Recommender Systems

Linear and Nonlinear Methods

Joseph DiGiovanni, Joo Ho Jeong, Grant Loofbourrow

June 2024

1 Introduction

This project seeks to explore recommendation systems through both linear and nonlinear approaches. For the linear approach, we employed collaborative filtering and the SVD factorization method popularized by Simon Funk. For the nonlinear approach, we utilized artificial neural networks via autoencoders. Recommendation systems are powerful tools for companies to be able to generate engagement, improve customer experience, and to sell products through personalized recommendations. These recommendations are produced by prior implicit feedback, and the two most popular approaches to utilize this data are known as collaborative filtering methods or content-based filtering (CF) methods. Content-based methods require a "questionnaire" of sorts to gather external data required to produce profiles of users. CF-methods compare the relationships between users and items, and only rely on past behavior from users, often in the form of "rating" a particular item. One of the challenges encountered in recommendation systems is that users often tend to rate a very small subset of items, resulting in extreme sparsity within a given data set, especially when analyzing millions of users. Collaborative filtering also allows for top-K recommendation algorithms to be performed, and some methodologies utilize kNN algorithms to group item-item or user-user relationships in the data. However, cosine similarity is most commonly utilized, and was the utilized method of this study.

2 Mathematical Formulation of SVD

2.1 SVD Introduction

This method of collaborative filtering employing models induced by Singular Value Decomposition (SVD) was popularized by Simon Funk in 2007 in his submission to the Netflix Prize competition to produce accurate recommendation systems, where he won 3rd place overall.

2.2 SVD Factorization

Let $A \in \mathbb{F}^{m \times n}$ be a rectangular matrix; the SVD of matrix A is the factorization $A = U\Sigma V^T$, where the columns of U are the eigenvectors of AA^T, the columns of V are the eigenvectors of A^TA, and Σ is a diagonal matrix with diagonal entries $\sigma_1, ..., \sigma_r$, which we denote the singular values of A, and where each $\sigma_j = \sqrt{\lambda_j}$, for $j = 1, ..., r$, the square roots of the nonzero eigenvalues of both A^TA and AA^T.

2.3 Rank-k Approximations to A

Since the singular values of Σ are in decreasing order, this allows us to approximate the matrix A by truncating the number of singular values used by taking the k highest singular values in Σ, allowing us to construct what is known as a *rank-k approximation* of A,

$$A_k = U_k \Sigma_k V_k^T, \tag{1}$$

which is the closest rank-k matrix to A. This allows for approximations of A in a k-dimensional latent space, reducing computational cost while still retaining much of the valuable information about A.

2.4 PQ Factorization Model

The popular PQ factorization method for recommendation systems is a minimization problem utilizing dimensionality reduction known as Latent Semantic Indexing (LSI). The approach used in this experiment is what is known as a model-based approach by looking at a *user-item matrix R*. Large user-item matrices are often extremely sparse and low rank, sometimes exceeding 98 percent emptiness. The SVD factorization method has proven to be very efficient at performing predictions on sparse data sets.

2.5 User-Item PQ Matrix Factorization

Let R be the user-item ratings matrix $|\mathcal{U}| \times |\mathcal{I}|$, with m many users on the rows, and n many items being rated on the column, and where a given entry r_{ui} denotes the rating given by a user to a specific item. The SVD factorization of R is given by

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

where $U = |\mathcal{U}| \times r$ is the matrix of left singular vectors (eigenvectors of RR^T), $V = |\mathcal{I}| \times r$ is the matrix of right singular vectors (eigenvectors of $R^T R$), and Σ is an $r \times r$ dimensional diagonal matrix with the singular values of R. However, this matrix factorization is rarely performed explicitly because of how computationally expensive it is, thus, it is more efficient to work with a rank-k approximation $\hat{R_k}$ of R by selecting the k highest singular values in Σ and their corresponding left and right singular vectors,

$$\hat{R_k} = U_k \Sigma_k V_k^T = U_k \Sigma_k^{\frac{1}{2}} \Sigma_k^{\frac{1}{2}} V_k^T = (U_k \Sigma_k^{\frac{1}{2}})(V_k \Sigma_k^{\frac{1}{2}})^T = PQ^T \tag{3}$$

where $P = U_k \Sigma_k^{\frac{1}{2}}, Q = V_k \Sigma_k^{\frac{1}{2}}$, and both matrices are full rank $k < r$. The u-th row of P is denoted $\mathbf{p}_u \in \mathbb{R}^k$, and the i-th row of Q is denoted $\mathbf{q}_i \in \mathbb{R}^k$. Here, P and Q can be thought of as projecting the rows of $|\mathcal{U}| \times k$ and the rows of $|\mathcal{I}| \times k$ into a k-dimensional latent space, determined by the chosen k singular values of Σ_k, also known as features. Thus, the ratings approximation we seek is the model-based prediction of a given rating r_{ui} found by the dot product of the two vectors,

$$\hat{r}_{ui} = \mathbf{q}_i^T \mathbf{p}_u. \tag{4}$$

2.5.1 Loss Function

The dot product of $\mathbf{q}_i^T \mathbf{p}_u$ captures the information about the user with a given item, modeled as inner products in the $k-$dimensional latent space. P, Q can be found by minimizing the squared Frobenius norm,

$$L(P,Q) = ||R - PQ^T||_F^2 = \sum_{u,i}(r_{ui} - \mathbf{q}_i^T \mathbf{p}_u)^2. \tag{5}$$

