Movie Recommendation System Based on Deep Learning Model

Xiaoying Yang

Jing Jin

Shuang Ma

Computer Science and Engineering

Santa Clara University

Abstract

In this research, we implement the movie recommendation system based on autoencoder neural network which is an unsupervised learning algorithm that applies backpropagation, and compare the prediction result with the model based on Matrix Factorization and that based on item-based K nearest neighbor method.

We used the MovieLens latest small dataset to test the performance of our model, which significantly outperforms the item based KNN method. Performance is measured as accuracy and mean squared error these two metrics. Specifically, the average prediction accuracy of KNN is only about 24.71% and the mean squared error is about 8.66, which are of poor quality. The accuracy of matrix factorization model is about 42.1%. The validation accuracy of autoencoder model is about 58% and the test accuracy is about 77%.

All these results show that the performance of our Autoencoder model outperforms the Matrix Factorization model, as well as the item based KNN method.

Keywords: Autoencoder, movie recommendation system, Matrix Factorization

1. Acknowledgement

We would firstly like to thank Professor Wang for this opportunity to get into a field that we are interested in. Second, we would like to thank Professor Wang for his advice and encouragement throughout the project.

2. Introduction

2.1 Objective

The objective of this project is to implement a movie recommendation system based on deep learning model. This system can predict the movie score, and recommend the movie with highest score.

2.2 What is the problem

The problem is to predict each movie score that the user hasn't watched yet, and recommend a movie which the user mostly prefers.

2.3 Why this is a project related to the class

We think the project is related to the class because the technique of recommendation system, for example, co-occurrence matrix, contend-based approach etc, are the implementation of data mining. All that we need to do is to training input datas to extract features, build models and find relationship between users and movie items.

2.4 Why other approach is no good

Generally, there are two common approaches for movie recommendation system, one is called collaborative filtering (CF) and another is called Content-based recommendation. However, neither of them can handle the cold start problem well. CF needs a considerable amount of previous history records to make high quality recommendation. Thus If the user-item matrix is sparse, it can hardly make accurate prediction. Theoretically, content-based approach can deal with the cold start problem, but it always fail to build an accurate and complete model due to the complicate and rich features of movie items.

2.5 Why you think your approach is better

In our project, we use the autoencoder to automatically train the model. It will keep learning the latent factors of both users and items. Through this way, our model is always updated and improved, and we can build a rich and accurate model through this training part.

2.6 Statement of the problem

Our project will design a movie recommendation system for users according to user's taste and preference. In our project, we will extract a rich features from existing data, build the user's interests, score each movie based on user's interests, and recommend the movie with the highest score.

2.7 Area or scope of investigation

The area of our project includes data mining, machine learning and recommendation system.

3 Theoretical Bases and Literature Review

3.1 Classification of Recommender Systems

Generally, there are three types of methods for recommender system: collaborative filtering methods, content-based methods and hybrid methods.

1) Collaborative Filtering (CF)

Collaborative filtering is a popular technique used in recommendation systems, which is based on the assumption that people tend to share similar views with those who have similar taste or preference. There are two basic types of CF. One is memory-based algorithms. Memory-based CF algorithms are widely used in early recommendation systems. In these algorithms, also explainable. However, the rating data is usually very sparse, making the prediction result less accurate and reliable.

And the other one is model-based algorithms. Model-based CF uses data mining and machine learning technique to find implicit patterns on training data and uses the result to make recommendations are made by using users' rating data to compute the similarities of users or items, such as user-based/item-based recommendations. This method is easy to implement and recommendations. One of the widely used

models is Matrix Factorization (MF). There are various MF models such as SVD, PCA, PMF. The intuition behind these methods are to find the hidden features that can be the underlying reasons for users' rating. This approach allows to take additional information into consideration so that it could do better with data sparsity [1]. But it can be difficult to explain the features and the prediction, and it is more expensive than naive CF methods to build the models.

2) Content-based Methods

In content-based methods, recommendations are made based on the descriptions of the items and the profiles of the users' interests [2]. Instead of using user item correlation, content-based RS generates item-item correlations to make recommendations. This approach can utilize different domains of information. Thus, even if the user has not provided ratings, the system may find latent factors from other sources and then make recommendations.

3) Hybrid Methods

A hybrid method combines two or more methods in order to overcome the drawbacks of each method.

3.2 Recommender System Based on Deep Learning

As an emerging and popular approach in data mining and machine learning, deep learning has been proposed to used in building recommender systems for improving the performance.

In order to reduce the effect of data sparsity in CF method, Gao et al.(2016) proposed a deep learning model using stacked Restricted boltzmann machines (RBM) to form Deep Belief Network (DBN) to predict the null values of null rating for reducing the sparsity of rating data [3].

Wang et al.(2014) used model based on deep belief network and probabilistic graphical mode to learn features from audio content. This model outperforms other deep learning models in both the warm-start and cold-start stages without relying on collaborative filtering [4]. Wang et al.(2015) developed a hierarchical Bayesian model (collaborative deep learning) which jointly performs deep representation learning for the content information and collaborative filtering for the ratings matrix. Experiment results show that CDL can extract effective deep feature representation from content and capture the similarity and implicit relationship between items (users) [5].

For content-based RS, Elkahky et al.(2015) used a multi-view deep learning model jointly learn from features of items from different domains and user features to address the recommendation quality and the system scalability [6].

3.3 Advantages and Disadvantages

Generally, CF methods rely on rating history to make recommendations so that these methods suffer from the problem of data sparsity. Thus, for cold start problem, when users have not provided rating values, or when new items rarely get ratings, these methods would not give accurate predictions for making recommendations. Previous research has shown that content information could used to reduce the effect of cold start problem.

Deep learning models used in recommender systems could automatically learning the latent factors for both users and items and previous research has shown that the quality of commendation could be improved by using deep learning models.

4. Approach and Hypothesis

4.1 Approach

Based on the research we discussed previously, we would build an autoencoder model to predict the user's ratings using history rating data. We will also implement in other methods, such as K-nearest neighbor and Matrix Factorization, and compare the performance of different methods.

