Bridging Recommender Systems and Dimensionality Reduction

March 26, 2017 Data Science & Tech Projects Data Science, Machine Learning

All posts in the series:

1. Linear Regression
2. Logistic Regression
3. Neural Networks
4. The Bias v.s. Variance Tradeoff
5. Support Vector Machines
6. K-means Clustering
7. Dimensionality Reduction and Recommender Systems
8. Principal Component Analysis
9. Recommendation Engines

At a first sight Recommender Systems (RS) and Dimensionality Reduction (DR) have pretty much nothing in common. They solve different problems in different domains of Machine Learning.

If we take a closer look though, we would realize this is far from being the case. It actually turns out that both RS and DR benefit from a particularly powerful mathematical operation called **Matrix Decomposition**. In this post we are going to try to understand why this process is so important and beneficial for such different ML problems.

Most sources of data can be represented as large matrices. Due to their size though, they are generally computationally expensive and inefficient to treat. It is therefore natural to try finding an alternative and somewhat less complex depiction of the problem. An idea could be to write the original matrix as a product of two or more smaller ones. There would be multiple benefits if he could get this done: first of all smaller entities are generally more manageable, second, we would expect the factorization to unveil hidden patterns in the data set. These smaller matrices would contain the same information stored in the original matrix, just organized in a much more condensed, structured and easier to interpret way.

Singular Value Decomposition

One of the most elegant algorithm to obtain just the above is the so called **Singular Value Decomposition**. The theorem behind this technique states that it is always possible to write a matrix $M \in m \times n$ as a product of three submatrices $U\Sigma V^T$, where (see below for an example taken from Wikipedia)

1. The columns of the matrix $U \in m \times n$ represent an orthonormal basis (unitary module and perpendicular to each other) called the *left singular vectors* of M (NB: theoretically SVD spits out $U \in m \times m$. This is the default result for linear algebra libraries such as numpy as well. Practically, it doesn't matter, as, depending on $m < n$ or $m > n$, the additional $|m - n|$ rows/columns in U will be multiplied by zero columns/rows in Σ).
2. $\Sigma \in n \times n$ is a diagonal matrix whose elements are called the *singular values* of M (NB: theoretically SVD spits out $\Sigma \in m \times n$, which is not a square matrix. Practically, this is not relevant as the additional $|m - n|$ rows/columns in Σ are defaulted to 0).
3. The rows of the matrix $V^T \in n \times n$ represent an orthonormal basis called the *right singular vectors* of M.

Consider the 4×5 matrix

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

A singular value decomposition of this matrix is given by $\mathbf{U\Sigma V^*}$

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{\Sigma} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{V^*} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \sqrt{0.2} & 0 & 0 & 0 & \sqrt{0.8} \\ 0 & 0 & 0 & 1 & 0 \\ -\sqrt{0.8} & 0 & 0 & 0 & \sqrt{0.2} \end{bmatrix}$$

For future reference, it is very important to keep in mind that SVD **provides an exact decomposition** of a matrix. There is no approximation in stating $M = U\Sigma V^T$. The right and the left terms are identical.

The relationship with the eigenvectors of the covariance of M

Before moving any further it is important to spend some time on the relationship between SVD and eigen-decomposition. The two operations are very closely related and surfacing these ideas will make our lives easier in the subsequent part of the post.

Let's start from the following (given $M = U\Sigma V^T$)

$$M^T = (U\Sigma V^T)^T = (V^T)^T \Sigma^T U^T = V\Sigma U^T \tag{1}$$

where we exploited the fact that Σ is diagonal, hence equal to its transpose. Now let's right multiply both members by M and obtain

$$M^T M = V \Sigma U^T U \Sigma V^T = V \Sigma^2 V^T \tag{2}$$

where we can simplify $U^T U = I$ due to U being orthonormal and $\Sigma\Sigma = \Sigma^2$ due to its diagonality. Now let's right multiply again, by V this time and obtain

$$M^T M V = \Sigma^2 V \tag{3}$$

Now, remember that given a matrix A, its eigenvalues (λ) and eigenvectors ($\mathbf{e}$) are the entities satisfying $A\mathbf{e} = \lambda\mathbf{e}$. This equation resembles very closely to (3). It is basically telling us that M right singular vectors are $M^T M$ eigenvectors and that Σ^2 are its eigenvalues. To summarize (skipping the proof for U)