However, in practice, P and Q are most often found by a introducing a regularization parameter λ to control the level of regularization (to prevent overfitting of the data), and the resulting error becomes

$$L(P,Q) = \sum_{r_{ui} \in R} (r_{ui} - \mathbf{q}_i^T \mathbf{p}_u)^2 + \lambda(||\mathbf{p}_u||^2 + ||\mathbf{q}_i||^2). \tag{6}$$

2.6 Prediction Algorithm

Seeing as how the ratings matrix R is generally low rank and extremely sparse, there are computational problems that arise from trying to approximate ratings for entries r_{ui} where there may be none. This is resolved by introducing weighted user and item biases b_u, b_i, respectively. Thus the rule for a ratings prediction is the following:

$$\hat{r}_{ui} = \mu + b_i + b_u + \mathbf{q}_i^T \mathbf{p}_u. \tag{7}$$

Here, μ is the overall average across all ratings, b_u, b_i are the biases mentioned above, which are determined by minimizing the regularized error squared,

$$\min_{p_*, q_*} \sum_{(u,i) \in \mathcal{K}} (r_{ui} - \hat{r_{ui}})^2 + \lambda(b_i^2 + b_u^2 + ||\mathbf{p}_u||^2 + ||\mathbf{q}_i||^2) \tag{8}$$

where $\mathcal{K}$ is the set of all (u, i) in the training set, and λ is the regularization term determined through cross validation, and the minimization is calculated by stochastic gradient descent (SGD), or alternating least squares. The ratings prediction equation (7) can be interpreted as a collection of its components; the global average, user biases, item biases, and the user-item interaction captured by the dot product. In this experiment, we utilized SGD since it is computationally efficient to perform.

We first calculate the error term $e_{ui} = r_{ui} - \hat{r_{ui}}$, and then the biases b_i, b_u and factors $\mathbf{p}_u, \mathbf{q}_i$ are determined by modifying the parameters by moving in the opposite direction of the gradient,

$$\begin{aligned}
b_u &\longleftarrow b_u + \gamma(e_{ui} - \lambda b_u) \\
b_i &\longleftarrow b_i + \gamma(e_{ui} - \lambda b_i) \\
q_i &\longleftarrow q_i + \gamma(e_{ui} p_u - \lambda q_i) \\
p_u &\longleftarrow p_u + \gamma(e_{ui} q_i - \lambda p_u),
\end{aligned}$$

where γ is the learning rate.

Algorithm 1 Stochastic Gradient Descent for SVD Factorization

Require: Ratings matrix R with known ratings r_{ui}, learning rate η, regularization rate λ, number of latent factors k

Initialize user bias b_u, item bias b_i, user vectors $p_u \in \mathbb{R}^k$, and item vectors $q_i \in \mathbb{R}^k$ with small random values
for each epoch **do**
    for each known rating r_{ui} in R **do**
        Compute the prediction $\hat{r}_{ui} = \mu + b_u + b_i + p_u^T q_i$
        Compute the error $e_{ui} = r_{ui} - \hat{r}_{ui}$
        Update biases:
        $b_u \leftarrow b_u + \eta(e_{ui} - \lambda b_u)$
        $b_i \leftarrow b_i + \eta(e_{ui} - \lambda b_i)$
        Update user and item vectors:
        $p_u \leftarrow p_u + \eta(e_{ui} q_i - \lambda p_u)$
        $q_i \leftarrow q_i + \eta(e_{ui} p_u - \lambda q_i)$
    end for
end for
return b_u, b_i, p_u, q_i

2.7 Accuracy of Predictions

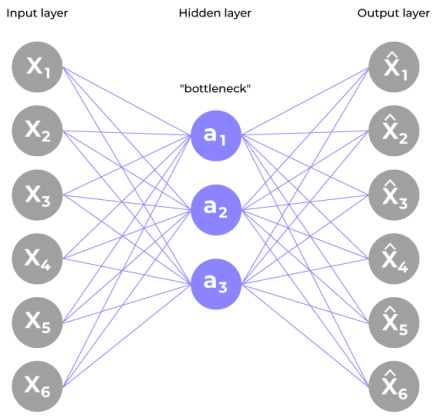
The accuracy of the predicted ratings is typically determined by calculating the Root Mean Squared Error (RMSE),

$$L(r_{ui}, \hat{r_{ui}}) = ||r_{ui} - \hat{r_{ui}}||_2 = \sqrt{\frac{\sum_{u,i}(r_{ui} - \hat{r_{ui}})^2}{N}} \tag{9}$$

Note, most of the above SVD mathematical formulation comes from *Recommender Systems Handbook* [1].