The major difference between our approach and previous research is that we use an autoencoder model for representation and prediction. We will improve the measurement of cosine similarity by considering user's difference and also improve the matrix factorization method by using the Stochastic gradient descent algorithm.

4.2 Hypothesis

Based on the analysis in previous part, we assume that our approach based on autoencoder model outperforms the matrix factorization model, also outperforms other baseline methods.

5. Methodology

5.1 How to collect data

In this research, we experiment on MovieLens data, which were collected by the Grouplens Research Project at the University of Minnesota. This data set consists of 9K movies by 677 users; including tag genome data with 12million relevance scores across 1,100 tags. Another dataset is from Netflix Prize dataset. There are over 480,000 customers and over 17,000 movies in the dataset. This dataset contain over 100 million ratings.

5.2 How to solve the problem

We employ several methods to predict the unrated user-movie pairs. Specifically, we use AutoEncoder, matrix factorization, and K-nearest neighborhood, and compare their accuracies and results. Accuracy is defined as the number of ratings the algorithm correctly reconstruct, as followed

$$\sum_{r \in R} I(\hat{r} = r)$$

Where R is the set of entries with user ratings, r is the actual ratings, and $\hat{r}$ is their corresponding predictions. I(x) equals 1 if x is true. In other words, unrated entries are not considered in computing the accuracies.

5.2.1 Algorithm Design

1) Item-Based *K*- Nearest Neighborhood (KNN) Algorithm

In this algorithm, in order to determine the rating on User u on Movie m, we can find other movie that are similar to movie m, and based on User u's rating on those similar movies we infer his rating on Movie m. In order to determine which movie are similar, in this study we use cosine similarity to measure the distance between different movie a and movie b. Base on this rating matrix, we find the nearest K neighbors of each movie and then use this neighbors' preference to recommend movie.

KNN finds the nearest K neighbors of each movie under the above defined similarity function, and use the weighted means to predict the rating (the KNN algorithm for movies leads to the following formula)

$$P_{m,u} = \frac{\sum_{j \in N_j^K(m)} sim(m,j) R_{j,u}}{\sum_{j \in N_j^K(m)} |sim(m,j)|},$$

Where $N_u^K(m)$:={ j: j belongs to the K most similar movies to Movie m and User u has rated j}, and sim (m, j) is the adjusted cosine similarity defined in the above formula, $R_{j,u}$ are the existent rating (of User u on Movie j) and $P_{m,u}$ is the prediction.

2) Matrix factorization

Matrix factorization characterizes both items and users by vectors of factors inferred from item rating patterns. High correspondence between item and user factors leads to a recommendation. Specifically, our matrix factorization model will be based on the latent factor model [1]. Firstly, we have a set of U users and a set of I items. Let R be the matrix of size |U| *|I| which contains all the ratings that the users have assigned to the movie they have watched. The goal of this method is to find two matrices P(|U| * K) and Q (|I| * K) such that their product approximately equals to R is given by :

$$R \approx P \times Q^T = \hat{R}$$

In this way, the Matrix factorization modes map both users and items to a joint latent factor space of dimensionality f user-item interaction are modeled as inner products in that space[1]. Hence, each item i is associated with a vector $q_i \in \mathbb{R}^f$, and each user u is associated with a vector $p_u \in \mathbb{R}^f$. For a given item i, the elements of q_i measure the extent to which the item possesses those factors positive or negative. The result dot product $q_i^T p_u$ catch the interaction between user u and item i, overall interest in the item characteristics. This approximates user u's rating of item i which is denoted by r_{ui} leading to the estimate:

$$\hat{r}_{ui} = q_i^T p_u$$

To learn the factor vectors p_u and q_i, the system minimized the regularized squared error on the set of known ratings as [1]:

$$\min_{q^*,p^*} \sum_{(u,i)\in\kappa} (r_{ui} - q_i^T p_u)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

Here , K is the set of the (u, f) pairs for which $\hat{R}_{uf}$ is known the training set. The constant λ controls the extent of regularization and is usually determined by cross-validation.

For the implementation, starting from random initialization, we apply backpropagation to gradually adjust P and Q so that their multiplication mimics the original rating matrix R, for the rated entries. For example, given an original rating matrix as followed,

[[5 0 3 0]
[4 0 1 1]
[2 1 0 5]
[1 0 3 2]
[0 1 4 0]]

Our algorithm reconstructs the rated entries into the following prediction matrix,

[[4.96455154 0.75023132 2.98824417 2.3140688]
[3.96502049 0.26110945 1.00800281 0.99885005]
[2.00084149 0.99794352 3.57319845 4.95005934]
[1.00437283 0.73495906 2.96880892 2.00042949]
[1.9277972 0.98570364 3.9651213 2.77312679]]

The rated entries are very close to the original ratings, with predictions on un-rated entries.

Again, the backpropagation is only applied to rated entries, to avoid overlearn from the rating matrix. We might explore other variations of the implementation, to improve the performance on big data.

3) Autoencoder

An autoencoder neural network is an unsupervised learning algorithm that applies backpropagation, setting the output values to be equal to the inputs. y(i) = x(i). In this study, we try to design an user-based (item-based) autoencoder which can take the input as each partially observed r(u) (r(i)), and then project it into a low-dimensional latent (hidden) space, and then reconstruct r(u) (r(i)) to the output layer to predict the missing ratings for purpose of recommendation. Formally, given a set S of vectors in $\mathbb{R}^d$, and some $k \in \mathbb{N}_+$, an autoencoder solves

$$\min_g \sum_{\mathbf{r} \in S} ||\mathbf{r} - h(\mathbf{r}; \theta)||_2^2,$$

Where $h(\mathbf{r}; \theta)$ is the reconstruction of input $\mathbf{r} \in \mathbb{R}^d$,

$$h(\mathbf{r};\theta) = f\left(\mathbf{W} \cdot g(\mathbf{V}\mathbf{r} + \mu) + \mathbf{b}\right)$$

For activation function f(.).g(.). Here , $\theta = \{\mathbf{W}, \mathbf{V}, \boldsymbol{\mu}, b\}$ for transformations $\mathbf{W} \in \mathbb{R}^{d \times k}, \mathbf{V} \in \mathbb{R}^{k \times d}$ and biases $\mu \in \mathbb{R}^k$, $\mathbf{b} \in \mathbb{R}^d$. This objective corresponds to an auto-associative neural network with a single, k-dimensional hidden layer. The parameter θ are learned using backpropagation.

