrix factorizations for recommender systems, specifically using the Netflix prize data set. Accuracy, run-time, and scalability are discussed for stochastic gradient descent and non-linear conjugate gradient.

1 Intro

A ton of research has come out on collaborative filtering for recommender systems since Netflix announced their million dollar prize in 2006. A quick summary of the contest: given a bunch of 1-5 ratings of movies by users, predict what those users would rate movies for which you don't have the ratings. The goal is to minimize the root mean square error (RMSE). Some of the most successful techniques have been based on matrix factorization methods. The ratings in the recommender system can be imagined as a gigantic $n_m x n_u$ matrix with missing entries, where n_m and n_u are the number of movies and users. These methods attempt to approximate this matrix, including the missing entries, by the product of two much smaller matrices, $Q \in \mathbb{R}^{n_f x n_m}$ and $P \in \mathbb{R}^{n_f x n_u}$, where n_f is the number of "features", or latent factors[KBV09].

There is a huge literature on methods for finding these two matrices. I'm going to look at two of these techniques and compare the accuracy and runtime, and examine their scalability. Everything was done in C++.

Who cares about RMSE

In class it was pointed out that the vast majority of improvement in RMSE came from fairly simple methods, with the last tiny bit needed to win the prize coming from a nasty combination of a bunch of complicated-looking stuff. The suggestion was that this complicated stuff was probably only worth it to win the prize, and perhaps as an intellectual exercise.

As a small defense for that tiny improvement, I'll mention this. In [K08], the eventual winners of the Netflix prize consider the related problem of coming up with the best K recommended films for a particular user. In their model of this problem, even modest decreases in RMSE for the original problem substantially improve the quality of the top K recommendations. If they are to believed, improving RMSE leads to better recommendations, which is after all the point of all this. Of course they're probably a little biased on the matter.

2 Set up

2.1 Notation

I'll use the notation from [K08].

- r_{ui} - actual rating of user u of movie i
- $\hat{r}_{ui}$ - predicted rating of user u of movie i
- K - the set of pairs (u, i) for which r_{ui} is known
- λ - regularization parameter to prevent over fitting

2.2 Data files, representation, training and test sets

The dataset for the Netflix prize is presented in $n_m = 17770$ csv files, one for each movie. Each line is a record of the form <user_id>,<rating>,<date>. I ignore the date information; the pros were only able to get very modest improvement from using it. Even without dates, it is an annoyingly large amount of data, and just parsing it takes a long time. So as a first step, I went through all the files and created and saved an array of length $n_r \approx 10^8$ with an entry for each rating. These entries are structs containing the movie id (short int, 2B), the user id (int, 4B), and the rating (unsigned char, 1B), and so this array takes up 700MB. The array can be loaded from disk into memory almost instantly, making it much more convenient than parsing all the files for each run.

Netflix included a list of $\approx 10^6$ ratings called the "probe set" which was designed to closely resemble their secret test set used for judging submissions. This list of ratings is included in the given training data, so it was necessary to pass through the training data and separate out all the probe ratings. I then split these probe ratings into two equal size sets, one for cross-validation (CV set) and another for reporting final performance (test set).

One annoying detail: the user IDs are not conveniently numbered from 0 to n_u but instead skip all over the place into the millions. It was necessary to map these down to [0,n_u] to use as array indices.

2.3 Baseline predictors

For both algorithms, I first obtained a set of baseline predictors: μ, b_u, and b_m, the overall average, individual user average, and individual movie average. The final prediction for a rating is $\hat{r}_{ui} = \mu + b_i + b_u + q_i^T p_u$. To find the baseline predictors, I ignored the final term and minimized the following loss function:

$$\sum_{(u,i)\in K} (r_{ui} - \mu - b_i - b_u)^2 + \lambda(\sum_u b_u^2 + \sum_i b_i^2)$$

I solved this once in advance using nonlinear conjugate gradient, cross-validating against the CV set to find the best choice of λ. Using only these baseline predictors, and ignoring any user-movie interaction, gave an RMSE of 1.02 on the test set. This isn't much better than the RMSE of 1.05 from using μ alone. I learned these separately from the matrix factorization, though perhaps it would be better to learn all the predictors simulataneously. I intend to investigate this in the future, in addition to determining what impact the baseline predictors have on the runtime and accuracy of the algorithms below.