- The left-singular vectors of M (U) are a set of orthonormal eigenvectors of MM^T.
- The right-singular vectors of M (V) are a set of orthonormal eigenvectors of $M^T M$.
- The non-zero singular values of M are the square roots of the non-zero eigenvalues of both $M^T M$ and MM^T ($\lambda = \sqrt{\sigma}$).

Having set the above premises, let's see how all of this applies to Dimensionality Reduction first and to Recommender Systems after.

Dimensionality Reduction

We have a matrix $M \in m \times n$ and we want to represent it in a more concise form, say a matrix $M' \in m \times n'$ with $n' < n$. We may want to do that for a number of reasons.

- Data visualization purposes. We cannot really display a matrix $\in \mathcal{R}^d$ where $d > 3$.
- We are working on a ML model which is very computationally expensive to train. A solution could be to reduce the number of features we feed the algorithm with.
- We need to save RAM, hence we have to reduce the size of the data set

The important thing here is that we want to come up with a matrix M' containing as much information as possible from the original M. We just want it to be a condensed form of M.

We can achieve this goal in two ways. Both using matrix decomposition. In both cases we will be performing **Principal Component Analysis (PCA)**. We will just apply it to different matrices.

Eigenvectors of $M^T M$

Remember, our starting point is M. So, first step here is to calculate $M^T M \in n \times n$ (the covariance matrix). Then we compute $M^T M$ eigenvectors (or M right singular vectors V). Let's align $M^T M$ eigenvectors as columns in a matrix (appending from left to right in decreasing order based on eigenvalues) and call the matrix $E \in n \times n$. Now, recall, we started with m points in an n-dimensional space and we want to shrink n to a value $n' < n$. To do this, we just need to select the n' leftmost E columns and project M on this new set of axes. So, in practice, the n' leftmost E columns constitute a matrix $\Theta \in n \times n'$ and $M\Theta = M' \in m \times n'$, which is exactly what we wanted.

How do we interpret what we have just done?

PCA takes a data set consisting of a set of tuples representing points in a high-dimensional space and finds the directions along which the tuples line up best. The idea is to treat the set of tuples as a matrix M and find the eigenvectors for MM^T or $M^T M$.

The matrix of these eigenvectors can be thought of as a rigid rotation in a high dimensional space. When we apply this transformation to the original data, the axis corresponding to the principal eigenvector is the one along which the points are most "spread out". More precisely, this axis is the one along which the variance of the data is maximized (confirmed by the highest eigenvalue associated with it). Put in another way, the points can best be viewed as lying along this axis, with small deviations from it. Likewise, the axis corresponding to the second eigenvector (associated with the second-largest eigenvalue) is the axis along which the variance of distances from the first axis is greatest, and so on.

How do we choose n'? i.e. how much do we shrink the data set keeping under control the amount of information we are losing?

We can achieve that checking the *proportion of variance explained* (PVE) by each principal component. Considering that the total variance of the data set is equal to ($M \in m \times n$)

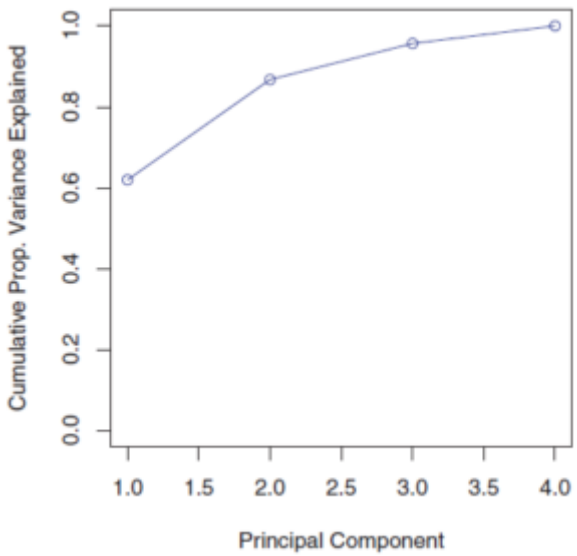
$$\sum_{j=1}^{n} \frac{1}{m} \sum_{i=1}^{m} x_{ij}^2 \tag{4}$$

