CHAPTER 1

INTRODUCTION

1.1 PROBLEM DESCRIPTION

The aim of our project is three-fold:

- To develop a new deep architecture using deep learning algorithms
- To build a recommender system using the above architecture
- To compare its performance against other existing recommender systems

Based on our survey of the existing work in this area (Refer chapter 2), we have identified several things that be done to improve the performance of deep learning-based recommenders and we intend to implement them.

1.2 PROJECT SCOPE

- Recommender systems are widely used in e-commerce and online business applications. Our recommender system can be deployed in any of these areas.
- Our recommender system can be deployed by organizations for applications that have large volumes of user transaction data available.
- Because of our use of deep learning models, the computation requirement of our system in applications will be too big for any home consumer use-case. Hence, our system would be restricted to enterprise use and use in cloud services. For example, our system can be used by an online movie streaming service like Netflix.

CHAPTER 2

REQUIREMENTS

2.1. HARDWARE REQUIREMENTS

- CPU – Quad core unit from Intel or AMD (Core i7 – 5500U or equivalent)
- RAM – 8 GB RAM or more
- HDD – At least 100 GB (Data size dependent)
- GPU – CUDA Enabled NVIDIA Graphics card (CUDA 5.0 or higher)

2.2. SOFTWARE REQUIREMENTS

- OS – Windows, MAC OS or any Linux Distro like Ubuntu 64-bit, Mint 64-bit, etc.
- Programming Language – Python 3.0+
- GPU software – CUDA version 9.0 or higher, cuDNN

2.3. LIBRARY REQUIREMENTS

- Numpy (version 1.1 or higher)
- Scipy (version 1.1.0)
- Keras (version 2.2.0)
- Tensorflow-gpu (version 1.10)
- Pandas (version 0.23.4)
- Matplotlib (version 3.0.0)

CHAPTER 3

LITERATURE REVIEW

3.1. TYPES OF RECOMMENDER SYSTEMS

There are two kinds of recommender frameworks to be specific - Content filtering and collaborative filtering. In collaborative based filtering, items are picked dependent on the connection between the present client and different clients of the framework. In content based filtering, items are prescribed dependent on the relationship amongst items and the client inclinations. What's more, there are additionally unique methodologies. For example – Hybrid, which are produced by blend of both collaborative oriented and content-based filtering which conquer the impediments of the customary strategies.

3.1.1. Content Based Filtering

The content-based filtering mainly focusses on the content data. In this method, the recommendations are made based on the client's preferences for specific attributes that define the item. His preferences are determined by the ratings that he/she has made in the past. The client profile is built based on the historic data of the client's transactions. Once the profile is constructed, the recommendations are made to the client based on how likely it is for him/her to like the item. For example, if a client likes an item A with a certain set of describing attributes, he is probably going to like an item B with similar attribute values.

3.1.2. Collaborative Filtering

This is the opposite of content-based recommender framework. The data sources are past client conduct. The client conduct is found out from the past associations of the client with items exhibited as client thing network. There is no requirement for express client profile creation. As the information sources are now from the past client conduct a communication. The manners by which the association with items accessible to the client can happen are either the express route by rating items or by utilizing the verifiable criticism

deducted from perspectives through the Net, perusing chronicles or client cooperation in interpersonal organizations. Data about the client is assembled and can be utilized to foresee new items utilizing Matrix factorization calculations.

3.2. LIMITATIONS

Collaborative filtering problems	Content based filtering problems
Data sparsity: The collaborative separating isn't helpful if there isn't enough accessibility of information for client and thing ratings	Limited content analysis: In case the content does not contain adequate information for items depiction, the suggestion might be significant to the client.
Cold start: This issue emerges just with new clients or items which has been entered in the framework out of the blue, it's hard to discover a Match as there is no adequate information	Over-specialization: We mean by the over-specialization issue the curiosity's low level of the proposed items. This issue alludes as a fundamental issue of a content-based recommender framework and it propose typically comparative content for the client and afterward no there are no new content to intrigue them
Synonymy problem: This issue emerges when a few items have a similar name or fundamentally the same as names of sections.	New user: At the point when there's no adequate information about the client to construct his/her profile, the proposal might possibly be precise
Gray sheep: This issue jumps out at the clients who has unique sentiments which don't have any significant bearing for any gathering of individuals. In such cases, the client won't be profited from the collaborative oriented separating.	

Table 3.1 – Limitations of Recommenders

3.3. DEEP LEARNING APPROACHES

Deep learning is a branch of machine learning. It very well may be utilized for both directed and undirected learning systems. The deep leaning models can be classified into two on the basis of recommender systems as

- Recommendation using neural building blocks
- Recommendation using hybrid models

Recommendation using neural building blocks

Recommendation using neural building blocks includes all the above discussed models of RNN, CNN, Boltzmann Machine etc., Here the recommendations can be implemented by solely depending on the deep learning techniques or by adding some deep learning or statistical techniques to the deep learning techniques to arrive to a more precise and accurate results.

Most of the neural network models involve in recommending using the collaborative filtering approach since the neural network focusses on comparing and recommending an item based upon the preferences of other users or items. While the algorithms like Recurrent Neural Networks (RNN) focusses on recommending the items based on the content-based filtering.

The most popular Deep Learning model used in Recommender systems is the Restricted Boltzmann Machine (RBM) due to its ability to learn probability distributions over a large set of inputs. In the case of recurrent networks, most researchers choose the Gated Recurrent Unit (GRU) model for their deep learning-based recommendation system. It is preferred over the Long Short-term Memory (LSTM) model due to its simpler implementation and lower compute complexity.

Recommendation using Hybrid models

Recommendation using hybrid models involve recommending the items by combining the neural network models

Some of the most popularly used recommendation algorithm using the hybrid models involve

- CNN with RNN
- RNN with Auto-Encoder
- CNN with Auto-Encoder

Though there are more possible combinations only few have been exploited.

Recommendation systems without deep learning

It is possible to build recommender systems without the deep learning by using statistical models or using some deep architectures. Some of the methods are:

1. **Using similarity**

This method involves recommending the items to the user by finding similarity between the users or the items. The similarity measure can be something like cosine, Jaccard similarity etc.,

2. **Using priority**

This method involves recommending an item to the user by assigning priorities to each item and based on the priority, the item will be recommended.

3. **Using correlation**

This method involves recommending based upon the correlations between the items where correlation is a measure of the strength between the two variables. The most commonly used method of correlation is Pearson.

4. **Using Lightfm**

Lightfm is a package that is available in python that is used to build a simple recommender system. It also combines all the data of the user as a latent factor in order to provide recommendations to the user.