3 The algorithms

3.1 Stochastic gradient descent

This method repeatedly passes through the training examples, and does a simple update of the predictors after each example. It is easy to implement and can give good results quickly, but has some drawbacks, two of which are:

1. there is no natural step-size for the update, and so this has to be tuned
2. it's not that easy to scale to many machines

There are ways to automatically tune the step-size, but this starts to complicate what is normally a very simple method. I found a reasonable starting step-size through trial and error, and then had this decay by a factor of .98 after each full pass through the data. This seemed to work well enough, though I didn't investigate very thoroughly.

I've read some papers on parallelizing SGD which looked promising[HOGWILD, Jellyfish]. One simple idea that I intend to evaluate is splitting the data into subsets, one for each core, and running SGD independently on these subsets, averaging the solutions together every so often. I believe this is part of the approach John Langford mentioned when discussing how to scale Vowpal Wabbit. I had actually seen the approach in [Jellyfish] back in 2009 on the the Netflix Prize forums[post]:

> - Parallelization. I reasoned that, if I wanted to run on n processors, I could divide the user into n groups and the movies into n groups, so that the user x movie matrix would look like an n x n grid. At any one time, you work on n different squares in that grid, chosen so that no two squares are in the same row or column; it takes n steps with n processors to process the entire set once. For instance: movie_block = thread number; user_block = (step + thread) % n.

3.2 Nonlinear conjugate gradient

This is an algorithm that relies only on gradient information, and so can be used anywhere you might consider using vanilla gradient descent. It's included in a lot of optimization libraries so there should be no need to hand code it; I used a library called ALGLIB[ALGLIB]. A line search is performed, so there is no need to tune a step-size. And for this problem, computing the gradient is an embarassingly parallel task; I used openMP which automatically parallelizes for loops with a single line of code, though there was also a small amount of bookkeeping involved.

The ALGLIB explanation of the algorithm is pretty clear:

> ...the direction to explore is chosen as a linear combination of the current gradient vector and previous search direction:
>
> $x_{k+1} = x_k + \alpha_k d_k$
>
> $d_{k+1} = -g_{k+1} + \beta_k d_k$
>
> α is chosen by minimization of $f(x_k + \alpha_k d_k)$. There are several formulas to calculate β, each of them corresponding to a distinct algorithm from the CG family. This approach, despite its simplicity, allows the accumulation of information about the function curvature - this information is stored in the current search direction.

John Langford said that Vowpal Wabbit initially used CG before switching to L-BFGS, which they found to be superior. In [LNCLPN] the two algorithms as well as SGD are compared on a number of problems and they found that (not surprisingly) it depends on the problem, but that CG tends to be better for high dimensional ($>10^4$ features) problems.

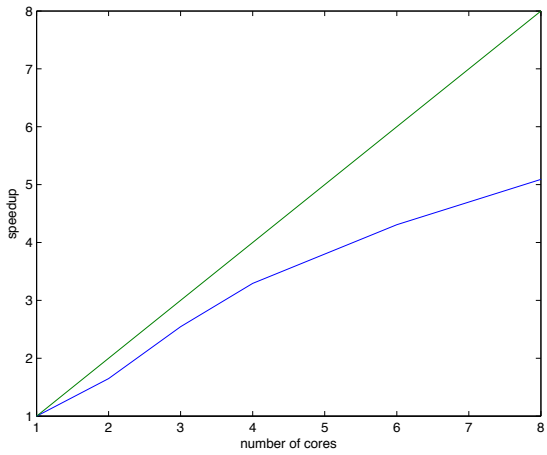

Figure 1: Speedup for function/gradient evaluation for CG

4 Experiments: how many cores does it take for parallel CG to beat serial SGD?

Desktop computers now typically come equipped with multicore CPUs, 4 being fairly common. Today you can get a Mac Pro with 12 cores. Though SGD is simple and works well, it might not be worth sacrificing all those extra cores. Using an off the shelf optimization algorithm, like CG or L-BFGS, could be a better choice if it's really easy to parallelize, like in this case. The machine I ran these experiments on has two Intel Xeon X5570 processors, for a total of 8 cores.

I used $n_f = 32$ features and stopped each algorithm when the training error failed to improve by more than 10^{-4} between iterations. This was large enough that overfitting had not yet set in; see below for more on this. Through trial and error I found regularization parameters λ for each algorithm: .02 for SGD and .035 for CG worked well. I have some code to automatically find the optimal λ's[1], but it's not very robust so I didn't use it here.