3 Mathematical Formulation of Autoencoders

3.0.1 Designing the Architecture and Training Method



(10)

Autoencoders are a class of neural networks used for unsupervised learning of efficient codings. They learn a compressed representation of input data and attempt to reconstruct the original input from this compressed form. This involves two main components: an encoder and a decoder, which are implemented as neural networks. The encoder maps the input data to a latent space representation, while the decoder maps this latent representation back to the original data space. The mathematical formulation is given for the architecture as follows: [5]

3.1 Encoder

Given an input vector $\mathbf{x} \in \mathbb{R}^n$, the encoder function f maps $\mathbf{x}$ to a latent space representation $\mathbf{z} \in \mathbb{R}^m$:

$$\mathbf{z} = f(\mathbf{x}) = \sigma(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e) \tag{11}$$

where:

- $\mathbf{W}_e \in \mathbb{R}^{m \times n}$ is the weight matrix of the encoder,
- $\mathbf{b}_e \in \mathbb{R}^m$ is the bias vector of the encoder,
- σ is an element-wise activation function (e.g., ReLU, sigmoid).

The encoder is essentially a neural network layer where the input is transformed into a lower-dimensional space. This transformation involves multiplying the input by weights, adding biases, and then applying an activation function to introduce non-linearity.

3.2 Decoder

The decoder reconstructs the input from the latent representation:

$$\hat{\mathbf{x}} = g(\mathbf{z}) = \sigma(\mathbf{W}_d\mathbf{z} + \mathbf{b}_d) \tag{12}$$

where:

- $\mathbf{W}_d \in \mathbb{R}^{n \times m}$ is the weight matrix of the decoder,
- $\mathbf{b}_d \in \mathbb{R}^n$ is the bias vector of the decoder.

The decoder is another neural network layer that takes the compressed data from the encoder and attempts to reconstruct the original input. This involves similar operations as the encoder: weights, biases, and activation functions.

3.3 Loss Function

The goal of training an autoencoder is to minimize the difference between the input $\mathbf{x}$ and the reconstruction $\hat{\mathbf{x}}$. This difference is measured using a loss function, typically the Mean Squared Error (MSE):

$$L(\mathbf{x}, \hat{\mathbf{x}}) = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 = \sum_{i=1}^{n} (x_i - \hat{x}_i)^2 \tag{13}$$

The loss function quantifies how well the autoencoder is performing. By minimizing this loss, the neural network adjusts its weights and biases to improve the accuracy of the reconstruction.

3.4 Optimization

To minimize the loss function, gradient descent or its variants are used. The gradients of the loss with respect to the weights and biases are computed using backpropagation.

3.5 Gradient Computation

The gradients are computed as follows:

3.5.1 Decoder Weights and Biases

$$\frac{\partial L}{\partial \mathbf{W}_d} = \frac{\partial L}{\partial \hat{\mathbf{x}}} \cdot \frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{W}_d} \tag{14}$$

$$\frac{\partial L}{\partial \mathbf{b}_d} = \frac{\partial L}{\partial \hat{\mathbf{x}}} \cdot \frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{b}_d} \tag{15}$$

3.5.2 Encoder Weights and Biases

$$\frac{\partial L}{\partial \mathbf{W}_e} = \frac{\partial L}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}_e} \tag{16}$$

$$\frac{\partial L}{\partial \mathbf{b}_e} = \frac{\partial L}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{b}_e} \tag{17}$$

Backpropagation is the process of adjusting the weights and biases in the network to minimize the loss. By computing the gradient of the loss function with respect to each weight and bias, the network learns how to improve its performance.

3.6 Forward Pass

The forward pass through the autoencoder is described by the following steps:

3.7 Encoding

$$\mathbf{z} = \sigma(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e) \tag{18}$$

3.8 Decoding

$$\hat{\mathbf{x}} = \sigma(\mathbf{W}_d\mathbf{z} + \mathbf{b}_d) \tag{19}$$

The forward pass is the process of computing the output of the network given an input. During this pass, the input is encoded into a compressed form and then decoded back into the reconstructed input.

3.9 Matrix Representation

The operations in an autoencoder can be represented as matrix-vector multiplications. For a single data point, the encoding and decoding processes involve:

3.10 Encoding

$$\mathbf{z} = \sigma(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e) \tag{20}$$

3.11 Decoding

$$\hat{\mathbf{x}} = \sigma(\mathbf{W}_d\mathbf{z} + \mathbf{b}_d) \tag{21}$$

Matrix-vector multiplication is a fundamental operation in neural networks. In the context of autoencoders, it is used to transform the input data into a latent representation and then back into the original data space.

4 Experimental Study

4.1 Data Selection and Composition

Kaggle's amazon book dataset was chosen as it was sufficiently large and easy to work with. The dataset consist of two csv files: `book_details.csv`, `reviews.csv`.

The data had 3 million unique reviews, all consisting of integer item-rating values in the range $1 \leq n \leq 5$. Prior to data cleaning, there was a large amount of meta-data in the book data that appeared to be outside the scope of this experiment to utilize, such as the full user-written reviews, "helpfulness" scores that anonymous users could give to a chosen review, and time stamps for when the reviews were published. Further, the meta data in the book data included "categories" that we initially considered using to cluster the books into using a kNN or k-mean algorithm, but considering the size of the data, we elected to remove the columns associated with the aforementioned meta data.

4.2 Data Preparation

4.2.1 Storage

Dataset was stored in a sql database, as the raw data is very large and unwieldy for collaboration, while using raw sql query to parse the dataset proved to be faster as well.

4.2.2 Data Cleaning

We observed that if a user did not have a unique ID tag in the data set, they were presumably given a shared ID tag of no value, "NaN," thus approximately 500,000 reviews shared the "NaN" user ID tag. Thus if two users with the "NaN" ID gave two unique reviews for the same book, the model would have difficulty being able to tell people apart. After accounting for and removing all "NaN" user IDs, our data resulted in 2,438,213 unique reviews across 1,008,972 unique users and 216,023 unique books with at minimum one associated rating.