The item-based Autoencoder model applies an autoencoder as above equation to the set of vectors $\{\mathbf{r}^{(i)}\}_{i=1}^{n}$ with two changes: First, we account for the fact that each $\mathbf{r}^{(i)}$ is partially observed by only updating during backpropagation those weight that are associated with observed inputs, as is common in matrix factorisation and restricted Boltzmann machine approaches. Second, we regularise the learned parameters so as to prevent overfitting on the observed ratings. Formally, the objective function for the Item-based Autoencoder model is , for regularisation strength $\lambda > 0$,

$$\min_{\theta} \sum_{i=1}^{n} ||\mathbf{r}^{(i)} - h(\mathbf{r}^{(i)};\theta))||_{\mathcal{O}}^2 + \frac{\lambda}{2} \cdot (||\mathbf{W}||_F^2 + ||\mathbf{V}||_F^2)$$

Where $||\cdot||_{\mathcal{O}}^2$ means that we only consider the contribution of observed ratings. User-based Autoencoder is derived by working with $\{\mathbf{r}^{(u)}\}_{u=1}^{m}$. In total, this model requires the estimation of $2mk + m + k$ parameters. Given learned parameter $\hat{\theta}$. This model predicted rating for user u and item i is

$$\hat{R}_{ui} = (h(\mathbf{r}^{(i)};\hat{\theta}))_u.$$

To reconstruct the rating matrix as well as predicting unrated ones, autoencoder learns the latent features from the rating matrix, and output the reconstructed input to mimic the actual input as much as possible. To allow for extension to item specific features like [4], we build an item-based model. Our autoencoder is a 4 layer feed forward neural network. Input is the rating for each item (677 neurons). 2 hidden layer (300 and 150 neurons respectively) learn the latent features from the items, and the output layer is same size as input (677 neurons) to mimic the input.

The network is trained using stochastic gradient descent, although we might apply other backpropagation algorithms like Rmsprop, Ada, Adam, etc., if they are better at searching in the weight space.
Since the rating matrix is very sparse, with most entries as 0. To avoid the algorithm to overlean the rating matrix and predict most un-rated entries as 0 as well, we restrict the backpropagation to only those rated entries. In other words, unrated entries are not backpropagated to adjust the weights. In the following graph, the solid lines are those weights that are updated; while the dash edges are connected but not actually updated in that input, since the ratings are miss.

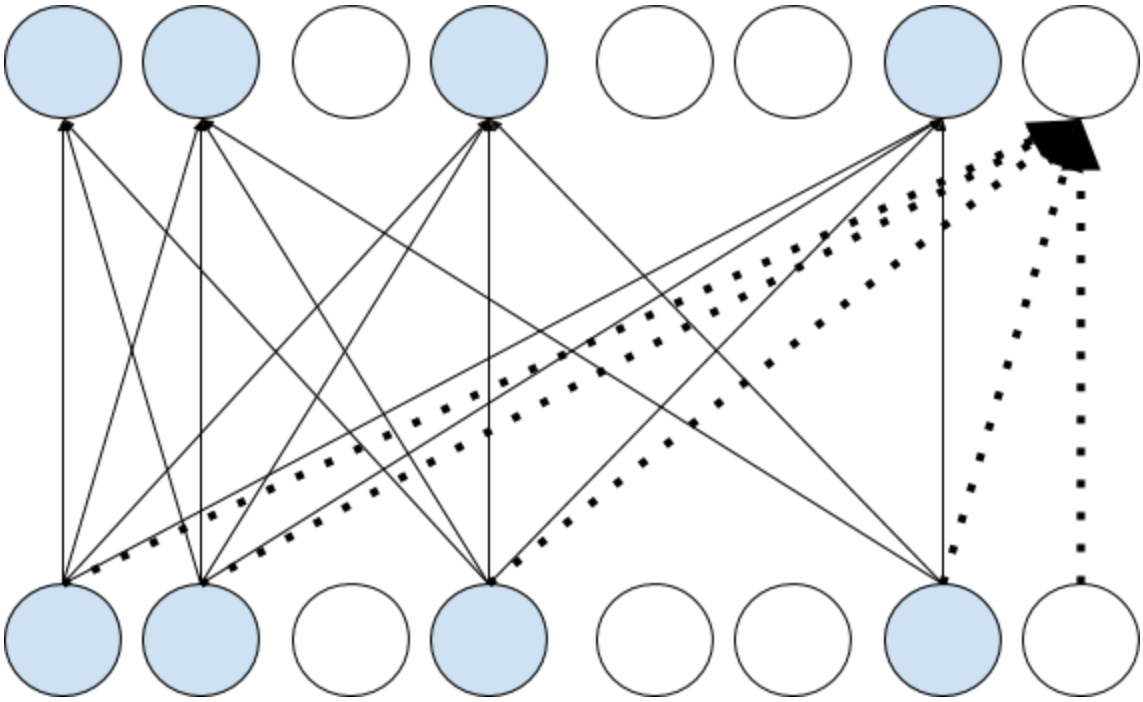

Figure 1: The backpropagation model of AutoEncoder.

There are several tunable knobs in this model.

- Number of hidden layers
- Size of hidden layer
- Learning rating
- Penalization strength

Depends on the specific backpropagation algorithm, there maybe other knobs. Given the current setting, our network need to learn 677 x 300 + 300 x 150 + 150 x 677 + (300 + 150 + 677) = 350777 parameters.