Nonlinear Transformation

As opposed to direct models, profound neural systems is fit for displaying the non-linearity in information with nonlinear initiations, for example, relu, sigmoid, tanh, and so on. This property makes it conceivable to catch the mind boggling and complex user-item communication designs. Traditional techniques, such as grid factorization, factorization machine, inadequate straight model are basically direct models. For instance, grid factorization models the user-item association by straightly joining user-item inactive components Factorization machine is an individual from multivariate direct family. It is settled that neural systems can inexact any ceaseless capacity with a subjective accuracy by shifting the enactment decisions and mixes. This property makes it conceivable to manage complex communication designs and absolutely mirror client's inclination.

Representation Learning

Deep neural networks are efficacious in learning the underlying informative factors and helpful representations from computer file. In general, an outsized quantity of descriptive data regarding things and users is on the market in real-world applications. Usage of this data provides the way to advance our understanding of things and users, thus, leading to a more robust recommender. As such, it's a natural option to apply deep neural networks to illustration learning in recommendation models.

The main Limitations of Deep Learning are:

1. Interpretability
2. Data requirement
3. Extensive Hyper parameter Tuning

Interpretability

Deep learning is well-illustrious to behave as black boxes and provide explicable predictions that appear to be a very difficult task. There is a standard argument against deep neural networks that the hidden weights and the activations are typically non-interpretable and has limited exploitability. However, this concern has typically been mitigated with the arrival of neural network models and have made-up the planet for deep neural models that relish improved interpretability. Whereas decoding individual neurons still cause a challenge for neural models, gift progressive models are already capable of some extent of interpretability and providing explicable recommendation.

Data Requirement

A second limitation is that deep learning needs a lot of data, within the sense that it needs adequate knowledge so as to completely support made parameterization. However, as compared with different domains (such as language or vision) within which tagged data is scarce, it is comparatively straightforward to gather a big quantity of knowledge inside the context of recommender systems analysis.

Extensive Hyper parameter Tuning

The well-established argument against deep learning is that the want for in depth hyper parameter calibration. However, we tend to note that hyper parameter calibration isn't associate exclusive downside of deep learning however machine learning generally (e.g., regularization factors and learning rate equally ought to be tuned for matrix factoring etc.,) Granted, deep learning could introduce extra hyper parameters in some cases.

CHAPTER 4

DESIGN AND IMPLEMENTATION

4.1. PROPOSED MODEL

A standard Long-Short Term Memory unit has the following components as a part of its recurrent node structure:

- *C* - cell state
- *i* - input gate
- *f* - forget gate
- *o* - output gate
- $\tilde{C}_t$ - Modified cell state
- *h* - modified cell state (cell output)

The output gate utilizes a tanh activation function to remove the vanishing gradient issue. The mathematical formulae associated with each gate are:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Where W_i, W_f, W_c and W_o represent the weights of the input, forget gate, cell state and output gate respectively, and the suffixes t and t-1 represent the current time step and the previous time step respectively. In recommendation systems that are based upon user preference, there is no need for the forget layer if no feature in the data is directly related to another. Hence, in our architecture, we drop the forget gate from the LSTM cell. In doing so, the computational complexity and the space complexity are both reduced. This is illustrated in Fig.4.1.

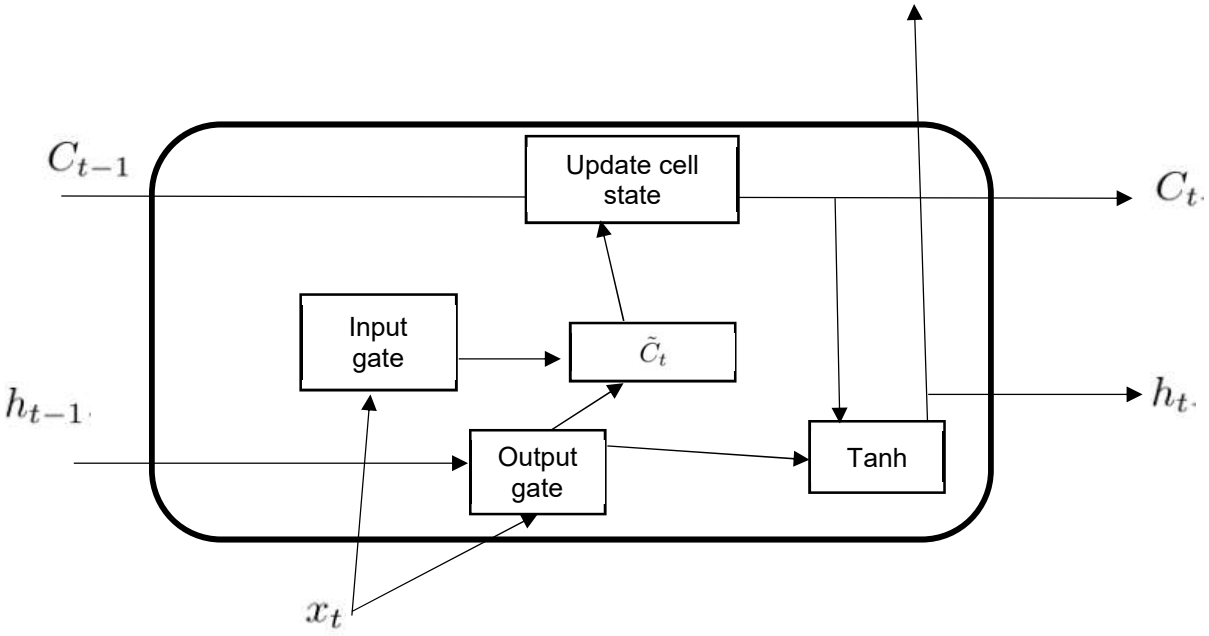


Fig. 4.1 – A single modified LSTM cell

The formulae remain almost the same as before, only now the forget gate input is set to zero.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$
$$C_t = \, i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

In every single cell of the RNN network, the forget gate is initialized to zero. Because of this, the model does not modify its existing knowledge present in the cell state, it merely accumulates information based on the input parameters. One thing to be noted however is that this architecture is not practical if there is a dependency between attributes in the data. For example, in our implementation, there is no direct dependency between different movie genres. If such a dependency existed, then a forget gate would be necessary.

Our recommendation model is a combination of a classifier and a recommender. It first classifies an item as likeable/unlikeable, and then orders the recommendations in order of confidence with which the prediction is made. The working of the recommender system is illustrated in fig.4.2.