The results for CG with varying number of cores were a bit weird. First, the speedup was significantly less than linear (fig 1), surprisingly so. I think I have a couple dumb overhead things in there that I could try to work around. Second, the answer changed based on the number of cores. This is a bit troubling.The P and Q matrices are initialized randomly, but I fixed the seed. I think the way I combine the gradients is a little broken. Anyway, the results are in the same ballpark.

Answer: 3.5 cores

See figure 2.

[1] Using a derivative-free 1D optimization algorithm, same as MATLAB's fminbnd

algorithm	time (s)	train error	cv error	test error
CG, 3 cores	3077	.772566	.924827	.926236
SGD, 1 core	2614	.751035	.927448	.929660
CG, 4 cores	2454	.772383	.924557	.925959
CG, 8 cores	1052	.789230	.930403	.931022

Figure 2: Algorithm performances with 32 features. Each algorithm terminated once training error failed to improve by 10^{-4} between iterations. I used $\lambda = .02$ for SGD and $\lambda = .035$ for CG as regularization parameters.

5 Some observations

5.1 BLAS

Since we're doing a lot of dot products, it makes sense to use an optimized linear algebra library. Macs have a good one, but on Linux I found Intel's MKL to be much faster than ATLAS, the default free option. My free license expired before I thought to take some benchmarks. The dot products $q_i^T p_u$ are between vectors of length n_f, so I looked at the run time per epoch (of SGD) with different numbers of features. It looks like just cycling through the data takes a while, suggesting that a better data representation could be worthwhile.

n_f	epoch time (s)
2	9
4	11
8	13
16	18
32	25
64	42

5.2 Data representation

(this is speculative but I felt like writing it down, please skip)

The representation I used was a huge array of (movie,user,rating) triples. This used around 700MB and required reading three items for each record. An obvious more compact representation would be an array of length $n_u \approx$5e5, with a list of (movie, rating) records for each user. This would use around 300MB and only require reading two items per record.

An even more compact representation again uses an array of length n_u, where each element contains:

1. an array of 5 short ints (maybe unsigned char is enough) of the counts for each rating
2. an array of the movies (also short ints) rated by that user, sorted by rating

The first thing is only $2 \cdot 5 \cdot$ 5e5 = 5MB (plus I guess the pointer to the array, another 4MB). The second thing takes up all the memory, its a short int per rating: $2 \cdot 10^8 = 200$MB.

These other representations can be harder to use, but it's still easy to scan through the data. And for this problem, that's all we really need to do. I implemented the 200MB version a while ago but deleted it after deciding it was too hard to sample. I wish I hadn't done that because I recall it was a good amount faster.

5.3 Overfitting

When using a more stringent stopping criterion than I did, both algorithms monotonically improve the training error on each iteration, but reach a minimum on cross-validation error after which continued iterations increased it. Unfortunately the cross-validation error did not appear to decrease monotonically to its minimum before increasing; it bounced around a little, so it seemed a bad idea to use this as a stopping criterion. However, this is exactly what they did in[TJB09] to achieve an RMSE of .9054 using (I think) the same model as I did with 50 features.

Some ideas on how I might fix this:

- Cross-validate on the whole probe set instead of only half, i.e. get rid of the test set. Less honest, but this is what people seemed to do in the literature.
- Use some sort of moving average of the decrease in cross-validation error as the stopping criterion.
- The bumpiness was only within the first twenty iterations or so, as far as I noticed. I could enforce a minimum number of iterations.

5.4 Poor accuracy

The best RMSE on the probe set I was able to achieve with 50 features was .914, well short of the .9054 in [TJB09]. I'm sure I could do a little better with a better λ, though I doubt I could get all the way there with just that. I wonder what else explains the difference.

6 Conclusion

This is hard. Fiddling with parameters and developing in C++ is not fun. But it was interesting to compare the two algorithms and see at what point the extra cores become worthwhile: 3.5.

References

[KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. Computer, 42(8):30–37, 2009.

[K08] Yehuda Koren, Factorization meets the neighborhood: a multifaceted collaborative filtering model. 2008

[ALGLIB] http://www.alglib.net/

[LNCLPN] Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y. Ng. On optimization methods for deep learning. In ICML, 2011.

[TJB09] A Toscher, M Jahrer, and RM Bell. The bigchaos solution to the netflix grand prize. 2009

[HOGWILD] F Niu, B Recht, C Re, and S Wright. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. 2011

[Jellyfish] B Recht and C Re. Parallel Stochastic Gradient Algorithms for Large-Scale Matrix Completion. 2011

[post] http://www.netflixprize.com/community/viewtopic.php?id=1498