4) Stochastic gradient descent

When the dataset is too big to fit in main memory, the entire training set can be very slow and sometimes intractable on a single machine. Stochastic Gradient Descent(SGD) can handle this problem by following the negative gradient of the objective after seeing only a single or a few training examples. In this algorithm loops through all ratings in the training set. For each given training case, the system predicts r_{ui} and computes the associated prediction error

$$e_{ui} \overset{def}{=} r_{ui} - q_i^T p_u.$$

Then it modifies the parameters by a magnitude proportional to γ in the opposite direction of the gradient yielding:

$$qi \leftarrow qi + \gamma \cdot (eui \cdot pu - \lambda \cdot qi)$$
$$pu \leftarrow pu + \gamma \cdot (eui \cdot qi - \lambda \cdot pu)$$

This popular approach combines implementation ease with a relatively fast running time.[1,7]. Thus, we will apply this approach in matrix factorization and Autoencoder model.

5.2.2 language used

Python

5.3 how to generate output

Following[1], we use a default rating of 3 for test users or items without training observations. We split the data into random 90%-10% train-test sets, and hold out 10% of the training set for hyperparameter tuning. We repeat this splitting procedure 5 times and report average RMSE 95% confidence.

5.4 How to test against hypothesis

In this study, we evaluate and compare different methods in music recommendation, k- nearest neighbourhood, matrix factorization and Autoencoder on the Movielens dataset. Our task is to predict all

the users' ratings on all the movies based on the training set. In other words, we try to minimize the RMSE (root mean square error) when predicting the rating on a test set.

$$SSE = \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$$

$$MSE = \frac{1}{n} \times SSE$$

We compare the test output value with the original data and then calculate its MSE. Also, we round the output data and then calculate how many values are exactly the same with the original data.

6. Implementation

6.1 Code

6.1.1 AutoEncoder- Generality description

All the codes of this project are written by Python. The first model is AutoEncoder, in this work, we design an item (movie) based autoencoder. Each input is from one (batch of) item inputs, containing the all user's ratings of this movie. Our work can easily transform to user based by changing the input from each column of the rating matrix to each row of the rating matrix. Without loss of generality, we assume item based from now on.

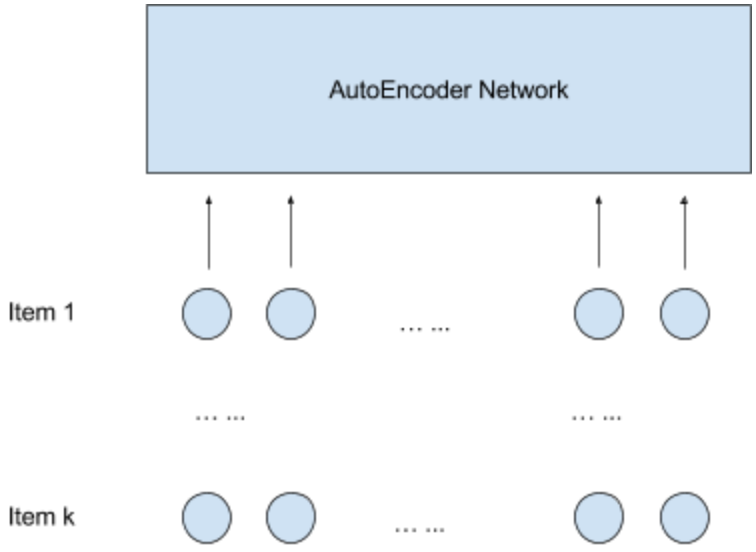

Figure 2: The way the input the data.

We explore several design options, including the number of layer, size of hidden layer and a set of parameters like learning rate, regularization rate, ets.. Our data set contains roughly 700 users and 9000 movies. Network more than 3 layer exhibit significant overfitting and suffers from very slow training. Our final autoencoder has only 1 hidden layer of 100 neurons.

6.1.2 Define a loss function

The loss function is a combination of sum of square errors (SSE) and a set of regularization to avoid overfitting. Specifically,

$$loss = \sum_{i=1}^{k} \left[\left(X_{ij} - O_{ij} \right)^2 * I(X_{ij} > 0) \right] + \beta * \sum \left(\alpha W_{ij}^2 + (1-\alpha) | W_{ij} | \right) + \sum_{i=1}^{k} (I(0 \geq O_{ij} || O_{ij} > 5) * O_{ij})^2$$

The first part is the normal SSE to measure the errors; note the only rated entries are computed here. This ensure the learning will reconstruct the rated entries as much as possible while not distorted by the predictions on unrated entries. The second part is a lasso regularizer, which combines penalty of both L2 and L1, balanced by the factor α.We explore the design of α here and settle to 0.9 since we value sparsity of the weight matrix over their small value. Experiment shows that we achieve better accuracy when setting bigger L1 regularization than L2. The third part is a penalization of wrong value. Since ratings are allowed between 0 and 5, we use L2 norm to penalize invalid rates.

```
def loss(self, X, Xn, reg = 0.001, sparsity = 0.1):
    W1, b1 = self.params['W1'], self.params['b1']
    #W2, b2 = self.params['W2'], self.params['b2']
    W3, b3 = self.params['W3'], self.params['b3']
    N = np.shape(X)[0]
    # forward
    Z1 = X.dot(W1) + b1
    H1 = sigmoid(Z1)                                    # N * H1
    Z3 = H1.dot(W3) + b3
    O = np.maximum(0, Z3)                       # N * I (== O)
    mask = (X + Xn > 0) * 1
    diff = (X + Xn - O) * mask
    # wrong value
    roundO = np.round(O * 2) / 2
    bigO = (roundO > 5) * 1
    smallO = (roundO <= 0) * 1
    wrongO = (bigO + smallO) * O * mask
     lossval = 0.5 * np.mean(np.square(diff)) + 0.5 * beta * (1 - l1l2) * (np.mean(W1 * W1) + np.mean(W3 *
W3)) / 2 + beta * l1l2* (np.mean(np.absolute(W1)) + np.mean(np.absolute(W3))) / 2 + 0.5 * gamma *
np.sum(wrongO * wrongO)

    return lossval, grads
```