Many initial attempts at working with the entire dataset resulted in computational difficulty due to extreme sparsity caused by a large number of users leaving very few reviews. Amplified across over a million users, this proved problematic. Since we did not have access to hardware capable of performing this degree of computation, the solution we decided on was to improve computational efficiency by taking the submatrix of all users who had given ≥ 20 reviews. The resulting user-item submatrix had dimension 6,842 (users) by 92,327 (reviewed books). While still quite large in its own right, this alone indicates that a significant percentage of the users in the data set had given relatively few ratings on average. For the purposes of this experiment, we felt that the size of this submatrix was sufficient.

From sql table for book reviews, each user and book ids were assigned a unique tag. They were matched and cleaned into a `ratings_user_tagged` sql table where its columns are `User_id`, `review/score`, `book_id`.

The review were then imported into a pandas dataframe that filtered out books with less than 20 reviews, and stored as a csv zip file.

User_id	review/score	book_id
AVCGYZL8FQQTD	4.0	1882931173
A30TK6U7DNS82R	5.0	0826414346
A3UH4UZ4RSVO82	5.0	0826414346
A2MVUWT453QH61	4.0	0826414346
A22X4XUPKF66MR	4.0	0826414346
...	...	...
AI1QNMVF2E3TN	5.0	B000NSLVCU
AOFGOUMXLMVZS	4.0	B000NSLVCU
A1SMUB9ASL5L9Y	4.0	B000NSLVCU
A2AQMEKZKK5EE4	4.0	B000NSLVCU
A18SQGYBKS852K	5.0	B000NSLVCU

The data that linked the two data sets was the book titles, so we created a dictionary of the form {'book ID': ['Book Title', # overall ratings]} to log all of the unique book titles and count the number of reviews given by all users to each book. This was used to more easily store the column names of the dataframe and call the book titles when needed, like during top-k recommendations (covered later).

5 SVD Evaulation Methodology

5.1 Cross Validation

To perform the SVD factorization on R, we chose a value of $k = 100$ for the k latent features corresponding to the k highest singular values in Σ. We specifically used the Surprise sklearn library, since it has a variety of packages designed to work with prediction models. For consistency, the metric chosen to compute accuracy was RMSE. To test the RMSE across the entire dataset, we first ran cross validation across 5 folds over 200 epochs and found the mean of the RMSE,

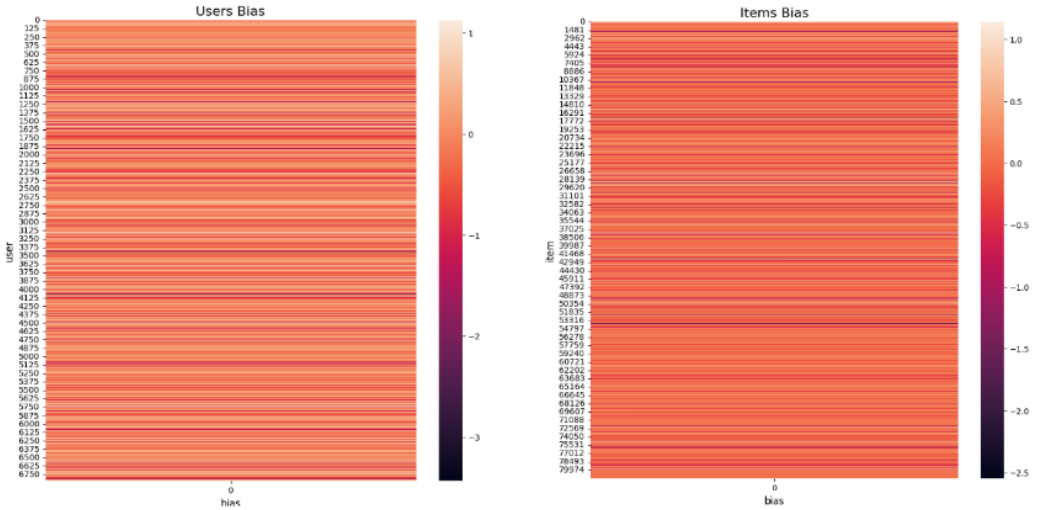
Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Mean RMSE	Std Dev
0.7729	0.7772	0.7729	0.7751	0.7732	0.7743	0.0017

with a mean fit time of 27.80 seconds and a mean test time of 0.70 seconds across all 5 folds.

5.2 Training and Test Set Splits

To check the RMSE in another method to compare against cross-validation, we then fit the dataset with an 80/20 train-test ratio, and calculated the RMSE of the test set across 200 epochs (stochastic gradient descent steps). Biases were included to account for overfitting and initialized at zero; the learning rate was set $\gamma = 0.005$, and regulation rate was set to $\lambda = 0.005$. At first, attempts at training a model with the number of epochs and number of features < 100 resulted in relatively high and unsatisfactory RMSE (> 1) but increasing both the number of features and epochs by a relatively small amount (100 features, 200 epochs) improved the model, with a result of RMSE $=$ 0.7692, which is an exceptionally accurate result. It was also within an acceptable delta of the cross validation. For context, an RMSE of .85 was considered prize-winning in the Netflix competition for many years, and even top submissions hovered around .95. Since our data set is much smaller than the Netflix competition, however, our relatively low (and thus good) RMSE is not particularly surprising.