(x_{ij} are the elements of M) and that the variance associated to the k^{th} principal component is equal to

$$\frac{1}{m} \sum_{i=1}^{m} z_{ik}^2 = \frac{1}{m} \sum_{i=1}^{m} \Big(\sum_{j=1}^{n} \phi_{jk} x_{ij} \Big)^2 \tag{5}$$

where Z_k is the projection of M on the k^{th} principal component Φ_k. Now, dividing (5) by (4) and summing up the first k principal components, gives the cumulative PVE, which looks something like the chart on the right (taken from *An Introduction to Statistical Learning* by G. James, D. Witten, T. Hastie and R. Tibshirani).

As you can see, the more components we add the more variance we explain. It's up to us to set a reasonable threshold.



Singular Value Decomposition on M

Another possibility consists in applying Singular Value Decomposition to M. If we did so we would obtain the three matrices we have talked about before, $U\Sigma V^T$. Now, if we *read* M as a matrix mapping m data points to a n-dimensional feature space, then we can *read* $U\Sigma V^T$ in the following way:

1. $U \in m \times n$ maps m points to a n-dimensional *concept space*, which can be seen as a condensed version of the original n-dimensional feature space, as if the n concepts grouped together "similar" features.
2. $\Sigma \in n \times n$ diagonal elements can be interpreted as the strength of the concepts in the *concept space*. i.e. how much can the data be summarized by the first concept? How much by the second concept? and so forth.
3. $V^T \in n \times n$ maps concepts back to features. i.e. which features adhere to each concept the most?

SVD provides us with the *best* axes to project our data on, where by *best* we mean that these axes gives the minimum matrix reconstruction error and maximum variance per axis. In practice the axes we are talking about are V columns ($\in n \times n$). So, if we project M onto the subspace mapped by these *best* n axes we get a new data set $M \times V \in m \times n$, mapping the old points to the new *concept space* (note that the same result can be achieved with $U\Sigma$ which returns the projected $m \times n$ data set; so, basically $MV = U\Sigma$ as proven here). Now, please note that the actual dimensionality reduction step still needs to be done!

Recall, Σ diagonal values are the strength of the concepts in the *concept space.* So, if we set the lowest of them to 0 we wouldn't lose much, right? We would give up the least important of the concepts, which is not a big deal. We can actually quantify how "big" the deal is, multiplying back $U \in m \times n-1$ by $\Sigma \in n-1 \times n-1$ by $V^T \in n-1 \times n$ and check the Frobenius norm of the difference between M and the reconstructed matrix. We would see that the error is very small. It is actually the smallest possible error. SVD by definition minimizes the Frobenius norm of the difference between M and $\widehat{M}$ (reconstructed) under the constraint $rank(\widehat{M}) = r$, if we use the r largest diagonal values of Σ to build $\widehat{M}$.

Bottom line, if we apply the above procedure we would shrink the $m \times n$ data set to $m \times n-1$ and then to $m \times n-2$ and so forth. Till when?

A rule of thumb is to keep 80-90% of the *energy* of Σ, where the *energy* is equal to $\sum \sigma_i^2$ (σ_i being the diagonal elements of Σ).

Side note on eigen-decomposition and SVD of a symmetric matrix

You may stumble upon a third way of performing dimensionality reduction. The result would be identical to the two first processes explained above, but it can be pretty mind blowing if you don't stop and think abut it.

The key to this approach is that **if M is symmetric** then $M = U\Sigma V^T = W\Lambda W^T$. So basically, the SVD of a symmetric matrix coincides with it eigen-decomposition (StackExchange to the rescue). This is interesting as, guess what, the covariance of a matrix M, $M^T M$ is actually symmetric. This means $U = V = W$, so the eigenvectors of $M^T M$ can also be calculated applying SVD to it and then picking either U or V.

