references.bib

1 Introduction

In this project we want to take a shot on writing a system that recommends items to users. This kind of system is generally called recommendation system. In the vast domain of recommendation systems we specialize in the subdomain of collaborative filtering.

The is characterized by the makings of the input data. Where recommendation systems try to utilize all kinds of input data, in collaborative filtering (CF) the input data only includes the user-interaction with the items in the form of ratings, as opposed to metadata, e.g. gender, country or state of residence or age.

In this, the input data for our system takes the form of a Matrix $I \in \mathbb{K}^{u \times i}$ where $\mathbb{K}$ is the space of all forms a rating can take, e.g. $\mathbb{K} = \{0, \dots, 5\}$. This is typically an interval scale or a binary as like and dislike. Ratings The dimension of I are determined by the number of users u and the count of items i a user can rate. Typically this matrix is sparse as users only rate some of the items.

In our case the data-set is made of a set of triples:

$$\mathcal{S} = \{(u, i, r) \mid u \in \mathbb{N}_+, i \in \mathbb{N}_+, r \in \{0, \dots, 5\}\}$$

It contains 43464 ratings of 1665 distinct users on 1194 items. The highest user-id is 5499 and the highest item-id is 2080. The corresponding matrix for this set $I \in \mathcal{K}^{5499 \times 2080}$ has about 3% entries that are not zero.

2 Method

The range of types of models for CF is vast. The currently best performing models are mostly ensemble methods that combine the strengths of multiple other models. Two of the good performing simple models that are part of those ensembles are models based on matrix factorization or approaches utilizing a distance measure between items and/or users to recommend based on a neighborhood. A good overview can be found in **Toescher2008**, that presented their solution to the Netflix price challenge as a large and well explained ensemble.

For this article we will implement a basic model that combines the strengths of neighborhood-models and matrix factorization by extending the widely known SVD++-Model. It has been proposed in **Koren2008**, a researcher part of the team BelKor that won the Netflix prize.

2.1 Data Structures

The following structures are used to model the CF problem, where k is the number neighbors taken into account and f is the count of latent factors that get estimated in the training process.

$b_{ui} = \mu + b_u + b_i$

$R(u)$ Function that delivers a set of all items that user u rated.

$N(u)$ Function that gives the set of all items that a user provided implicit feedback for.

$S^k(i)$ Function that returns the set of k item most similar to item i in terms of the similarity-measure s_{ij}.

$R^k(u)$ Function that computes $R(u) \cap S^k(i)$.

$w_{ij} \in \mathbb{R}^{i \times i}$ Global user-independent weights for the interaction between items.

$c_{ij} \in \mathbb{R}^{i \times i}$ Global user-independent weights for implicit preference.

$q_i, p_u, y_j \in \mathbb{R}^f$ Factor vectors.

$s_{ij} \in \mathbb{R}^{i \times i}$ The matrix holding the similarity between items i, j.

$n_{ij} \in \mathbb{R}^{i \times i}$ The matrix counting how many times user have rated both items i, j.

$p_{ij} \in \mathbb{R}^{i \times i}$ The matrix holding the Pearson correlation between items i, j.

The neighborhood for the function $S^k(i)$ is defined by the similarity measure $s_{ij} = \frac{n_{ij}}{n_{ij}+\lambda_2} \cdot p_{ij}$. The Pearson-correlation is weighted by the count of users that rated both items i, j.

Generally other measurements for the similarity can be used, e.g. cosine or manhattan distance.

2.2 Prediction

The prediction for an unknown rating $\hat{r}_{ui}$ is modeled as follows;

$$\begin{aligned}
\hat{r}_{ui} &= \mu + b_u + b_i && (1)\\
&+ q_i^T \left(p_u + |N(u)|^{-1/2} \sum_{j \in N(u)} y_j \right) && (2)\\
&+ |R^k(i;u)|^{-1/2} \sum_{j \in R^k(i;u)} (r_{uj} - b_{uj})\, w_{ij} && (3)\\
&+ |N^k(i;u)|^{-1/2} \sum_{j \in N^k(i;u)} c_{ij} && (4)
\end{aligned}$$

User and Item properties Term (**??**) models the basic properties of users and items without interactions as a sum of the global average μ, a user-bias b_u and an item-bias b_i.