6.1.3 Training

We use backpropagation algorithm to learn the weights and bias (*W* and *b* above), which are randomly initialized. We use Adam update rules which keeps track of both first and second moment of the gradients, and use that to compute the momentum to update the parameters. Specifically, given a gradient *g* of a parameters *p*, we keep track of the running average of its first and second moment,

$$m_p = \alpha * m_p + (1 - \alpha) * g$$
$$n_p = \beta * n_p + (1 - \beta) * g^2$$
$$p = p - learnigrate * m_p / \sqrt{n_p}$$

α is set to 0.9 and β is set to 0.99. Compared with normal stochastic gradient descent, this has the benefit of achieving "per-parameter" learning rates. While the learning rate is a global parameters, using the update rule above, parameters that are frequently updated or updated with big gradients will have their effective learning rate decrease rapidly; on the opposite, parameters that are infrequently updated or updated with small gradients will have their effective learning rates increase. This is a smarter way of update compared with the globally fixed learning rate setting in typical stochastic gradient descent.

We set the global learning rate to 0.01, lasso regularization rate to be 0.002 and the wrong value penalization rate to be 0.01. These rate also gradually decay after each 10 epoch with a decay rate 0.9 -- after each 10 epoch, these rate decreases by 10%.

```
def train_and_test(self, update = 'Adam', learning_rate = 0.01, reg_rate = 0.001, decay = 0.99):
  loss_hist = []
  train_acc = []
  m_W1 = np.zeros_like(self.params['W1'])
  m_W3 = np.zeros_like(self.params['W3'])
  m_b1 = np.zeros_like(self.params['b1'])
  m_b3 = np.zeros_like(self.params['b3'])
  n_W1 = np.zeros_like(self.params['W1']
  n_W3 = np.zeros_like(self.params['W3'])
  n_b1 = np.zeros_like(self.params['b1'])
  n_b3 = np.zeros_like(self.params['b3'])

  for i in range(NUM_ITER):
    X_batch, X_batch_noise = self.get_train_batch()
    lossval, grads = self.loss(X = X_batch, Xn = X_batch_noise, reg = reg_rate)
    if update == 'SGD':
      self.params = update_params_sgd(self.params, grads, learning_rate)
    else:
      self.params = update_params_adam(self.params, grads, learning_rate,
                m_W1, m_W3, m_b1, m_b3,
                n_W1, n_W3, n_b1, n_b3,
              alpha_W, beta_W, alpha_b, beta_b)
    X = X_batch + X_batch_noise
    O,acc = self.evaluate(X_batch + X_batch_noise, (X > 0) * 1)
    train_acc.append(acc)
```

```
    loss_hist.append(lossval)
    if i % EPOCH_SIZE == EPOCH_SIZE - 1:
      X_batch_valid = self.get_validate_batch()
      X = (self.train_ratings + self.noise_train_ratings + self.validate_ratings)[:,0:DATA_SIZE].T
          O,validate_acc = self.evaluate(X, (X > 0) * 1)

      if i % (DECAY_RATE * EPOCH_SIZE) == (DECAY_RATE * EPOCH_SIZE - 1):
        learning_rate *= decay
        reg_rate *= decay
      loss_hist = []
      train_acc = []
                X = (self.train_ratings + self.noise_train_ratings + self.validate_ratings +
self.test_ratings)[:,0:DATA_SIZE].T
   O,test_acc = self.evaluate(X, (X > 0) * 1)
```

6.1.4 Evaluate Accuracy

Accuracy is measured as the number of rated entries that are correctly constructed. This is different from SSE in that we require the reconstructed ratings to be exactly the same as the input. We round our predictions to the nearest 0.5 (for example, 3.4 and 3.65 all round to 3.5) and compare them with the inputs.

```
def evaluate(self, X, mask):
   H1 = sigmoid(X.dot(self.params['W1']) + self.params['b1'])
   #H2 = sigmoid(H1.dot(self.params['W2']) + self.params['b2'])
   #O = np.maximum(0, H2.dot(self.params['W3']) + self.params['b3'])
   O = np.maximum(0, H1.dot(self.params['W3']) + self.params['b3'])
   O = np.round(O * 2) / 2
   #acc = float(np.sum(((O * mask) - y) ** 2)) / np.sum(mask)
   acc = float(np.sum((X == O) * 1 * mask)) / np.sum(mask)
   return O, acc
```

6.2 Matrix Factorization

Similar with the AutoEncoder model, we use backpropagation to learning the latent user vector and latent movie vestor, first we choose random number as the vector value and then use stochastic learning algorithm to tune the model.

```
for i in range(NUM_ITER):
  pred = np.dot(P, Q)
  diff = (train - pred) * train_mask
  err = 0.5 * np.sum(np.square(diff)) + beta * 0.5 * (np.sum(np.square(P)) + np.sum(np.square(Q)))
  ddiff = -diff * train_mask
  P -= alpha * (ddiff.dot(Q.T) + beta * P)
```

```
Q -= alpha * (P.T.dot(ddiff) + beta * Q)
if i % 10 == 0:
  pred = np.round(pred * 2) / 2
  train_acc = float(np.sum((pred == train) * 1 * train_mask)) / num_train
     train_acc_hist.append(train_acc)
  validate_acc = float(np.sum((pred == validate) * 1 * validate_mask)) / num_validate
  validate_acc_hist.append(validate_acc)
```

6.3 Design document and flowchart

The basic flow chart of training model:split the dataset into three parts : training dataset, validation dataset, and test dataset; the proportion in our study is 90%, 10% and 10% separately. First, we try to build the model and then use the loss function to evaluate the training accuracy with the loss function we define above.Then according to the loss function result to do backpropagation, in this study we use stochastic gradient descent to do backpropagation and then adjust the model, after many iteration when the result become stable then we check the validation result and tune those hyperparameters such as learning rate and penalization rate manually, sometime you need to rebuild the model . In our case, we first built a four layer model and then we find that there is too much overfitting in our result, which means that the training accuracy is much higher than the test accuracy and validation accuracy. Then we try to reduce the penalization rate but the training accuracy is also reduced too. So we rebuild the model into 3 layers.. Finally, we get the validation result which fulfill the requirement and then use this trained model to test the testing dataset and compare the different models(Figure 3.).