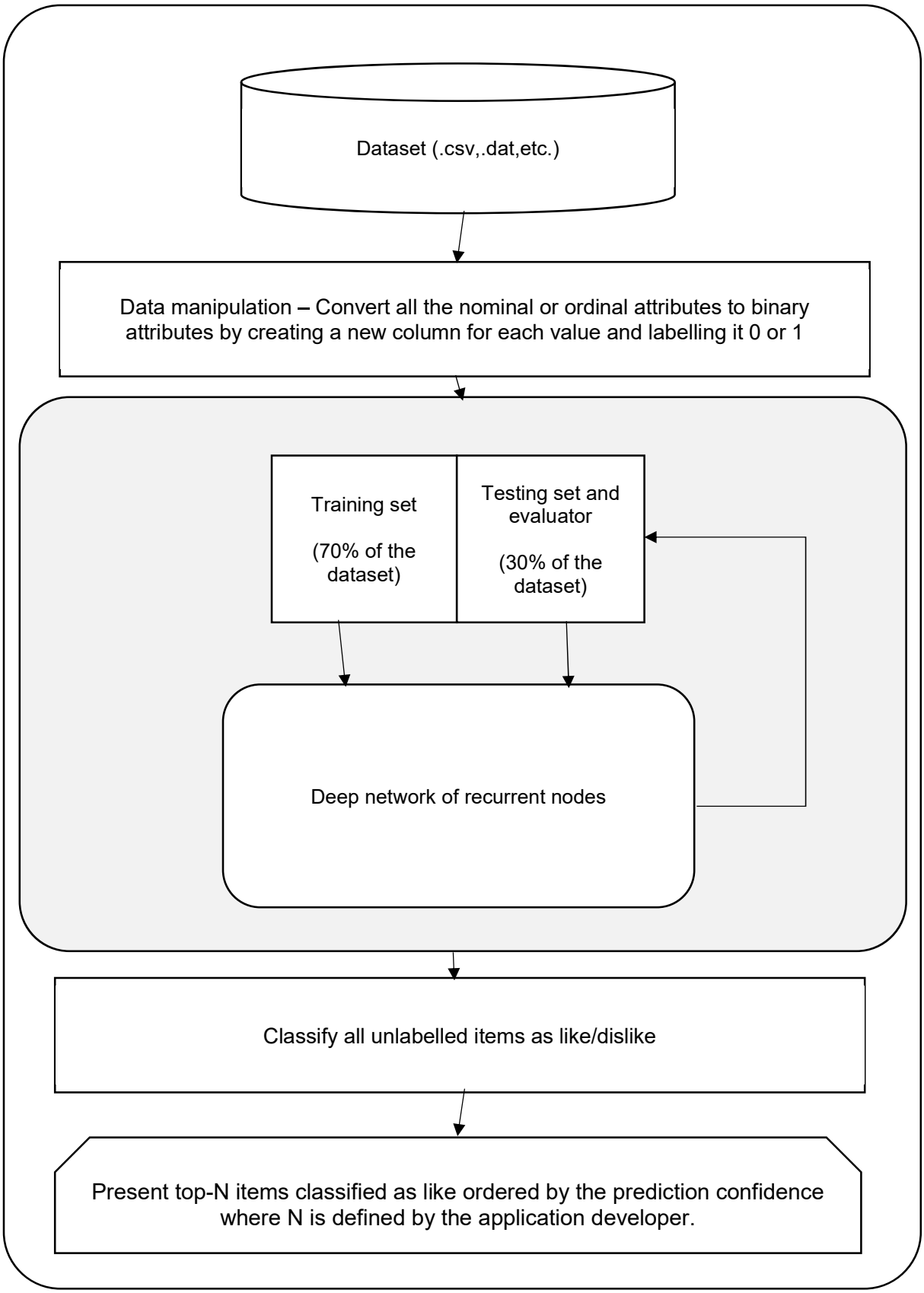


Fig 4.2 – Recommender Architecture

As seen in fig. 4.2, the architecture takes as input a dataset and converts it to a form containing N binary attributes and one binary target class label that indicates whether the user likes/dislikes the item. Here the term user refers to the entity to which the recommendation is made, and the term item refers to each individual entity whose attributes are passed through the network. This manipulation can be done using the pandas library in python. It should be noted that each model is designed for an individual user, so the dataset needs to be trimmed accordingly.

The dataset is then split into a training set and a testing set with a 70-30 split. The training set is first passed through the network. The network consists of N recurrent cells where N is the number of attributes in consideration. The activation function used in each cell state is a Sigmoid activation function, and between recurrent nodes the activation function used is a tanh function. The output dimension is a single binary value with a hard-sigmoid activation function. The loss function used is binary cross entropy. The optimizer function used is the Adam optimizer. The metric used to evaluate the training and testing performance is the accuracy measure, i.e., the ratio between the number of correct classifications to the number of classifications made.

Once the model is trained with all of the training data and once the testing step is complete, the model is then supplied with all of the items in the dataset that the user has not yet seen. For example, if a user has watched only 30 of the movies in a database of 1000 movies, then the recommender passes the other 970 movies through the network. The items are then classified as like or dislike (1/0).

After the classification step comes the recommendation step. In addition to the classification, the confidence measure is also recorded. Ordered by this measure, the top-N items are recommended to the user. The value of N is decided by the application developer based on his/her requirements. Since a recurrent model is used, the model can be updated easily with new instances while also preserving the previous weight matrix.

4.2. IMPLEMENTATION

The implementation of our project can be split into 4 parts:

- LSTM Recommender implementation
- Modified LSTM architecture
- Modified LSTM recommender
- Implementation of recommenders using different models

4.2.1. LSTM Recommender implementation

The LSTM recommender was implemented using python. The Keras library in python was used with tensorflow-gpu as a backend.

```
model = Sequential()
model.add(Embedding(input_dim = 200, output_dim = 50, input_length = input_length))
model.add(LSTM(activation='sigmoid',units = 100, recurrent_activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))

print ('Compiling...')
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Sigmoid activation was used at the output layer and also as the recurrent activation function. The optimizer function chosen was the ADAM optimizer.

4.2.2. Modified LSTM Architecture

To build the modified architecture described in Fig.4.1, first we had to create a sub-class of the LSTMCell class in the keras library.

```
from keras import activations
from keras import initializers
from keras import regularizers
from keras import constraints

class ModifiedLSTMCell(LSTMCell):
```

In the inherited class, the change in the architecture was made at the formula level, where the recurrent value of the forget gate was initialized to zero.

```
i = self.recurrent_activation(x_i + K.dot(h_tm1_i,
                                          self.recurrent_kernel_i))
f = 0
c = f * c_tm1 + i * self.activation(x_c + K.dot(h_tm1_c,
                                                self.recurrent_kernel_c))
```

Another sub-class was created for the LSTM class defined in the keras library. In the LSTM sub-class, the cell builder function was modified to include the ModifiedLSTMCell. The activation function was also changed from hard-sigmoid to sigmoid.

```
class ModifiedLSTM(LSTM):

    def __init__(self, units,
                 activation='tanh',
                 recurrent_activation='sigmoid',
```

4.2.3. Modified LSTM Recommender

After the modified LSTM architecture was built, we constructed a recommender using it. The model was designed as a network of n units, where n equals the number of features. Two additional dropout layers with 0.5 as the parameter were added in the model description. The recurrent activation function used was tanh, and the output layer activation used was a hard sigmoid activation.