Further, when the model is trained without biases, the RMSE $=$ 1.3275 (with 100 features and across 200 epochs), which indicates that the model is significantly improved by the addition of the user and item biases. Both user and item biases are shown in the figure below to give a sense of their distribution and how they relate to the prediction equation.

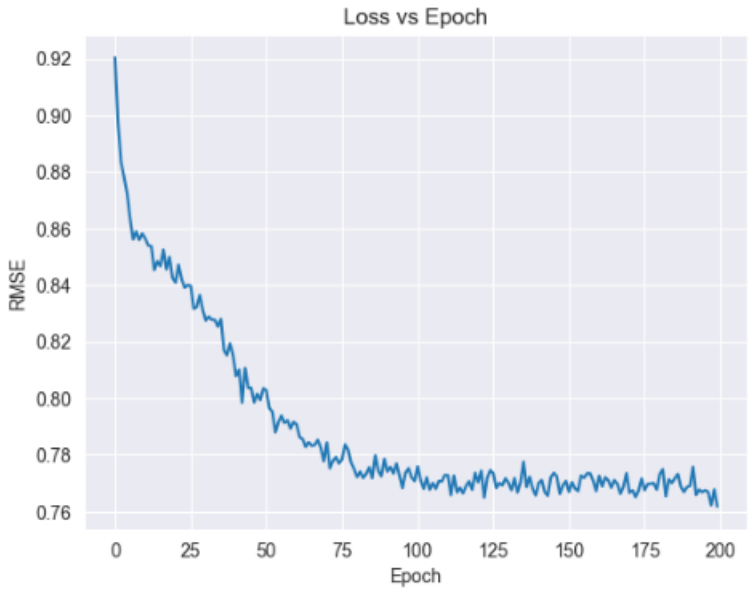


5.3 SVD++

In the spirit of experimentation, another SVD method, known as SVD++, was tested at 100 features across 200 epochs and returned a RMSE of 0.7695, which interestingly enough was less accurate by .0002, but much more computationally expensive to perform. As a result, we chose to continue using just SVD for the remainder of the experiment to simplify the process.

5.4 Loss Function

A graph of the RMSE Loss function for SVD changing over SGD epochs is shown below in Fig.(1).



The calculated RMSE appeared to converge around the value of 0.77, and the model did not appear to significantly improve either by increasing features or increase epochs to run SGD on, so we truncated the epochs to 100 to save computational cost.

5.5 SVD Evaluation Results

Recall that the rule for a prediction to a rating r_{ui} is the equation

$$\hat{r}_{ui} = \mu + b_i + b_u + \mathbf{q}_i^T \mathbf{p}_u. \tag{2.6}$$

Once the biases were minimized by SGD we ran predictions to check for both the 10 most accurate and 10 least accurate predictions in the testing set, as shown in table 1 and 2, respectively. The predictions were made against known ratings removed for the model to test against in the test data subset.

Observe that the model was able to exactly predict every r_{ui} for table 1, and every prediction $\hat{r_{ui}}$ was identical in that they predicted a rating of 1 when the known rating was 5. It is worth mentioning that as we adjusted the learning rate and regulation rate, the "worst 10" predictions wound up being wildly off as the model overall improved accuracy, which is somewhat interesting. Since these subsets don't necessarily tell us anything remarkable about the model's ability to make predictions, 10 additional predictions were selected at random to observe behavior of the model,

uid	iid	r_{ui}	$\hat{r_{ui}}$
A37Z5KLYEKK1S6	B00071T3YQ	5.0	5.0
A1EKTLUL24HDG8	B000FOENMW	5.0	5.0
A14OJS0VWMOSWO	1930051999	5.0	5.0
A1QN4CO0P3TI18	B00005WNTY	5.0	5.0
A295A2TPG8JJ7Y	B000GVE24S	5.0	5.0
A2NJO6YE954DBH	B000NPKRG4	5.0	5.0
A23D8ZZ60GTS3G	B0006AQJKE	5.0	5.0
A2W6DTJ46BK15Y	B0007JGCOC	5.0	5.0
A3PMW8WLB2K5RD	B00005X479	5.0	5.0
A14OJS0VWMOSWO	1410728757	5.0	5.0

uid	iid	r_{ui}	$\hat{r_{ui}}$
A7Y6AVS576M03	B000720VZU	1.0	5.0
A2BBZ3NI8MWH93	0806514760	1.0	5.0
A2VN986E5KR1NV	B000AMXBPO	1.0	5.0
A2BM5NTLX7CES1	1556433204	1.0	5.0
A1PXH6NS60C8V5	B000GVI7JY	1.0	5.0
A19N28I3SLUH97	0553295632	1.0	5.0
A36CI9LLL400DU	079104520X	1.0	5.0
AT04V4H3AO3T5	0965296601	1.0	5.0
A2X4G2SLD87MFG	1561385344	1.0	5.0
A4SY5RCGC5CH9	0850525934	1.0	5.0

which are shown in table 3. Predictions are rounded to 4 decimal places. As can be seen from the table, the model is able to predict most of the ratings within a relatively reasonable margin of error (± 1.234). Random predictions were generated for all 6,842 users.