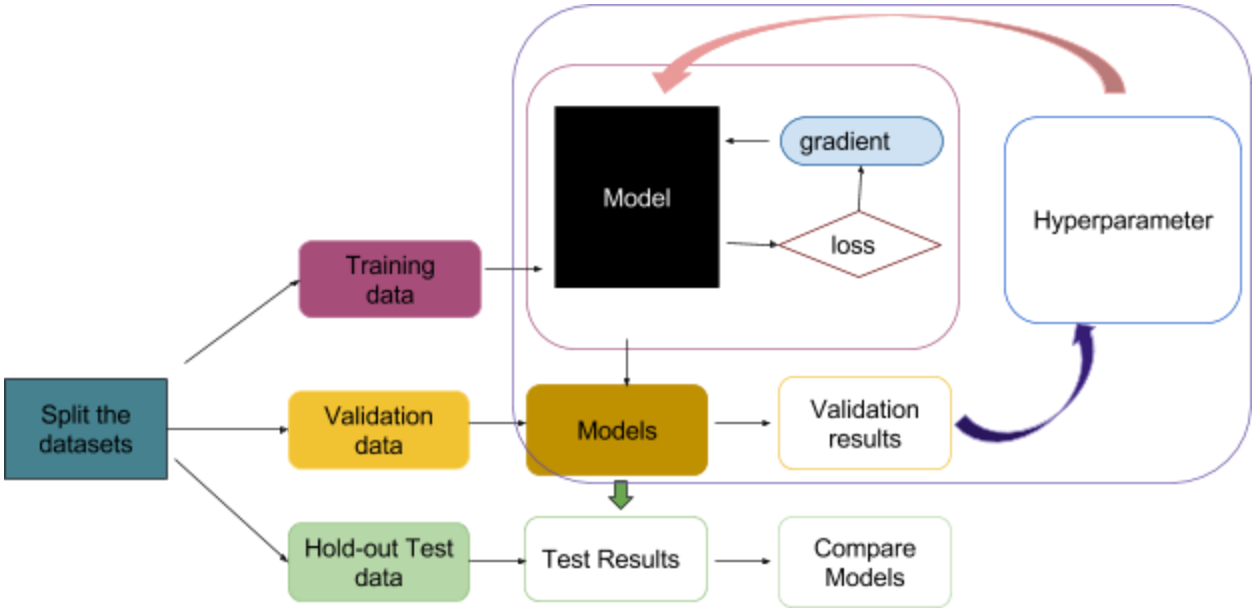

Figure 3: The flowchart of model training.

At this study, for autoencoder model we try deeper architecture which content 4 layers but the it is too much fitted which means that the training dataset accuracy is much higher than the validation dataset. To avoid this problem, we use three layers structure (Figure 4.)rather than four layers. Below is the structure of our model.

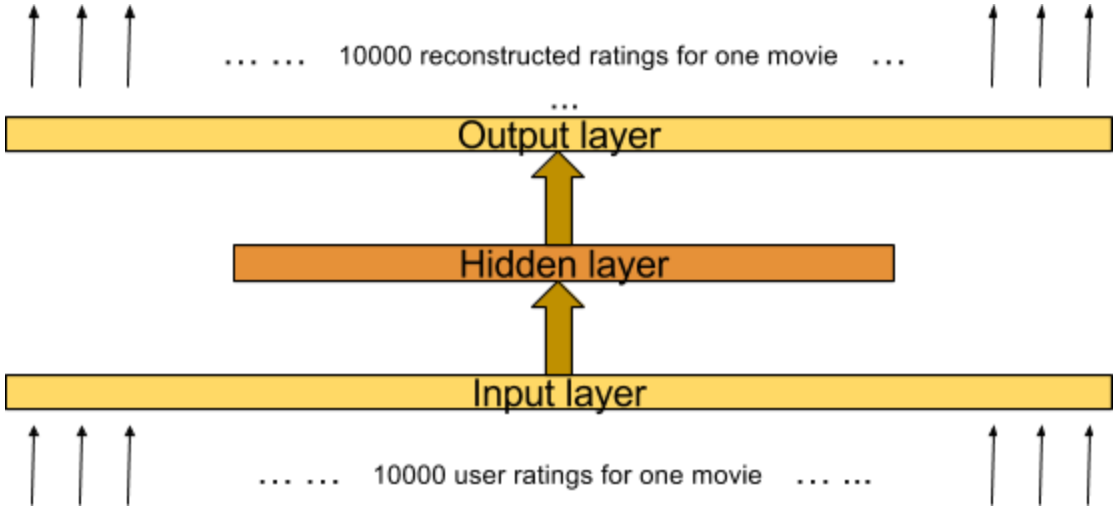

Figure 4: The architecture of the final autoencoder model.

7. Data analysis and discussion

7. 1 Output generation

For the model of AutoEncoder, we split the data into random 90%-10% train-test sets, and hold out 10% of the training set for hyperparameter tuning. We repeat this splitting procedure 5 times and report average MSE and accuracy.

For KNN and Matrix Factorization method, we use 90% of data for prediction and compare the prediction result to the left 10% of data, and calculate the accuracy and MSE for the prediction data which correspond to the 10% data get rid of at the first place.

7.2 Output analysis

Since we randomly split the whole dataset into training, validation and testing sets, we run the KNN and autoencoder model five times in order to try to avoid the impact of the random selection and split. However, the matrix factorization model is extremely time consuming so that we only run the model twice and the result does not show much difference.