```
model = Sequential()
model.add(Embedding(input_dim = 100, output_dim = 50, input_length = input_length))
model.add(Dropout(0.5))
model.add(ModifiedLSTM(activation='sigmoid',units = input_length, recurrent_activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='hard_sigmoid'))
```

4.2.4. Implementation of recommenders using different models

In order to compare our recommender system with recommenders built using other popular deep learning models, we chose to implement 2 recommenders using the following deep architectures

- **Restricted Boltzmann Machine (RBM)**

  Restricted Boltzmann Machine is prominently used with collaborative filtering approach to provide recommendations to users. It is a simplistic deep learning model that contains only two layers: the input layer and visible layer. The model trains data that is mainly based on two phases: input processing and reconstruction of input. Due to the probabilistic nature of the model, it takes in values of user ratings ranging from 0 to 1, with 0 meaning that the user does not like the item and 1 meaning the user likes the item.

  RBM uses conditional probability for modeling hidden features 'h_j':

$$p(h_j = 1|\mathbf{v}) = \frac{1}{1 + e^{(-(b_j + W_j v_i))}} = \sigma(b_j + \sum_i v_i w_{ij})$$

  Where 'σ' is sigmoid function as activation function
  'b_j' is the hidden bias
  'v_i' is the visible unit

  RBM is an energy-based model, which is considered as a metric for quality in certain deep learning models.

  In the model, energy function is given by:

$$E(\mathbf{v}, \mathbf{h}) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

Where 'a_i' is the visible bias

- **Deep Auto Encoder**

Autoencoder is a deep neural network usually used for dimensionality reduction.Simple autoencoder consists of an input layer, one hidden layer and an output layer. The number of neurons are same in both input and output layers. From the given input it performs dimensionality reduction to a latent/hidden state and then it reconstructs the input. The reconstruction error can be reduced using back propagation. It works as follows,

For encoding,

f(x) = σ(W1 x+ b1)

For decoding :

g(y) = σ(W2 y+ b2),

where,

σ - sigmoid or tanh function

x – input

b – bias

The complete autoencoder is represented as

nn(x) = g(f(x))

Assuming that the model works on rows, we get the predicted vector r'i of the form

r'i= σ (Vui)

Activating the hidden units ui iteratively, we get the low rank matrix say U. The final matrix corresponds to the low rank matrix V. The output of the autoencoder performs the matrix factorization R' = σ (UVT). Similarly, it is possible to iterate over the columns to compute r'j and get the final matrix R'.

CHAPTER 5

RESULTS AND ANALYSIS

5.1. TESTING ENVIRONMENT

The model we designed was tested against an LSTM based recommender system a Restricted Boltzmann Machine (RBM) based recommender, and an auto encoder-based recommender. The dataset used by all 3 models was the MovieLens 1M dataset. We chose this particular dataset for our testing since it is a very popular dataset used by many researchers, and we could compare our results with other existing systems to test for correctness.

For the LSTM recommender and for our model, the data manipulation was done using SQL. The movie attributes were first broken down into binary attribute valued columns. The ratings data file was then separated into separate datasets for each user in the system. The SQL statements used can be viewed under APPENDIX A – Code, and the resulting dataset can be viewed in APPENDIX B – Screenshots.

For the RBM based recommender, the data manipulation was done using the pandas library in python. In the dataset, the parameters taken into consideration are User ID, Movie ID, Movie Title and Movie Genre. These parameters are vital for the recommendations done by the model. The testing environment is made to work in such a manner that for a particular user (User ID), recommendations of films sorted in a descending order based on the Recommendation Score will be offered along with the Movie ID, Movie Title, and Movie Genre.

All 3 models were run on a HP Envy – 15 Laptop with a Core i7-5500U processor, 16 GB RAM and a GeForce GTX 850M GPU with CUDA 5.0 capability.

5.2. PERFORMANCE ANALYSIS

The performance of the model is measured based on 2 metrics – training time, and accuracy of the test predictions. All of the models were trained with a dataset training-testing split of 70-30. The comparison was performed in two separate tests. In the first phase of our analysis, the architecture was compared with the LSTM architecture.

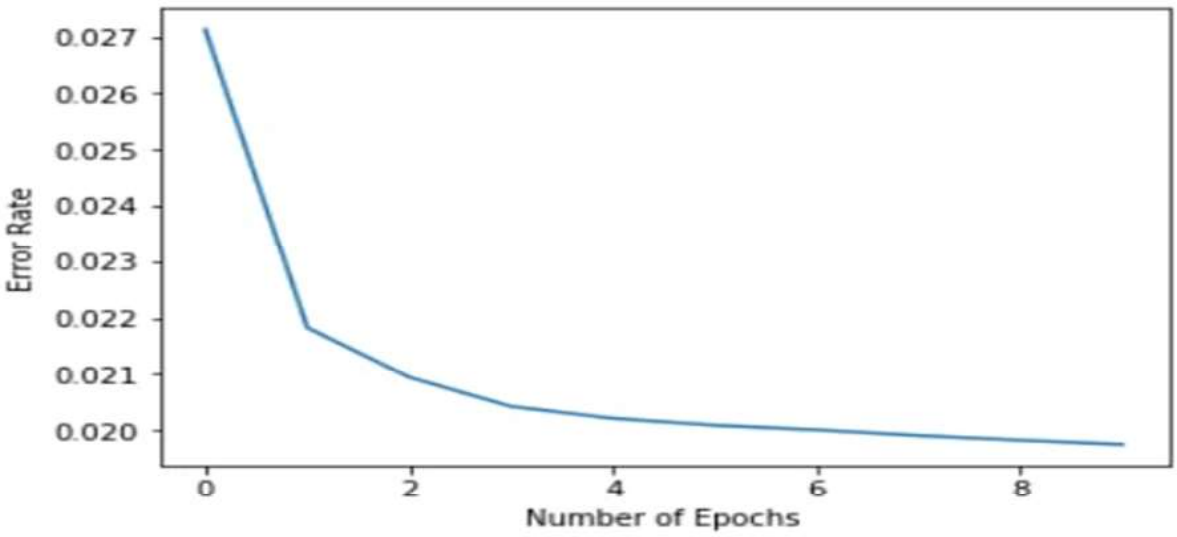
Below are the results from the testing (for user 1):

- LSTM Recommender – Training time : 1 minute 20 seconds
  Prediction accuracy : 84.91%
- Custom Recommender – Training time : 1 minute 14 seconds
  Prediction accuracy : 86.91%

Hence, our model has achieved better results than an LSTM based recommender and also took less time to train due to the absence of one gate in the architecture. The screenshots of the execution, training and results can be viewed in APPENDIX B – Screenshots.

The second phase of our analysis was to compare our model's performance with other recommender systems. In this case, we compared it with a recommender built using an RBM (Restricted Boltzmann Machine) and an Auto Encoder. The performance results were as follows:

- RBM Recommender – Training time : 1 minute 30 seconds
  Prediction accuracy : 86.62%
- AE Recommender – Training time : 2 minute 03 seconds
  Prediction accuracy : 83.20%



Based on the above results, we see that our model's performance with this dataset is comparable with both the RBM and the AE recommender.

CHAPTER 6

CONCLUSION

6.1. OBSERVATIONS

Based on our testing and analysis, we have identified the following areas for improvement with respect to the model:

- The memory of the user's previous preference network can be stored at every training iteration for future use. This can be exploited by the parent application for use cases that require the user's past preferences.
- Although our architecture performs slightly better than the LSTM in this use case, it may not perform at the same level for other use cases. It needs to be further improved for this.
- This recommender cannot be used for a new user in the system. There must be enough data available prior to the creation of the model.
- While the accuracy of the recommendation made is high, it is impractical to develop a separate network for every single user.

6.2. FUTURE RESEARCH DIRECTIONS

There is a lot of scope for future improvement in the field of deep learning in recommender systems. Some areas in which research can be continued further are:

- Increasing the interpretability of the recommendations – The issue with the current deep learning recommenders is that the recommendations made are not clearly interpretable since it is not possible to visualize the working of the network. This can be tackled as a research project in the future.
- Larger domain knowledge in recommendations – Currently the recommender systems only make recommendations based on the data supplied through the dataset. One improvement that could be made to this is allowing for recommendations to be made based on a wider knowledge of the problem domain, and even knowledge from different domains.
- Improved evaluation metrics – Currently the performance of recommenders is measured only by the accuracy of the predictions made. However, this is not a very good way of determining the quality of a recommendation. It is important to find a new way to evaluate the performance of recommender systems.

BIBLIOGRAPHY

1. Linden, G., Smith, B., and York, J. (2003). Amazon. com recommendations: Item-to-item collaborative filtering. Internet Computing, IEEE, 7(1), 76-80.

2. Ricci, F., Rokach, L., Shapira, B., and Kantor, P.B. (2011). Introduction to Recommender Systems Handbook. Springer.

3. Segaran, T. (2007). Programming Collective Intelligence: Building Smart Web 2.0 Applications. O'Reilly.

4. Waila, P.; Singh, V.; Singh, M. (26 April 2016). "A Scientometric Analysis of Research in Recommender Systems". Journal of Scientometric Research: 71–84. doi:10.5530/jscires.5.1.1

5. Liu H., Kong X., Bai X., Wang W., Bekele T. M., and Xia F., "Context-based collaborative filtering for citation recommendation," IEEE Access, vol. 3, pp. 1695–1703, 2015.

6. B. Gipp, J. Beel, and C. Hentschel, "Scienstein: A research paper recommender system," in Proceedings of the international conference on emerging trends in computing (icetic'09), 2009.

7. P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, "GroupLens: an open architecture for collaborative filtering of netnews," in Proceedings of the 1994 ACM conference on Computer supported cooperative work, 1994

8. Azaria, A. Hassidim, S. Kraus, A. Eshkol, O. Weintraub, and I. Netanely, "Movie recommender system for profit maximization," in Proceedings of the 7th ACM conference on Recommender systems, 2013

APPENDIX A - CODE

LSTM BASED RECOMMENDER (PYTHON)

