Building recommender systems: Funk's algorithm for simple matrix factorisation

Maximilian Kless

Problem

A recommender system is a tool used to predict interactions - for example ratings - between a set of users and items. In our case, we are given a number of ratings and have to predict how a user will rate an item.

Data

Our dataset consists of 461806 triples of the form (u, i, r) that represent user u's rating on item i. From these data, we are to predict the ratings of 81495 user-item tuples.

Approach

From our triples, we can construct a highly sparse ($< 4\%$ density) user-item-matrix $R \in \mathbb{R}^{users \times items}$ that contains all ratings from the dataset. By using these to approximate the missing values in R we can predict a rating simply by reading off the appropriate cell in the completed matrix $\overline{R}$.
We assume that certain generalities can be derived from existing user-item-interactions. We try to extract these generalities, encode them through a number of latent factors and represent users and items in this factor space. From the users' and items' locations in this space we can predict a users' ratings on unseen items. This approach is identical to filling R, since the product of both user and item latent-factor matrices U and I returns our best approximation of $\overline{R}$. Reading off a cell in $\overline{R}$ then becomes calculating the dot-product of the respective rows/columns in the latent-factor matrices like so:

$$R_{ui} = \sum_{k=1}^{K} U_{uk} \cdot I_{ik} \tag{1}$$

Matrix factorization

As discussed above, we will generate two oblong matrices $U \in \mathbb{R}^{users \times k}$ and $I \in \mathbb{R}^{k \times items}$ where k is the number of latent factors (features) - this number k is predefined and functions as a key parameter in the effectiveness of this algorithm. There are many different ways to generate U and I - we will use Simon Funk's SVD-like algorithm to decompose R into U and V. For each rating, we can compute our prediction error e_{ui}:

$$\epsilon_{ui} = r_{ui} - \hat{r}_{ui} = r_{ui} - \sum_{k=1}^{K} (u_{uk} \cdot v_{ik}) \tag{2}$$

FunkSVD minimizes the *SSE* (sum of squared errors) and thus the *RMSE* - both are monotonically related - through solving the following optimization problem:

$$\begin{aligned}(U^*, V^*) &= \underset{\substack{U \in \mathbb{R}^{m \times k} \\ V \in \mathbb{R}^{n \times k}}}{\operatorname{argmin}} \sum_{(u,i) \in \mathcal{T}} \epsilon_{ui}^2 \\ &= \underset{\substack{U \in \mathbb{R}^{m \times k} \\ V \in \mathbb{R}^{n \times k}}}{\operatorname{argmin}} \sum_{(u,i) \in \mathcal{T}} \left(r_{ui} - \sum_{k=1}^{K} (u_{uk} \cdot v_{ik}) \right)^2 \end{aligned} \tag{3}$$

Taking the partial derivatives results in the following *update rules* that we can insert into Funks algorithm - we then calculate optimal values for matrices U and I through simple gradient descent.

$$\begin{aligned} u_{uk} &+= -\lambda \frac{\partial}{\partial u_{uk}} \epsilon_{ui}^2 = \lambda(\epsilon_{ui} v_{ik}) \\ v_{ik} &+= -\lambda \frac{\partial}{\partial v_{ik}} \epsilon_{ui}^2 = \lambda(\epsilon_{ui} u_{uk}) \end{aligned} \tag{4}$$

Inserting these update rules into the following algorithm will approximate an SVD optimally - if our trainset were to cover all possible ratings, this would be identical to standard SVD algorithms.

```
FunkSVD(k, lrate, iter, M):
Create Matrix U [k x m]
Create Matrix I [n x k]
For each n_k from 0 to k:
    Initialize values in U[n_k] and I[n_k]
    For each rating (u,i,r) in M:
        counter = 0
        While counter < iter:
         # update rules
            err = r - predict_rating(u,i)
            err = err * lrate
            uv = U[n_k][u]
            U[n_k][u] += err * I[n_k][i]
            I[n_k][i] += err * uv
            counter ++
return U and I
```

Here, k is the number of features we want to compute, lrate stands for the learning rate, iter for the number of iterations we want to go through per rating and feature and M is our incomplete user-item matrix.
The function `predict_rating()` predicts a rating for our user-item pair. If we have not yet calculated any or too little feature values, it could do this by using the simple item mean, or - after having calculated a sufficient number of feature values - will return the cross product of all defined values in U_u and I_i. While this prediction can be designed in different ways, all implementations will use the cross product of latent feature vectors sooner or later.