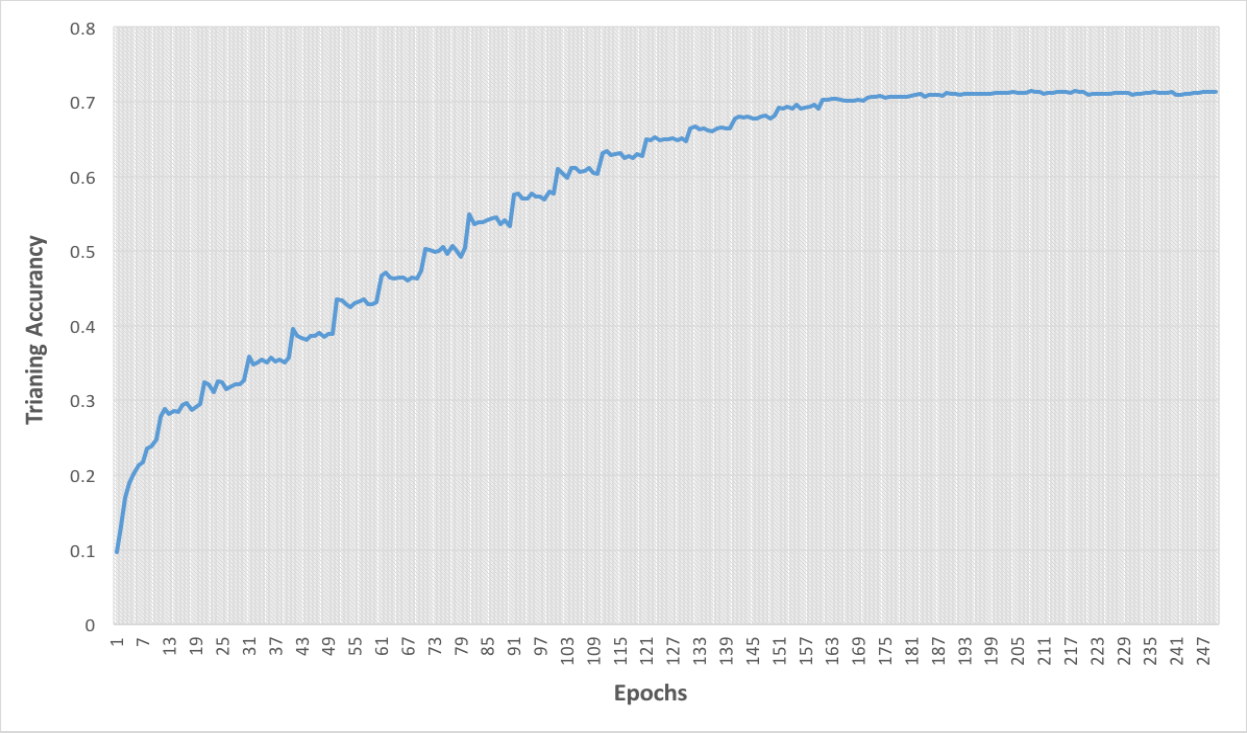

Figure 5: Validation Performance of AutoEncoder model. The y-axis displays the Training Accuracy; the x-axis displays Epochs.

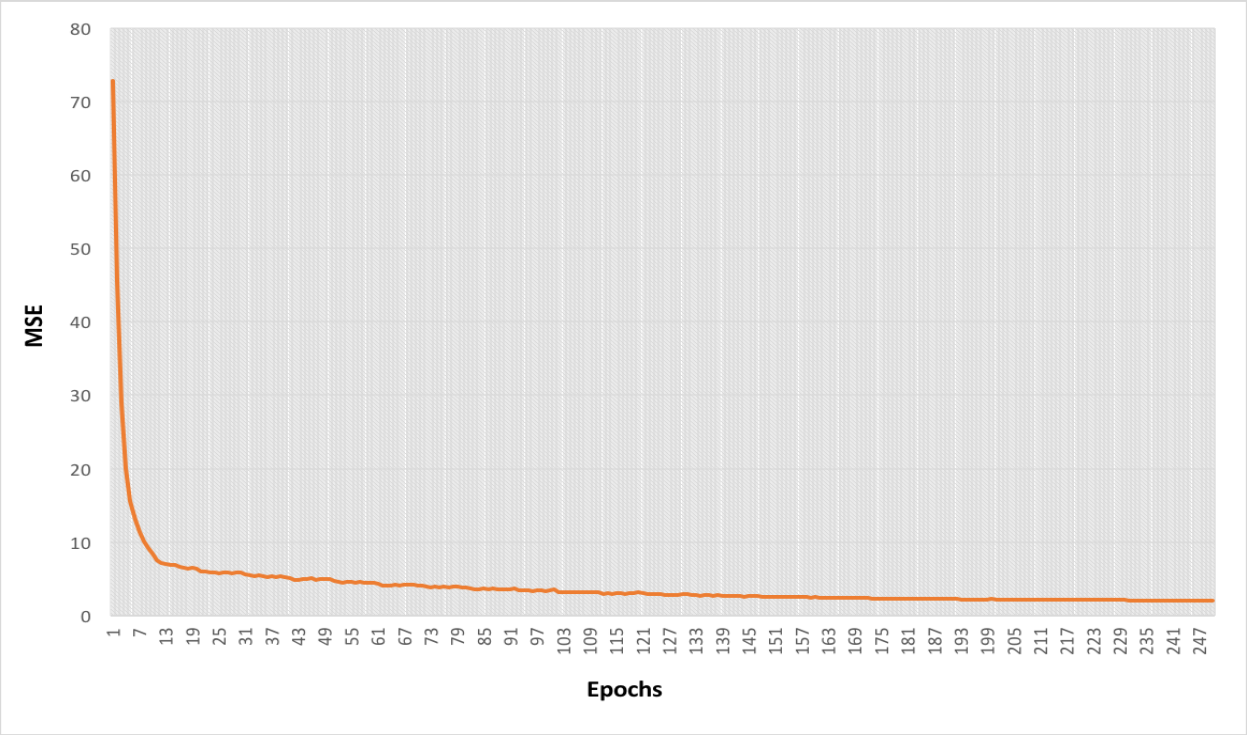

Figure 6: Validation Performance of AutoEncoder model. The y-axis displays the MSE(Mean Square Error); the x-axis displays Epochs.