Nothing beats StackExchange!

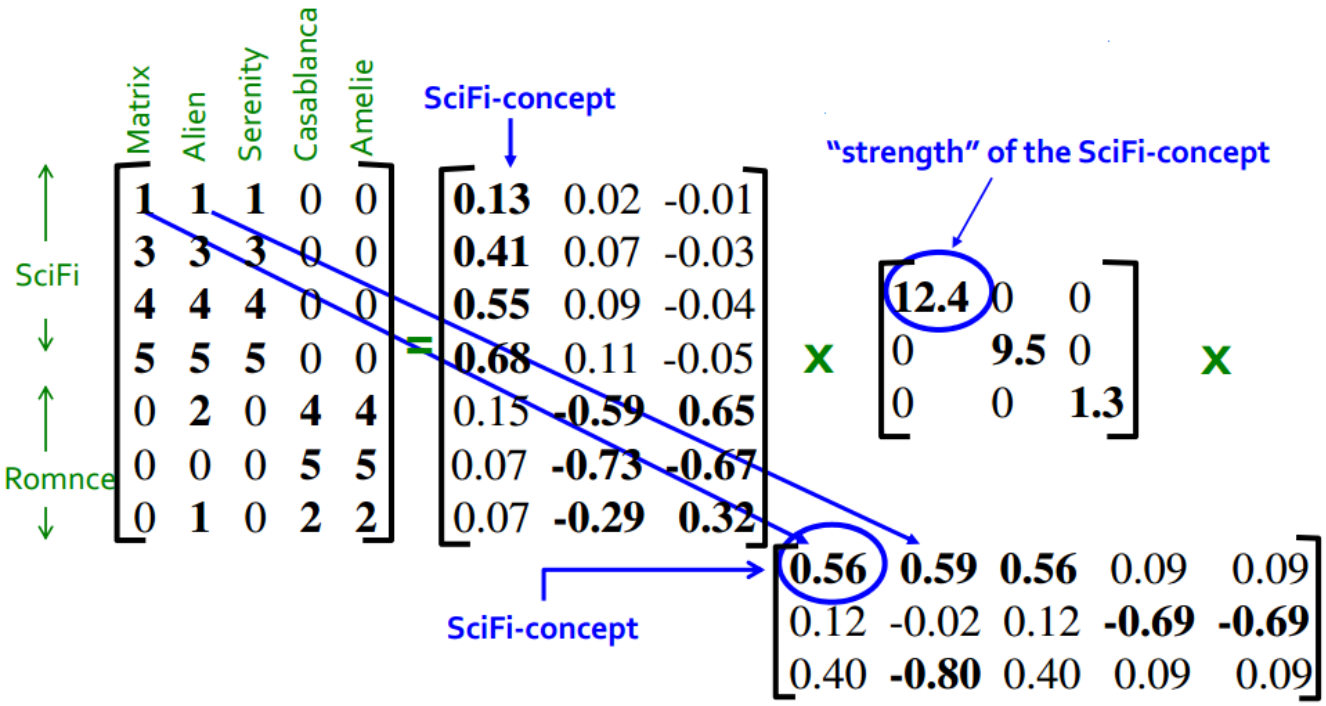
I really did my best to summarize the above concepts, but it would be almost impossible to beat the elegance and transparency of the following posts on StackExchange. Take the time to go over them as they just make everything so cristal clear!

1. How to explain PCA to your family
2. We say that PCA minimizes the reconstruction error and maximises the variance of the data along principal components. Why?
3. The first principal direction is given by the covariance eigenvector with the largest eigenvalue. Why?
4. Can you elaborate on the mathematical proof between eigen decomposition and SVD?

Bridging Dimensionality Reduction to Recommender Systems

I have to acknowledge that everything we have discussed so far can seem pretty abstract. To bring us back from theory to practice, unexpectedly, Recommender Systems (RS) come to the rescue, proving how close they are to Dimensionality Reduction. If SVD sounded a bit esoteric to you, let's see how the *concept space* makes much more sense when applied to a utility matrix users $\times$ movies.