Regularization

To prevent overfitting, especially on users or items with few interactions, we add a regularization term and parameter γ that punishes large values in U and V:

$$(U^*, V^*) = \underset{\substack{U\in\mathbb{R}^{m\times k}\\ V\in\mathbb{R}^{n\times k}}}{\operatorname{argmin}} \sum_{(u,i)\in\mathcal{T}} \left[\epsilon_{ui}^2 + \gamma \left(\left(\sum_{k=1}^{K} u_{uk} \right)^2 + \left(\sum_{k=1}^{K} v_{ik} \right)^2 \right) \right] \tag{5}$$

Taking the partial derivatives of this optimization problem results in a new set of update rules:

$$\begin{aligned} u_{uk} &+= \lambda(\epsilon_{ui} v_{ik} - \gamma u_{uk}) \\ v_{ik} &+= \lambda(\epsilon_{ui} u_{uk} - \gamma v_{ik}) \end{aligned} \tag{6}$$

We can insert these - and the parameter gamma - into the algorithm described above and replace our old update rules.

Parameters

We need to provide multiple parameters to our algorithm - most important of all the number of features, number of iterations, learning rate and a way to initialize the matrices U and V.
As mentioned, the amount of features is a key factor in making successful predictions - extracting too many features will lead to $\overline{R}$ to be heavily overfitted to our dataset while too few features can't hold enough information about existing user-item-relationships. Through experimentation, we have found that using 150 latent factors achieves optimal performance on our data.
The number of iterations establishes how often each feature is adjusted per rating. To avoid long computation times and overfitting, we want to avoid choosing too many iterations. Through experimentation, we have found that performing 20 iterations per rating achieves great computational performance without sacrificing accuracy in our predictions.
The learning rate is *"a rather arbitrary number"* [1] , according to Simon Funk. We have chosen a value of 0.005.
For initialization, we simply fill both matrices with randomly distributed ($\sigma = 0.1, \mu = 0$) values. In his submission for the Netflix Challenge, Simon Funk used a static 0.1 for initialization.
Testing different regularization parameters lead us to choose a value of 0.01.

Performance

The FunkSVD algorithm famously leads to good results - oftentimes outperforming other established methods like *K Nearest Neighbours*. Testing different approaches to learn our dataset, our tweaked FunkSVD outperformed all other algorithms.

Algorithm	Mean RMSE
Tweaked FunkSVD	**0.5036**
Surprise Standard SVD	0.5118
Tweaked User-based KNN	0.5333
Tweaked Item-based KNN	0.5333

We were further able to increase this performance through some post-processing - most importantly rounding predictions that are sufficiently close to whole values off to the nearest integer. This removes the error of some predictions completely, while increasing other errors. The key factor in using this technique successfully is choosing the threshold from which you round up or down. We use a value of 0.1.

Conclusion

While the SVD and related algorithms are already reaching the limits of successful recommendation - as shown during the Netflix challenge, where dozens of laboratories didn't manage to beat SVD's performance through monolithic algorithms - current top-of-the-line algorithms like the Netflix Grand Prize Winner "BellKor's Pragmatic Chaos" make use of ensemble techniques to combine many different recommender systems, collaborative filtering and content-based filtering included. However, staying in the domain of purely collaborative filtering techniques, ensemble methods are also a way to further increase performance. For example, one could train different models (using both multifaceted algorithms and parameter combinations) to gain "different perspectives" on a learning problem and combine those linearly - calculating optimal weights for each algorithm through linear regression

or letting a neural network approximate optimal results through backpropagation. One could also devise hybrid methods where different predictions are made depending on each problems data - the users' amount of ratings, the popularity of the item, etc. - combining predictors in a highly nonlinear fashion. The possibilities are numerous and I have no doubt the field will only continue to grow and produce innovative and interesting ideas.

References

1. Simon Funk. Netflix update: Try this at home.
2. Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.
3. Nicolas Hug. Surprise documentation, 2015.
4. Adam Wagman. Netflix svd derivation.