	AutoEncoder	**Matrix Factorization**	**KNN**
Training accuracy	0.7096	0.8700	/
Validation accuracy	0.5869	0.4211	/
Test accuracy	0.7733	/	0.2471

Table 1: Accuracy comparison among Autoencoder, MF and KNN models

	AutoEncoder	**Matrix Factorization**	**KNN**
MSE	2.0519	2.6509	8.6580

Table 2: MSE comparison among Autoencoder, MF and KNN models

In Figure 5 and 6, our training accuracy keeps increasing and mean square error keeps decreasing during the training part, and finally the accuracy is nearly 80% and MSE is less than 2%. Thus autoencoder model presents efficient learning and inference procedures for movie data sets in our project.
In addition, a variety of models have been proposed for movie recommendation system, for example, matrix factorization (MF), k nearest neighborhood (KNN), etc. In table 1 and 2, we also make comparison among these models. It shows that KNN model performs very poorly which confirms that simple model without training part will not work well in our cases. It also shows that MF is not good enough in this task even though it keeps training the model. Besides, MF training takes lots of time to finish which is in lower efficiency. In autoencoder model, the performance is obvious the best because the MSE is typically smaller in the data sets and the accuracy is relatively higher compared to KNN and MF model.

7.3 Compare output against hypothese

Generally speaking, this study confirms the hypotheses of this study. The accuracy of AutoEncoder is better than Matrix Factorization and k- Near Neighbor. Our result provide a new potential method for recommendation system. Also, the AutoEncoder model is much efficient than Matrix Factorization so AutoEncoder should be more practical than Matrix Factorization.

7.4 Abnormal case explanation (the most important task)

We adopted Stochastic learning gredient to train Matrix Factorization and AutoEncoder, however, the efficiency shows highly different between these two model. We spend about 15min to run the whole data in AutoEncoder model, however, the Matrix Factorization need about 10 hours to finish the program. Also, because of the time limitation(we need to finish the whole project in two weeks), we do not have too much time to tune the model of Matrix Factorization, which means that it might be more accurate than the current model. Another problem of our study is that we did not run the final test of Matrix Factorization.

We supposed to use 4 layers model of AutoEncoder, but we finally adopt 3 layers model because of the overfitting problem. It is obvious that the training accuracy of 4 layers model is much higher than 3 layers. Thus we might need to make more tuning to the 4 layers model to conquer the overfitting problem which might increase the accuracy of the AutoEncoder model.

7.5 Discussion

K nearest neighbor is a relatively naive model and the result is also the least accurate one in our three models. The goal of Matrix Factorization is to retrieve the information hidden in the input, which is used to provide prediction of recommendation of movies to users.

Autoencode is a relatively new and efficient neural network model and previous research also shows a success using autoencoder model in recommendation system. Autoencoder model would learn the latent features and infer the unknown ratings in a more dynamic way. Also, matrix factorization involves find the product of two matrix that are as close as possible to the true ratings. The result of learning through autoencoder would be more flexible instead of the linear transformation.

8. Conclusions and recommendation

8.1 Summary and conclusions

In summary, we implement the movie recommendation system based on autoencoder neural network which is an unsupervised learning algorithm that applies backpropagation, and compare the prediction result with the model based on Matrix Factorization and that based on item-based K nearest neighbor method.

We used the MovieLens latest small dataset to test the performance of our model, which significantly outperforms the item based KNN method. Performance is measured as accuracy and mean squared error these two metrics. Specifically, the average prediction accuracy of KNN is only about 24.71% and the mean squared error is about 8.66, which are of poor quality. The accuracy of matrix factorization model is about 42.1%. The validation accuracy of autoencoder model is about 58% and the test accuracy is about 77%. All these results correspond to our hypothesis that the autoencoder model outperforms the other two baseline models.

8.2 Recommendations for future studies

There are still some aspects that could be improved in our research and work.

First, the size of dataset we used is limited. We could use large datasets to see if larger data size would improve the performance of our model. Currently, we run the model in our local machine. As the data size gets larger, we could run it in Spark cluster more efficiently.

Since the Matrix Factorization algorithm is too much time-consuming, we can improve the algorithm to be more efficient. Also, the Matrix Factorization model can be improved by tune the parameter.

Also, there are also some extensions that could be improved in our model. In this study, we only apply three layer model of AutoEncoder, in the future study, we can try to make the model deeper which means to add more layers. Besides, the parameter can tuning more to see if the model can be improved. Currently, we only used the rating information as input to our autoencoder model. In the future work, we could use other information such as item information in our autoencoder model. Specifically, one approach of implementation is to use two autoencoder model one to learn the user information and the other to learn the item information. Also, by considering the content information of the movies could also improve the performance of our model.

Theoretically, online learning with newly added data would also improve the quality and performance of our autoencoder model.

9. Bibliography

[1] Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, *42*(8).

[2] Pazzani, M. J., & Billsus, D. (2007). Content-based recommendation systems. In *The adaptive web* (pp. 325-341). Springer Berlin Heidelberg.

[3] Gao, T., Li, X., Chai, Y., & Tang, Y. (2016, August). Deep learning with consumer preferences for recommender system. In *Information and Automation (ICIA), 2016 IEEE International Conference on* (pp. 1556-1561). IEEE.

[4] Wang, X., & Wang, Y. (2014, November). Improving content-based and hybrid music recommendation using deep learning. In *Proceedings of the 22nd ACM international conference on Multimedia* (pp. 627-636). ACM.

[5] Wang, H., Wang, N., & Yeung, D. Y. (2015, August). Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1235-1244). ACM.

[6] Elkahky, A. M., Song, Y., & He, X. (2015, May). A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *Proceedings of the 24th International Conference on World Wide Web* (pp. 278-288). ACM.

[7] Takács, G., Pilászy, I., Németh, B., & Tikk, D. (2007). Major components of the gravity recommendation system. *ACM SIGKDD Explorations Newsletter*, *9*(2), 80-83.