```
import tensorflow as tf
from keras import backend as K
from keras.layers import Layer, Dense, Dropout, LSTMCell, LSTM, GRU, RNN, Embedding
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
from random import shuffle
df = pd.read_csv("C:/Users/siddharth/Documents/progs/ml-1m/ml-1m/user1_ratings.csv")
print(df)
df1 = df[['a1','a2','a3','a4','a5','a6','a7','a8','a9','a10','a11','a12','a13','a14','a15','a16','a17','a18']]
df2 = df[['like']]
dataset = df1.values
dataset = dataset.astype('float32')
dataset2 = df2.values
dataset2 = dataset2.astype('float32')
train_size = int(len(dataset) * 0.65)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size], dataset[train_size:len(dataset)]
train_op, test_op = dataset2[0:train_size], dataset2[train_size:len(dataset2)]
print(len(train), len(test))
print(dataset)
print(train)
print(test)
print(train_op)
print(test_op)
def create_model(input_length):
    print ('Creating model...')
    model = Sequential()
    model.add(Embedding(input_dim = 200, output_dim = 50, input_length = input_length))
    model.add(LSTM(activation='sigmoid',units = 100, recurrent_activation='sigmoid'))
    model.add(Dense(1, activation='sigmoid'))
```

```
  print ('Compiling...')
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
    return model

model = create_model(len(train[0]))
hist = model.fit(train, train_op, epochs=100,batch_size=10)

scores = model.evaluate(dataset, dataset2, verbose=0)
print("Test accuracy : %.2f%%" %((scores[1]*100)))
```

MODIFIED LSTM RECOMMENDER CODE (PYTHON)

```
import tensorflow as tf
from keras import backend as K
from keras.layers import Layer, Dense, Dropout, LSTMCell, LSTM, GRU, RNN, Embedding
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
from random import shuffle
df = pd.read_csv("C:/Users/siddharth/Documents/progs/ml-1m/ml-1m/user1_ratings.csv")
print(df)
df1 = df[['a1','a2','a3','a4','a5','a6','a7','a8','a9','a10','a11','a12','a13','a14','a15','a16','a17','a18']]
df2 = df[['like']]
dataset = df1.values
dataset = dataset.astype('float32')
dataset2 = df2.values
dataset2 = dataset2.astype('float32')
train_size = int(len(dataset) * 0.65)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size], dataset[train_size:len(dataset)]
train_op, test_op = dataset2[0:train_size], dataset2[train_size:len(dataset2)]
print(len(train), len(test))
print(dataset)
print(train)
print(test)
print(train_op)
print(test_op)
from keras import activations
from keras import initializers
from keras import regularizers
from keras import constraints
```