Let's do it and check out the below example (taken from the Mining Massive Data Sets Stanford course material). Here we have 6 users rating 5 movies. It turns out that, out of these, the first 3 are SciFi whilst the other 2 can be classified as Romance. Now, as you can see, the U matrix maps users to the genres concept space, suggesting that the first 4 users really like SciFi. On the contrary V^T maps movies to genres, highlighting that the first 3 movies have a strong SciFi component, whereas the last 2 tend to Romance. What about the third column in U and the third row in V^T? Noise. We can confidently state that, as the third diagonal element in Σ is negligible compared to the other 2, telling us that the "strength" of the this component is very low. (more details on this example here [@min 7:47] and here to dig further into how to leverage the concept space in case we need to "query" our data set to make predictions, i.e. "will a new user like Matrix?").



It is interesting to see how naturally SVD offers a nice and clean interpretation of a Recommendation-Engines-related problem. What we'll do next is actually going down this road a lot more in detail trying bridging DR and RS even further.

First of all, we'll briefly introduce RS, walking through the various available frameworks in order of complexity. Respectively Content Based (CB), Collaborative Filtering (CF) and the modern industry approaches combining CB, CF and Machine Learning in new algorithms known as either Low Rank Matrix Factorization Systems (wasn't SVD factorizing a matrix too?) or Latent Factor Recommender Systems.

This very last piece will let us close the loop with Dimensionality Reduction techniques.

Recommender Systems

Everybody is familiar with Recommender Systems. If you have ever shopped on Amazon you couldn't have possibly missed the list of items the website constantly suggests to you. Some of them make more sense than others, in any case we have to acknowledge that the "you may also like" products are curated in a pretty decent way. How does Amazon guess what I am most likely to buy next? How does Netflix knows I am going to watch the movies it is suggesting to me?

There are three main approaches to answer these questions

1. Content Based Systems
2. Collaborative Filtering

3. Latent Factor Models

We are going to scratch the surface of all of them. Let's get started.

Content Based

The idea behind CB systems is to build a profile for each user and item. A profile is nothing else than a vector with as many entries as the number of features we have chosen to describe the object. Let's suppose our objects are movies and our users are Netflix subscribers. We can think of features as attributes of the object. There are countless features we can come up with: actors, director, year of release, genre and so on so forth. For the sake of simplicity we are going to use just two of them: Julia_Roberts and Denzel_Washington. We summarize every movie in the catalog by only these two boolean attributes. Hence, a film is described by a two dimensional profile whose first entry is 1 if the movie stars Julia Roberts (0 otherwise), and whose second entry is 1 if it stars Denzel Washington(0 otherwise). Ok, that was easy. We have the items' profiles.

How do we get users' profiles?

What we could do is getting all the movies watched by a user and then extract the Julia_Roberts and the Denzel_Washington features averaging them out across movies. So, say, out of the 10 films Francesco watched 3 starred Julia Roberts, then the Julia_Roberts attribute would be 0.3. Applying the same logic to Denzel_Washington, we get 0.7. This is of course a super simplistic version of the story. First of all no company would go ahead with just two features, but most importantly we would need to address a number of potential issues. Do we want a straight average? How do we normalize star ratings? How do we weigh positive ratings versus negative ones?

In any case, after all of it, the result is a two dimensional vector describing Francesco's tastes (seems like he loves Denzel Washington!). How do we now predict which movies Francesco should watch next?

This is just a matter of checking how similar Francesco is to the Netflix catalog. There are a number of ways to quantify similarity. One of the most effective in high dimensional spaces is the *cosine distance*, which is based on calculating the cosine of the angle θ between a pair of vectors. If the two are parallel, then $\theta = 0$ and $cos\theta$ is maximized to 1. The more θ grows, the less similar the vectors are, the smallest $cos\theta$ is going to be. Mathematically, given a user vector $\mathbf{u}$ and an item vector $\mathbf{i}$ the cosine of the angle θ between the two is equal to

$$cos\theta = \frac{\mathbf{u} \cdot \mathbf{i}}{\|\mathbf{u}\|_2 \|\mathbf{i}\|_2} \tag{6}$$

So basically, Content Based Recommender Systems can be summarized as building n-dimensional users' and items' profiles, calculating the cosine distance for every pair, ranking items by similarity in descending order and eventually recommending the top K items to every user.