User and Item Interactions Term (**??**) describes the interaction between user and item factors. It represents the factorization part of the model. To the user-profile-factor p_u the sum over the factors associated with the items implicitly rated by the user is added.

Weighted nearest items Term (**??**) adds the sum over the nearest neighbors weighted by the global item-interaction weights.

Implicit preference Term (**??**) introduces the implicit preferences. For the nearest neighbors we add all user-independent weights from c_{ij}.

2.3 Training

For the training we do as Koren suggests and minimize the squared error with gradient descent.

$$\min \sum_{(u,i,r)\in\mathcal{S}} (r_{ui} - \hat{r}_{ui})^2$$

From the partial derivatives the following updates follow:

$$\begin{aligned}
b_u &\leftarrow b_u + \gamma_1 \cdot (e_{ui} - \lambda_6 \cdot b_u) \\
b_i &\leftarrow b_i + \gamma_1 \cdot (e_{ui} - \lambda_6 \cdot b_i) \\
q_i &\leftarrow q_i + \gamma_2 \cdot (e_{ui} \cdot (p_u + |N(u)|^{-1/2} \sum_{j\in N(u)} y_j) - \lambda_7 \cdot q_i) \\
p_u &\leftarrow p_u + \gamma_2 \cdot (q_{ui} \cdot q_i - \lambda_7 \cdot y_j) \\
&\forall j \in N(u): \\
y_j &\leftarrow y_j + \gamma_2 \cdot (e_{ui} \cdot |N(u)|^{-1/2} \cdot q_i - \lambda_7 \cdot y_j) \\
&\forall j \in R^k(u): \\
w_{ij} &\leftarrow w_{ij} + \gamma_3 \cdot (|R^k(i;u)|^{-1/2} \cdot e_{ui} \cdot (r_{uj} - b_{uj}) - \lambda_8 \cdot w_{ij}) \\
&\forall j \in N^k(u): \\
c_{ij} &\leftarrow cij + \gamma_3 \cdot (|N^k(i;u)|^{-1/2} \cdot e_{ui} - \lambda_8 \cdot c_{ij})
\end{aligned} \tag{5}$$

The time for one iteration depends on the count of factors to be learned. On an Intel Xeon CPU E5-2630 v2 we get roughly about processed ratings per second. A whole iteration over the dataset with 434641 ratings takes 2.414 hours. For the netflix dataset Koren suggests about 30 iterations. This leads to a training time of about three days.

Potentially the training time could be reduced massively by parallelization. As updates of the data structures are applied for every training point one could partition the training set in batches and apply the updates in parallel while excluding the case of multiple threads working on the same user/item.

2.4 Parameter Estimation

One of the hardest problems with the implementation of the Hybrid model is the large number of parameters. We have to choose values for the count of iterations and factors, as well as the eleven lambdas and gammas steering the training of our model. Hyper parameter tuning is generally a hard problem as it is still subject to active research.

We will try a rather simple ad-hoc approach and use a derivative-less optimization method on the cross-validation RMSE-error calculated on a small subset of the dataset. Using only a subset of the training data does not guarantee good generalization on the whole dataset, but a full training run takes days we have too use less samples to make cross-validation feasible.

3 Implementation

To ease the training and evaluation of recommendation models we implement an abstract base class `Recommender` that provides the abstract function `fit` and `predict`. Furthermorh we add a function `save` to serialize our model and save it into files. Given `fit` and `predict` it should be straight forward to implement a cross validation.h Nonetheless, in the context of recommendation systems we have to be aware of the concept of multiple ratings by the same user. Generally, in Machine Learning we try to predict the performance of algorithms by a strict divide between test and training-data. If we have interactions of the same user in the test and in the training set, this concept is kind of broken. On the other hand, in this case, our algorithm performs good because of the knowledge about the user. If nothing about the previous interactions of the user known, we can not take the neighborhood into account. In this case won't partition the set by user but in other settings it could be necessary. This concept is referred to as *strong generalization* by **Liang2018** We use `numpy.seterr(all='raise')` to avoid proceeding after overflows or occurrences of `np.nan`, as choosing the wrong parameters can lead to numerical errors that we want to be aware of.