```
class ModifiedLSTMCell(LSTMCell):

    def __init__(self, units,
                 activation='tanh',
                 recurrent_activation='sigmoid',
                 use_bias=True,
                 kernel_initializer='glorot_uniform',
                 recurrent_initializer='orthogonal',
                 bias_initializer='zeros',
                 unit_forget_bias=True,
                 kernel_regularizer=None,
                 recurrent_regularizer=None,
                 bias_regularizer=None,
                 kernel_constraint=None,
                 recurrent_constraint=None,
                 bias_constraint=None,
                 dropout=0.,
                 recurrent_dropout=0.,
                 implementation=1,
                 **kwargs):
        super(LSTMCell, self).__init__(**kwargs)
        self.units = units
        self.activation = activations.get(activation)
        self.recurrent_activation = activations.get(recurrent_activation)
        self.use_bias = use_bias

        self.kernel_initializer = initializers.get(kernel_initializer)
        self.recurrent_initializer = initializers.get(recurrent_initializer)
        self.bias_initializer = initializers.get(bias_initializer)
        self.unit_forget_bias = unit_forget_bias

        self.kernel_regularizer = regularizers.get(kernel_regularizer)
        self.recurrent_regularizer = regularizers.get(recurrent_regularizer)
        self.bias_regularizer = regularizers.get(bias_regularizer)

        self.kernel_constraint = constraints.get(kernel_constraint)
        self.recurrent_constraint = constraints.get(recurrent_constraint)
        self.bias_constraint = constraints.get(bias_constraint)

        self.dropout = min(1., max(0., dropout))
        self.recurrent_dropout = min(1., max(0., recurrent_dropout))
        self.implementation = implementation
        self.state_size = (self.units, self.units)
        self.output_size = self.units
        self._dropout_mask = None
        self._recurrent_dropout_mask = None

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.kernel = self.add_weight(shape=(input_dim, self.units * 4),
                                      name='kernel',
                                      initializer=self.kernel_initializer,
                                      regularizer=self.kernel_regularizer,
                                       constraint=self.kernel_constraint)
        self.recurrent_kernel = self.add_weight(
```

```
shape=(self.units, self.units * 4),
            name='recurrent_kernel',
            initializer=self.recurrent_initializer,
            regularizer=self.recurrent_regularizer,
            constraint=self.recurrent_constraint)

        if self.use_bias:
            if self.unit_forget_bias:
                def bias_initializer(_, *args, **kwargs):
                    return K.concatenate([
                        self.bias_initializer((self.units,), *args, **kwargs),
                        initializers.Ones()((self.units,), *args, **kwargs),
                        self.bias_initializer((self.units * 2,), *args, **kwargs),
                    ])
            else:
                bias_initializer = self.bias_initializer
            self.bias = self.add_weight(shape=(self.units * 4,),
                                        name='bias',
                                        initializer=bias_initializer,
                                        regularizer=self.bias_regularizer,
                                        constraint=self.bias_constraint)
        else:
            self.bias = None

        self.kernel_i = self.kernel[:, :self.units]
        self.kernel_f = self.kernel[:, self.units: self.units * 2]
        self.kernel_c = self.kernel[:, self.units * 2: self.units * 3]
        self.kernel_o = self.kernel[:, self.units * 3:]

        self.recurrent_kernel_i = self.recurrent_kernel[:, :self.units]
        self.recurrent_kernel_f = (
            self.recurrent_kernel[:, self.units: self.units * 2])
        self.recurrent_kernel_c = (
            self.recurrent_kernel[:, self.units * 2: self.units * 3])
        self.recurrent_kernel_o = self.recurrent_kernel[:, self.units * 3:]

        if self.use_bias:
            self.bias_i = self.bias[:self.units]
            self.bias_f = self.bias[self.units: self.units * 2]
            self.bias_c = self.bias[self.units * 2: self.units * 3]
            self.bias_o = self.bias[self.units * 3:]
        else:
            self.bias_i = None
            self.bias_f = None
            self.bias_c = None
            self.bias_o = None
        self.built = True

    def call(self, inputs, states, training=None):
        if 0 < self.dropout < 1 and self._dropout_mask is None:
            self._dropout_mask = _generate_dropout_mask(
                K.ones_like(inputs),
                self.dropout,
                 training=training,
```

```
            count=4)
        if (0 < self.recurrent_dropout < 1 and
            self._recurrent_dropout_mask is None):
            self._recurrent_dropout_mask = _generate_dropout_mask(
                K.ones_like(states[0]),
                self.recurrent_dropout,
                training=training,
                count=4)

        # dropout matrices for input units
        dp_mask = self._dropout_mask
        # dropout matrices for recurrent units
        rec_dp_mask = self._recurrent_dropout_mask

        h_tm1 = states[0]  # previous memory state
        c_tm1 = states[1]  # previous carry state

        if self.implementation == 1:
            if 0 < self.dropout < 1.:
                inputs_i = inputs * dp_mask[0]
                inputs_f = inputs * dp_mask[1]
                inputs_c = inputs * dp_mask[2]
                inputs_o = inputs * dp_mask[3]
            else:
                inputs_i = inputs
                inputs_f = inputs
                inputs_c = inputs
                inputs_o = inputs
            x_i = K.dot(inputs_i, self.kernel_i)
            x_f = K.dot(inputs_f, self.kernel_f)
            x_c = K.dot(inputs_c, self.kernel_c)
            x_o = K.dot(inputs_o, self.kernel_o)
            if self.use_bias:
                x_i = K.bias_add(x_i, self.bias_i)
                x_f = K.bias_add(x_f, self.bias_f)
                x_c = K.bias_add(x_c, self.bias_c)
                x_o = K.bias_add(x_o, self.bias_o)

            if 0 < self.recurrent_dropout < 1.:
                h_tm1_i = h_tm1 * rec_dp_mask[0]
                h_tm1_f = h_tm1 * rec_dp_mask[1]
                h_tm1_c = h_tm1 * rec_dp_mask[2]
                h_tm1_o = h_tm1 * rec_dp_mask[3]
            else:
                h_tm1_i = h_tm1
                h_tm1_f = h_tm1
                h_tm1_c = h_tm1
                h_tm1_o = h_tm1
            i = self.recurrent_activation(x_i + K.dot(h_tm1_i,
                                          self.recurrent_kernel_i))
            f = 0
            c = f * c_tm1 + i * self.activation(x_c + K.dot(h_tm1_c,
                                                self.recurrent_kernel_c))
            o = self.recurrent_activation(x_o + K.dot(h_tm1_o,
                                          self.recurrent_kernel_o))
```