Collaborative Filtering

Another popular approach to Recommender Systems is called Collaborative Filtering. One of the aspects you may have noticed when dealing with the Content Based framework is that each user is treated individually, cutting any possible connection with other customers. This has some interesting pros, such as the ability to cope with users having very unique tastes, but it also presents some major cons, most notably the need to find meaningful features to summarize items-users. That's generally super hard to get. For instance, if you had to suggest a picture, which attributes would you choose? Good luck with that.

Collaborative Filtering introduces a radically different approach, based on the following very simple observation. If I have to recommend an item to a user, why don't I look for similar customers and check what these people thought about the item in question?

This is the so called user-user collaborative filtering approach. The idea makes sense but as soon as we start thinking about it we realize that it is not that easy to find "similar customers". Well, of course, we can pre-process the data set with some clustering or simply apply user-user cosine distance. We would definitely find the N nearest neighbors of each user, but what if we applied the same logic to products instead of people? Everything would be much easier. Think about it. A person can like both romance and thriller movies, but a movie can either be a thriller or a romance one. The point is that items are by definition much more prone to split in buckets that users. Item are simpler and more predictable than users. This observation is what the item-item collaborative filtering approach leverages, and it turns out to hugely outperform the user-user framework in most cases.

So let's say we have to predict r_{xi}, the rating of user x on item i. Then what we need to do is to find a set  N(i,x) of the top N items most similar to i (and already rated by user x), and calculate an average of these ratings weighted by how similar each item is to item i. Something like this

$$r_{xi} = \frac{\sum_{j \in N(i,x)} s_{ij} r_{xj}}{\sum_{j \in N(i,x)} s_{ij}} \tag{7}$$

where s_{ij} is the similarity between items i and j. Side note: it is a good idea to normalize (subtract mean rating m_i from movie i) items' ratings before calculating similarities. The cosine distance we get after this pre-processing step takes the name of *centered cosine distance* (which, by the way, is nothing else than Pearson's Correlation).

Equation (7) is intuitive and actually very effective!

The biggest CF's pro is that it works with any kind of product, with no need to build any features set (a major problem for CB). Despite this point, CF has some serious drawbacks.

1. Cold start: at least some ratings are needed to provide a recommendation. How do we deal with new users/products? (CB suffers from this issue too)
2. Sparsity: given the utility matrix, it is generally not easy to find similar users/items. The data set is just too sparse as only a tiny fraction of users has rated a tiny fraction of items
3. Popularity Bias: this is the so called "Harry Potter Effect". Due to how equation (7) is built, the system tends to recommend popular items, making it harder to spot hidden gems.

Something worth exploring to address the above issues is the adoption of **hybrid methods**, consisting in making predictions via two or more models and then merging results together to spit out a final recommendation. An example of such an approach is the integration of Collaborative Filtering with a Global Baseline.

Let's directly throw the Global Baseline formulation into the mix, then we will discuss the significance of each term

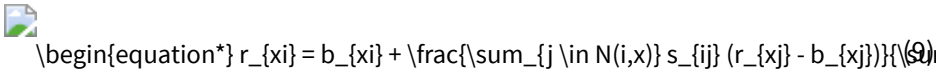
$$b_{xi} = \mu + b_x + b_i \tag{8}$$

Equation (8) is composed of:

1. b_{xi}: global baseline for the rating provided by user x on item i
2. μ: average rating of all items across all users
3. b_x: deviation of user xs's average rating versus μ. i.e. if user x is a though rater, his/her average rating would be lower than the global average, hence b_x will be a negative number, say -0.2

4. b_i: deviation of item is's average rating versus μ. i.e. if movie i had been a success, its average rating would be higher than the global average, hence b_i will be a positive number, say 0.5

Now, given the Global Baseline formulation introduced in (8), we can correct (7) and obtain the following enhanced equation



where we subtract b_{xj} within the summation as we have already added it at the beginning and we don't want to double count.