uid	iid	r_{ui}	$\hat{r_{ui}}$
A1U0HC6Z37DUZ0	B000PGIL1O	5.0	3.766
A2RG6V2HH02U0M	051602275X	5.0	4.7703
A1A7VS5J7OR71D	047121888X	3.0	4.1221
A208XOXQ8KL31Y	0743219678	5.0	4.2822
AVV7VCHCFY3BI	B00007FYHS	4.0	4.0975
ANTGGAQYPSWHF	0201577933	5.0	4.0263
A2EDZH51XHFA9B	0975548239	4.0	3.3839
A2FG29CGPQV29N	1597222860	4.0	4.5728
ADDB0Y73L2CHU	0140860096	5.0	5.0
A1Y8RRW1GERC74	0312284675	5.0	4.3446

5.6 top-K recommendations

A very useful feature of collaborative-filtering methods is the ability to produce personalized top-K recommendations for a user. This is often performed by utilized cosine similarity, or, the angular cosine distance between two vectors. Since users and items are stored as vectors, measuring the distance between users and items allows us to determine relationships between vectors and generate potential recommendations for items the user has *not* yet rated.

The cosine similarity equation and top-K recommendation algorithm are the following:

$$\cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \cdot ||\mathbf{v}||} = \frac{(\sum_{i=1}^{n} \mathbf{u}_i \mathbf{v}_i)}{\sqrt{\sum_{i=1}^{n} (\mathbf{u}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{v}_i)^2}} \tag{22}$$

Algorithm 2 Top-k Recommendation using Cosine Similarity for SVD

Require: User matrix $P \in \mathbb{R}^{m \times k}$, item matrix $Q \in \mathbb{R}^{n \times k}$, number of recommendations k, user u

Initialize an empty list *recommendations*
Initialize an empty priority queue *pq*
for each item i not rated by user u **do**
    Compute the cosine similarity $sim_{ui} = \frac{p_u \cdot q_i}{\|p_u\| \|q_i\|}$
    Insert (i, sim_{ui}) into *pq*
    if $pq.size() > k$ **then**
        Remove the element with the lowest similarity from *pq*
    end if
end for
while *pq* is not empty **do**
    Extract the item i with the highest similarity from *pq*
    Append i to *recommendations*
end while
return *recommendations*

A list of top-k recommendations was generated for every user in the data set, and for $k = 5$. The first 5 users and their recommendations from the model's testing set were selected for the included table. The following 10 randomly selected predictions from the testing set was chosen:

user	iid_1	iid_2	iid_3	iid_4	iid_5
A1JH5J1KQAUBMP	B0006AQG7U	B000NR8BAG	B000IOHQKA	0613706633	0808510258
A2FEW4CBGB34YE	B000F159IC	0192503561	B000NKJM74	1901768600	9626341963
A2EDZH51XHFA9B	0553289713	B000PKC0QM	B000J521DU	B000MK50EE	0736645586
A2OJW07GQRNJUT	1931741654	B00071O7VK	B0006E1ZWC	B000PSAEFI	B0007HI9U4
AQE41QO3NEUMW	1569470219	B000N77VO8	015603252X	1558614893	B0007EQKTO

The first User, for example, would then have their top-5 recommendations decoded from the book ID's in the first row,

Predictions for User: A1JH5J1KQAUBMP	# Reviews
The time machine: An invention	639
The Adventures of Huckleberry Finn	762
Catch 22 (catch-22)	1044
Middlemarch (Turtleback School and Library Binding Edition) (Penguin Classics)	210
Harper Lee's To Kill a Mockingbird (Barron's Book Notes)	2396

In some cases, we noticed in the data that there were multiple "versions" of the same books being reviewed, but they would have distinct and unique book IDs. Further, the cosine similarity algorithm would sometimes recommend these multiple versions of the same book, which implies, at least to us, that the credibility to the predictions seems relatively sound.

5.7 SVD Evaluation Conclusions

The SVD factorization method is powerful, and appears to have a relatively high degree of accuracy with predicting known ratings and generating meaningful top-k recommendation lists. There are other known linear methods that could be applied in this situation, such as Nonnegative Matrix Factorization (NNMF), k-Nearest Neighbors Algorith, (kNN), and the aforementioned SVD++, although for the purposes of this experiment we chose to limit results to primarily just SVD.

6 Autoencoder Evaluation Methodology

For the evaluation of the autoencoder, Pytorch was used to implement the architecture for its similarity to NumPy and compatibility with GPUs. Training and running the model on the GPU would require considerable computation costs and time. Hence, Generating a randomly generated 10000 by 500 sparse user-item matrix to test a suitable architecture for the computation was the natural choice as it allowed small-scale testing on the CPU instead of the GPU. The resulting matrix was generated with 80% sparseness. Then the matrix dataframe was converted to a sparse matrix implementation `coo_matrix` in Scipy to save memory and storage. Autoencoder was then implemented with a symmetrical structure of 4-layer encoder MLP, 1 bottleneck layer, and 4-layer decoders. The encoder comprises of 2048, 1024, 512, 256 perceptron layers and 16 perceptron layer for the bottleneck while the decoder implements the encoder layer in reverse.