```
else:
        if 0. < self.dropout < 1.:
            inputs *= dp_mask[0]
        z = K.dot(inputs, self.kernel)
        if 0. < self.recurrent_dropout < 1.:
            h_tm1 *= rec_dp_mask[0]
        z += K.dot(h_tm1, self.recurrent_kernel)
        if self.use_bias:
            z = K.bias_add(z, self.bias)

        z0 = z[:, :self.units]
        z1 = z[:, self.units: 2 * self.units]
        z2 = z[:, 2 * self.units: 3 * self.units]
        z3 = z[:, 3 * self.units:]

        i = self.recurrent_activation(z0)
        f = self.recurrent_activation(z1)
        c = f * c_tm1 + i * self.activation(z2)
        o = self.recurrent_activation(z3)

    h = o * self.activation(c)
    if 0 < self.dropout + self.recurrent_dropout:
        if training is None:
            h._uses_learning_phase = True
    return h, [h, c]

def get_config(self):
    config = {'units': self.units,
              'activation': activations.serialize(self.activation),
              'recurrent_activation':
                  activations.serialize(self.recurrent_activation),
              'use_bias': self.use_bias,
              'kernel_initializer':
                  initializers.serialize(self.kernel_initializer),
              'recurrent_initializer':
                  initializers.serialize(self.recurrent_initializer),
              'bias_initializer': initializers.serialize(self.bias_initializer),
              'unit_forget_bias': self.unit_forget_bias,
              'kernel_regularizer':
                  regularizers.serialize(self.kernel_regularizer),
              'recurrent_regularizer':
                  regularizers.serialize(self.recurrent_regularizer),
              'bias_regularizer': regularizers.serialize(self.bias_regularizer),
              'kernel_constraint': constraints.serialize(self.kernel_constraint),
              'recurrent_constraint':
                  constraints.serialize(self.recurrent_constraint),
              'bias_constraint': constraints.serialize(self.bias_constraint),
              'dropout': self.dropout,
              'recurrent_dropout': self.recurrent_dropout,
              'implementation': self.implementation}
    base_config = super(LSTMCell, self).get_config()
    return dict(list(base_config.items()) + list(config.items()))
```

```
class ModifiedLSTM(LSTM):

    def __init__(self, units,
                 activation='tanh',
                 recurrent_activation='sigmoid',
                 use_bias=True,
                 kernel_initializer='glorot_uniform',
                 recurrent_initializer='orthogonal',
                 bias_initializer='zeros',
                 unit_forget_bias=True,
                 kernel_regularizer=None,
                 recurrent_regularizer=None,
                 bias_regularizer=None,
                 activity_regularizer=None,
                 kernel_constraint=None,
                 recurrent_constraint=None,
                 bias_constraint=None,
                 dropout=0.,
                 recurrent_dropout=0.,
                 implementation=1,
                 return_sequences=False,
                 return_state=False,
                 go_backwards=False,
                 stateful=False,
                 unroll=False,
                 **kwargs):
        if implementation == 0:
            warnings.warn('`implementation=0` has been deprecated, '
                          'and now defaults to `implementation=1`.'
                          'Please update your layer call.')
        if K.backend() == 'theano' and (dropout or recurrent_dropout):
            warnings.warn(
                'RNN dropout is no longer supported with the Theano backend '
                'due to technical limitations. '
                'You can either set `dropout` and `recurrent_dropout` to 0, '
                'or use the TensorFlow backend.')
            dropout = 0.
            recurrent_dropout = 0.

        cell = ModifiedLSTMCell(units,
                        activation=activation,
                        recurrent_activation=recurrent_activation,
                        use_bias=use_bias,
                        kernel_initializer=kernel_initializer,
                        recurrent_initializer=recurrent_initializer,
                        unit_forget_bias=unit_forget_bias,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=kernel_regularizer,
                        recurrent_regularizer=recurrent_regularizer,
                        bias_regularizer=bias_regularizer,
                        kernel_constraint=kernel_constraint,
                        recurrent_constraint=recurrent_constraint,
                        bias_constraint=bias_constraint,
                         dropout=dropout,
                         recurrent_dropout=recurrent_dropout,
```

```
                implementation=implementation)
    super(LSTM, self).__init__(cell,
                           return_sequences=return_sequences,
                           return_state=return_state,
                           go_backwards=go_backwards,
                           stateful=stateful,
                           unroll=unroll,
                           **kwargs)
    self.activity_regularizer = regularizers.get(activity_regularizer)

def call(self, inputs, mask=None, training=None, initial_state=None):
    self.cell._dropout_mask = None
    self.cell._recurrent_dropout_mask = None
    return super(LSTM, self).call(inputs,
                              mask=mask,
                              training=training,
                              initial_state=initial_state)

@property
def units(self):
    return self.cell.units

@property
def activation(self):
    return self.cell.activation

@property
def recurrent_activation(self):
    return self.cell.recurrent_activation

@property
def use_bias(self):
    return self.cell.use_bias

@property
def kernel_initializer(self):
    return self.cell.kernel_initializer

@property
def recurrent_initializer(self):
    return self.cell.recurrent_initializer

@property
def bias_initializer(self):
    return self.cell.bias_initializer

@property
def unit_forget_bias(self):
    return self.cell.unit_forget_bias

@property
def kernel_regularizer(self):
    return self.cell.kernel_regularizer
@property
def recurrent_Regularizer(self):
```

```
        return self.cell.recurrent_regularizer

    @property
    def bias_regularizer(self):
        return self.cell.bias_regularizer

    @property
    def kernel_constraint(self):
        return self.cell.kernel_constraint

    @property
    def recurrent_constraint(self):
        return self.cell.recurrent_constraint

    @property
    def bias_constraint(self):
        return self.cell.bias_constraint

    @property
    def dropout(self):
        return self.cell.dropout

    @property
    def recurrent_dropout(self):
        return self.cell.recurrent_dropout

    @property
    def implementation(self):
        return self.cell.implementation

    def get_config(self):
        config = {'units': self.units,
                  'activation': activations.serialize(self.activation),
                  'recurrent_activation':
                      activations.serialize(self.recurrent_activation),
                  'use_bias': self.use_bias,
                  'kernel_initializer':
                      initializers.serialize(self.kernel_initializer),
                  'recurrent_initializer':
                      initializers.serialize(self.recurrent_initializer),
                  'bias_initializer': initializers.serialize(self.bias_initializer),
                  'unit_forget_bias': self.unit_forget_bias,
                  'kernel_regularizer':
                      regularizers.serialize(self.kernel_regularizer),
                  'recurrent_regularizer':
                      regularizers.serialize(self.recurrent_regularizer),
                  'bias_regularizer': regularizers.serialize(self.bias_regularizer),
                  'activity_regularizer':
                      regularizers.serialize(self.activity_regularizer),
                  'kernel_constraint': constraints.serialize(self.kernel_constraint),
                  'recurrent_constraint':
                      constraints.serialize(self.recurrent_constraint),
                  'bias_constraint': constraints.serialize(self.bias_constraint),
                  'dropout': self.dropout,
```

```
                'recurrent_dropout': self.recurrent_dropout,
                'implementation': self.implementation}
        base_config = super(LSTM, self).get_config()
        del base_config['cell']
        return dict(list(base_config.items()) + list(config.items()))