This is a good improvement over a simple CF, as the introduction of a global view of the data helps solving issues like sparsity and cold start.

Even if we are going in the right direction, the previous approach is still too simplistic for models currently used in industry. We'll see below how to give a significant boost to recommendations engines.

Latent Factor Models – A modern Recommender System

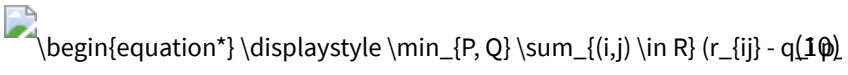
Not sure you have noticed but so far we haven't really applied any machine learning algorithm. Both CB and CF can be seen as way of extracting information from an existing data set via some basic linear algebra. Nobody is really learning anything here.

The idea we want to propose now is to add another layer to the previously explored ones. An optimization layer leveraging matrix factorization.

Let's start from the utility matrix R users $\times$ movies. As we know, R has plenty of missing values and the whole point of Recommender Systems is to predict them. Now, what if we could write R as a product of two smaller matrices, say R = P Q^T (R \in m \times n, P \in m \times k, U^T \in k \times n where k are the *factors* spanning the space we are projecting users and movies into)? Such an achievement would let us automatically populate every entry in R, r_{ij} would just be equal to p_i q_j^T. Wait, this rings a bell! Isn't this exactly what SVD is supposed to do? It actually is. That's great news as we already know that by definition SVD minimizes the RMSE \sum_{ij} (r_{ij} - (U \Sigma V^T))^2. Tying this back to the factorization we wanted to achieve, P would be equal to U and Q^T to \Sigma V^T.

It seems we are done. SVD offers us the solution we were looking for. There is a complication though. The summation within the RMSE is over all the entries in R. In other words SVD assumes that all the entries of the matrix are provided, while we know this is far from being the case. Applying SVD to R means assuming every missing entry equals to a zero rating, which is completely wrong.

Even though we can't directly apply Singular Value Decomposition, we can still tweak it for our needs. After all, its original formulation is pretty close to what we wanted to achieve. The tweak I am talking about consists in re-formulating SVD as the following minimization problem



where now the summation is just over the non-null elements in R. (10) is actually a pretty straightforward least squares optimization problem, which can be solved by gradient descent without much trouble. What are the parameters we are optimizing to minimize (10)? Well, all the entries in P and Q^T, so $(m \times k) + (k \times n)$.

(10) looks already good but a very reasonable enhancement could be added without much effort: global effects. After all, right now, our prediction is just based on $q_i p_j^T$. Here what the same prediction would look like with the addition of the global baseline

$$\begin{equation*} \widehat{r_{ij}} = \mu + b_j + b_i + q_i p_j^T \end{equation*} \tag{11}$$

This is great. Now we can plug (11) in (10), add regularization terms (to prevent overfitting) and get the final optimization problem