Bottleneck Considerations
Training Method was implemented as the pseudocode below:

Algorithm 3 Training Method for training the AutoEncoder model

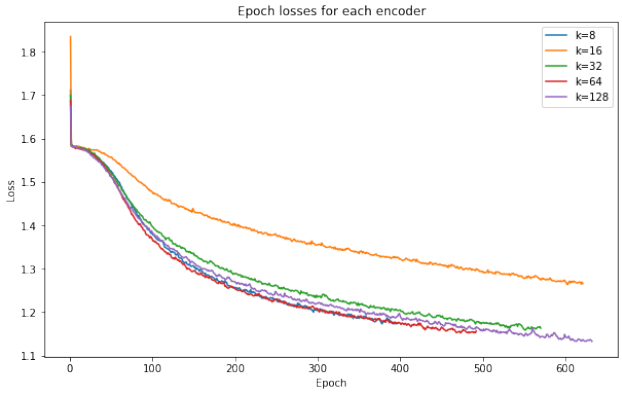
Require: *batches*: Training data batches
Require: *n_epochs*: Number of epochs (default=100)
Require: *min_delta*: Minimum change in loss to qualify as improvement (default=0.0001)
Require: *lr*: Learning rate (default=0.001)
Require: *patience*: Number of epochs with no improvement after which training will be stopped (default=10)
Ensure: θ: Trained model parameters

```
Initialize learning rate lr
Initialize Mean Squared Error (MSE) loss
Set best_loss to infinity
Set patience_counter to 0
Initialize epoch_losses as an empty list
Initialize epochs as an empty list
for epoch = 1 to n_epochs do
    Set epoch_loss to 0
    for each batch in batches do
        Perform a forward pass to get the output
        Perform a backward pass to compute gradients
        Update the model parameters using the ADAM optimizer
        Accumulate the loss for the current epoch
    end for
    Compute the average loss for the epoch as epoch_loss
    if epoch_loss < best_loss − min_delta then
        Update best_loss to the current epoch_loss
        Reset patience_counter to 0
    else
        Increment patience_counter by 1
    end if
    if patience_counter ≥ patience then
        Print early stopping message with current epoch and loss
        Break out of the loop
    end if
end for
```

With early stopping and Pytorch's ADAM optimizer(Note: an iteration of the SGD) to save computational cost.(Refer to Adam: A Method for Stochastic Optimization[5] for details)

Bottleneck Considerations

However, the autoencoder's bottleneck size and the corresponding latent space size could factor into the loss since the autoencoder's latent space dimension would also determine the information entropy, which implied the need to consider other bottlenecks. the bottleneck sizes of 8, 16, 32, 64 and 128 were chosen for comparison by epochs and their losses with the parameter being 1000 epochs, learning rate of 0.001. For each bottleneck, an autoencoder was trained and the epoch losses were recorded and then visualized as a plot on the CPU.



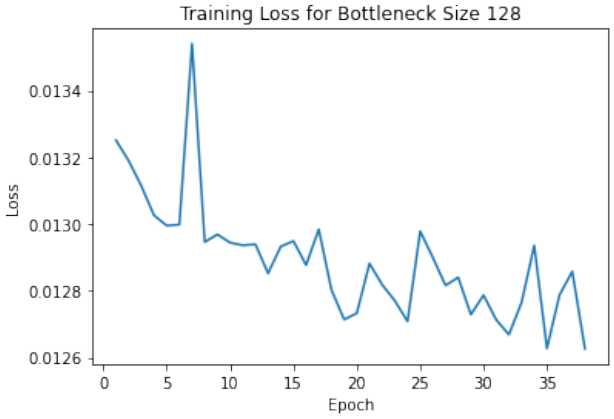
The visualization determined that bottleneck size of 128 gave the motimal optimal final epoch loss, therefore size 128 was chosen.

6.0.1 Training

With the bottleneck size and the architecture defined, the dataset was imported and trained with the same parameters except setting the `n_epochs=1000`. The training was done using *Lambda Labs*^{TM} Nvidia A[10] -enabled VM with 30 cpus.[6]

6.0.2 Results

Training Loss



The training took 48 epochs with a loss of 0.01267.

Accuracy To test the accuracy, `user_ratings` df was masked with 5% masking and ran the inference model on it to create predictions. Then randomly picked 10 rows to run the inference.

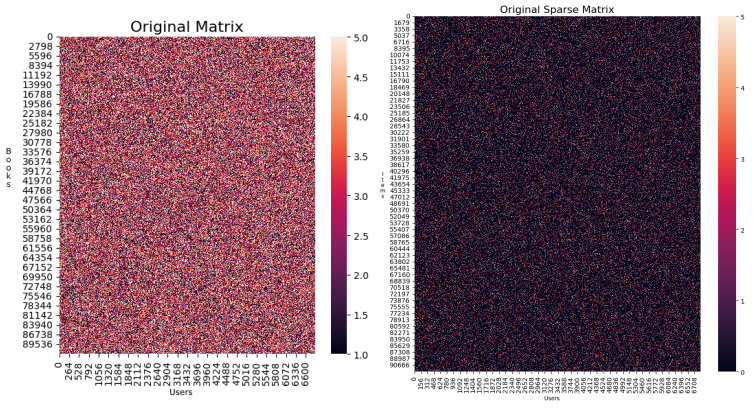
Row	User_id	review/score	book_id
1	A30TK6U7DNS82R	5.0	0826414346
3	A2MVUWT453QH61	4.0	0826414346
5	A2F6NONFUDB6UK	4.0	0826414346
6	A14OJS0VWMOSWO	5.0	0826414346
11	A373VVEU6Z9M0N	5.0	0829814000
34950	A2PK3NTC9RMEF4	3.0	0786182431
34958	A32ZKBXJJ45BRY	3.0	B00085PL4C
34967	A25JH6CO4DVINS	4.0	0255364520
34969	AOFGOUMXLMVZS	4.0	B000NSLVCU
34970	A1SMUB9ASL5L9Y	4.0	B000NSLVCU