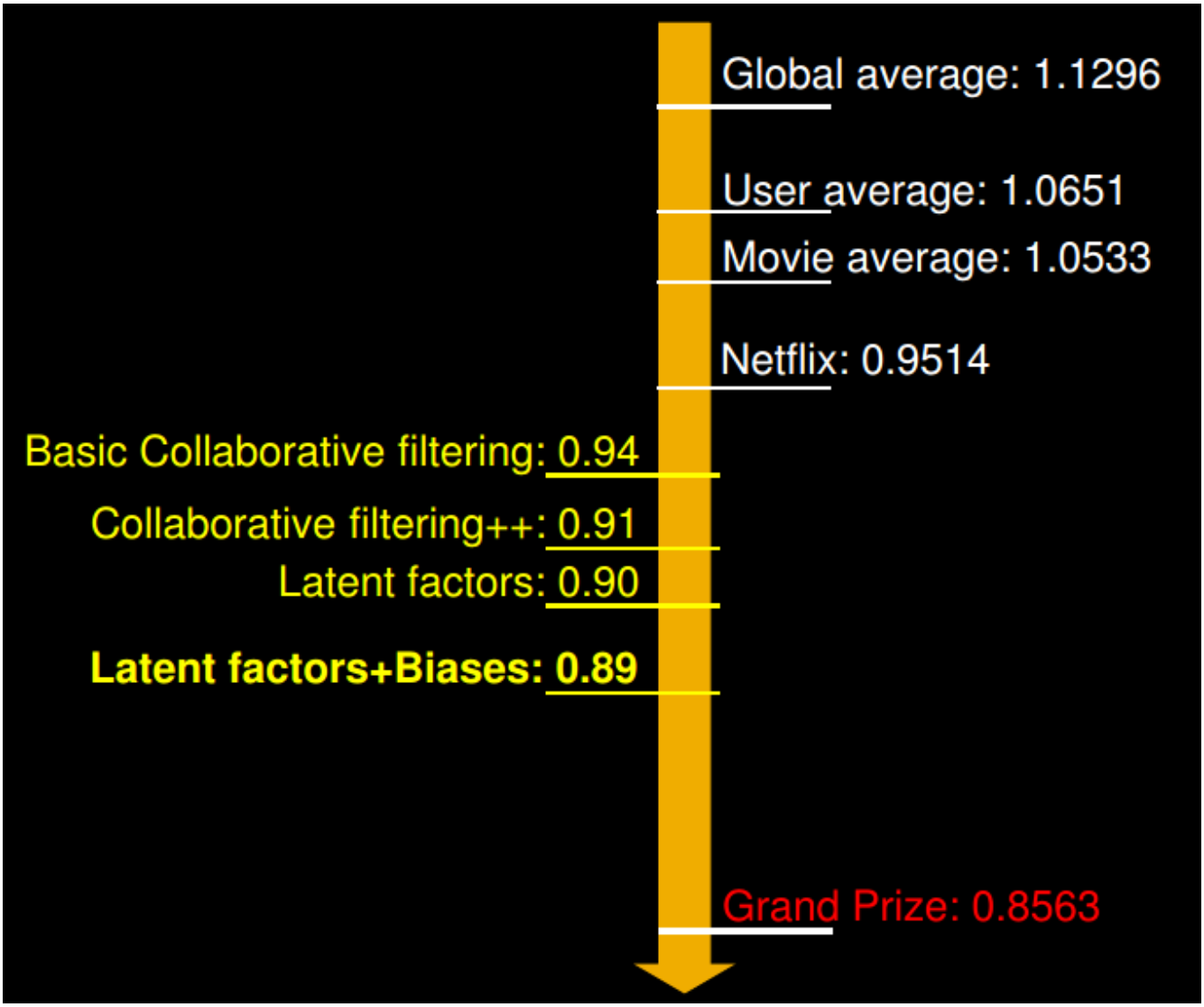
$$\begin{equation*} \displaystyle \min_{P, Q, b_j, b_i} \sum_{(i,j) \in R} (r_{ij} - (\mu + b_j + b_i + q_i p_j^T))^2 \end{equation*} \tag{12}$$

So now we have $(m \times k) + (k \times n) + m + n$ parameters to optimize, where m is the number of movies and n the number of users for each of which we have to calculate a deviation from the global average μ.

To conclude let's have a closer look at how all these models perform.

A perfect example is provided by the Netflix Challenge. In October 2006 Netflix released a an internal data set of users $\times$ movies , offering a 1M$ prize for anyone who could achieve an improvement of more than 10% in the Netflix proprietary recommendation engine, which at the time scored 0.9514 RMSE.

In the image below (taken from the Mining Massive Data Sets Stanford course material) you can check the progress. As you can see a basic CF already beats Netflix lowering the RMSE to 0.94. As soon as the model starts growing in complexity the error goes down, getting to a pretty outstanding 0.89 if we apply (12). The last mile, down to 0.8563, was achieved merging multiple engines together in a giant model with billions of parameters, which was actually never implemented in production.



by ***Francesco Pochetti***

Comments

0 comments