uid	*iid*	r_{ui}	$\hat{r_{ui}}$	$\hat{r_{autoencoder}}$
12008	AIEEK7AHXKZCC	5.0	4.6737	0.0007
12860	AVY0UMR56U0CH	3.0	4.3636	0.0019
43792	A2NQMDEXQFZ2WB	5.0	4.8745	0.1700
22848	A2WLZD9BY669HY	2.0	3.8604	-0.0029
10345	A3K77GJCI8FRYU	5.0	4.1051	0.0086
47202	A21ZEHK7K4KOS8	3.0	3.4228	-0.0018
51503	A37MT7AGRZRLH9	5.0	4.5377	0.0014
41481	AKB2PYODH0TKS	4.0	3.7985	-0.0010
19990	A1F0O78HKGBRKO	4.0	4.1852	0.0445
52648	A2GBJQ9THOYDAJ	5.0	3.8359	0.0111

The resulting rmse for the `svd_predicted_rating` and `autoencoder_predicted_rating` are 0.898 and 4.209 while one-to-one comparison shows a large descrepancy against the original rating. This revealed a major problem with autoencoders, where they are prone to overfitting despite the initial analysis of the low rmse.
This lead to the hypothesis that the sparseness of the matrix affects the training and loss functions.

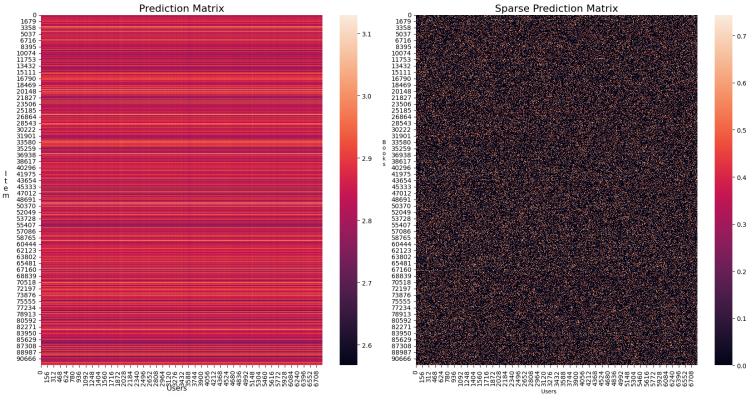
6.0.3 Analysis

To analyze this reason, a simulated matrix with the same dimensions was generated and had undergone through the previous training step on the A10 VM and been loaded into the model. The simulated matrix was created with it's sparseness at 0.8 to simulate the former matrix. Then the simulated matrices were converted into a `coo_matrices` and respectively had 0.05 masked counterparts created for it. To easily compare the results, the prediction matrices were converted to pandas dataframes, created heatmap visualization, and calculated their corresponding RMSE against the original matrices. The original matrices for the sparse and non-sparse datasets are



(23)

While the predictions for them are:



(24)

Using the RMSE to compare, with 2.0255 for the non-sparse matrix and 1.5695 for the sparse matrix, the sparse matrix is better. However, the visualization's numbers show that the sparse matrix's zeros significantly contribute to the smaller RMSE and matrix element numbers centered around 0.7 to 0. Also, the non-sparse matrix's prediction matrix's elements converge to numbers near 2.83 as indicated by large stripes, suggesting that high input dimensionality also contributes to the overfitting as well as the input and layer dimensions for the architecture would have to be significantly larger for proper encoding and decoding. Hence, the experiment showed that the autoencoder is highly prone to overfitting with sparse, high-input-dimension matrices.

6.1 Conclusion

The resulting empirical test reveals that autoencoders do not perform well with highly sparse data matrices such as the dataset used, as they are heavily skewered towards zeros and has high input

dimension. On the other hand, SVD factorization methods appeared to work very successfully for the given sparse matrix problem. It is very likely that ANN would be able to provide even more accurate results than SVD factorization would, but given the constraints of the experiment, we were unable to utilize them to their fullest potential.

7 References

[1] F. Ricci, L. Rokach, B. Shapira, P. B. Kantor, *Recommender Systems Handbook*, Springer, 2011.

[2] R. H. Singh, S. Maurya, T. Tripathi, T. Narula, G. Srivastav, *Movie Recommendation System using Cosine Similarity and KNN*, International Journal of Engineering and Advanced Technology (IJEAT), June 2020.

[3] N.Hug, Using prediction algorithms, https://surprise.readthedocs.io/en/stable/prediction_algorithms.html (updated 2015).

[4] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016.

[5] D. P. Kingma, J. Ba,*Adam: A Method for Stochastic Optimization*, https://arxiv.org/abs/1412.6980.

[6] https://www.nvidia.com/en-us/data-center/products/a10-gpu/