    @classmethod
    def from_config(cls, config):
        if 'implementation' in config and config['implementation'] == 0:
            config['implementation'] = 1
        return cls(**config)


def _generate_dropout_mask(ones, rate, training=None, count=1):
    def dropped_inputs():
        return K.dropout(ones, rate)

    if count > 1:
        return [K.in_train_phase(
            dropped_inputs,
            ones,
            training=training) for _ in range(count)]
    return K.in_train_phase(
        dropped_inputs,
        ones,
        training=training)


def _standardize_args(inputs, initial_state, constants, num_constants):

    if isinstance(inputs, list):
        assert initial_state is None and constants is None
        if num_constants is not None:
            constants = inputs[-num_constants:]
            inputs = inputs[:-num_constants]
        if len(inputs) > 1:
            initial_state = inputs[1:]
        inputs = inputs[0]

    def to_list_or_none(x):
        if x is None or isinstance(x, list):
            return x
        if isinstance(x, tuple):
            return list(x)
        return [x]

    initial_state = to_list_or_none(initial_state)
    constants = to_list_or_none(constants)

    return inputs, initial_state, constants

def create_my_model(input_length):
    print ('Creating model...')
    model = Sequential()
    model.add(Embedding(input_dim = 200, output_dim = 50, input_length = Input_Length))
```

```
model.add(Dropout(0.5))
   model.add(ModifiedLSTM(activation='sigmoid',units = input_length,
recurrent_activation='tanh'))
   model.add(Dropout(0.5))
   model.add(Dense(1, activation='hard_sigmoid'))

   print ('Compiling...')
   model.compile(loss='binary_crossentropy',
                 optimizer='adam',
                 metrics=['accuracy'])
   return model

model = create_my_model(len(train[0]))
hist = model.fit(train, train_op, epochs=100,batch_size=10)

scores = model.evaluate(dataset, dataset2, verbose=0)
print("Test accuracy : %.2f%%" %((scores[1]*100)))
```

RBM BASED RECOMMENDER (PYTHON)

```
import pandas as pd

movies = pd.read_csv('movies.csv', header = None)
ratings = pd.read_csv('ratings.csv', header = None)

movies.columns = ['MovieID']
ratings.columns = ['UserID', 'MovieID', 'Rating']

#To remove indexing errors
movies['List Index'] = movies.index

#Merging movies with ratings by MovieID
movrat = movies.merge(ratings, on = 'MovieID')

#Group by UserID
userGroup = movrat.groupby('UserID')
userGroup.head()

#Amount of users used for training
amountOfUsedUsers = 1500

#Training list
trainingSet = []

#For each user in the group:
for userID, curUser in userGroup:

   #Create a temp that stores every movie's rating
   temp = [0]*len(movies)
```

```
#For each movie in curUser's movie list
    for num, movie in curUser.iterrows():

        #Divide the rating by 5 and store
        temp[movie['List Index']] = movie['Rating']/5.0

    #Add the list of ratings into the training list
    trainingSet.append(temp)
    if amountOfUsers == 0:
        break
    amountOfUsers -= 1
#Creating placeholder variables so that they can be initialized later
import tensorflow as tf
hiddenLayer = 30
visibleLayer = len(movies)
visibleBias = tf.placeholder("float", [visibleLayer]) #Number of unique movies
hiddenBias = tf.placeholder("float", [hiddenLayer]) #Number of features we're going to learn
w = tf.placeholder("float", [visibleLayer, hiddenLayer])

#Creating the visible and hidden layer units and setting sigmoid activation function
#Phase 1: Input Processing
visibleUnit = tf.placeholder("float", [None, visibleLayer])
hiddenUnit = tf.nn.sigmoid(tf.matmul(visibleUnit, w) + hiddenBias)

#Phase 2: Reconstruction
visibleUnit1 = tf.nn.sigmoid(tf.matmul(hiddenUnit, tf.transpose(w)) + visibleBias)
hiddenUnit1 = tf.nn.sigmoid(tf.matmul(visibleUnit1, w) + hiddenBias)

#RBM training parameters and functions:
#Learning rate
alpha = 1.0

#Create the gradients
wPosGrad = tf.matmul(tf.transpose(visibleUnit), hiddenUnit)
wNegGrad = tf.matmul(tf.transpose(visibleUnit1), hiddenUnit1)

#Calculating the Contrastive Divergence
cd = (wPosGrad - wNegGrad) / tf.to_float(tf.shape(visibleUnit)[0])

#Create methods to update the weights and biases
updatew = w + alpha * cd
updateVisibleBias = visible Bias + alpha * tf.reduce_mean(visibleUnit - visibleUnit1, 0)
updateHiddenBias = hiddenBias + alpha * tf.reduce_mean(hiddenUnit - hiddenUnit1, 0)

#Mean Absolute Error as error function
error = visibleUnit - visibleUnit1
errorSum = tf.reduce_mean(error * error)

#Initializing the variables
#Current weight
import numpy as np
curw = np.zeros([visibleLayer, hiddenLayer], np.float32)

#Current visible unit biases
curVisibleBias = np.zeros([visibleLayer], np.float32)
```

```
#Current hidden unit biases
curHiddenBias = np.zeros([hiddenLayer], np.float32)

#Previous weight
prevw = np.zeros([visibleLayer, hiddenLayer], np.float32)

#Previous visible unit biases
prevVisibleBias = np.zeros([visibleLayer], np.float32)

#Previous hidden unit biases
prevHiddenBias = np.zeros([hiddenLayer], np.float32)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

#Training RBM with 100 epochs
epochs = 100
batchsize = 150
errors = []
for i in range(epochs):
    for start, end in zip( range(0, len(trainingSet), batchsize), range(batchsize, len(trainingSet), batchsize)):
        batch = trainingSet[start:end]
        curw = sess.run(updatew, feed_dict={visibleUnit: batch, w: prevw, visibleBias: prevVisibleBias,
hiddenBias: prevHiddenBias})
        curVisibleBias = sess.run(updateVisibleBias, feed_dict={visibleUnit: batch, w: prevw, visibleBias:
prevVisibleBias, hiddenBias: prevHiddenBias})
        curHiddenBias = sess.run(updateHiddenBias, feed_dict={visibleUnit: batch, w: prevw, visibleBias:
prevVisibleBias, hiddenBias: prevHiddenBias})
        prevw = curw
        prevVisibleBias = curVisibleBias
        prevHiddenBias = curHiddenBias
    errors.append(sess.run(errorSum, feed_dict={visibleUnit: trainingSet, w: curw, visibleBias:
curVisibleBias, hiddenBias: curHiddenBias}))
    print (errors[-1])
    import matplotlib.pyplot as plt
    plt.plot(errors)
    plt.xlabel('Number of Epochs')
    plt.ylabel('Error Rate')
plt.show()

#Selecting the input user
inputUser = [trainingSet[100]]
#Feeding in the user and reconstructing the input
hh0 = tf.nn.sigmoid(tf.matmul(visibleUnit, w) + hiddenBias)
vv1 = tf.nn.sigmoid(tf.matmul(hh0, tf.transpose(w)) + visibleBias)
feed = sess.run(hh0, feed_dict={ visibleUnit: inputUser, w: prevw, hiddenBias: prevHiddenBias})
rec = sess.run(vv1, feed_dict={ hh0: feed, w: prevw, visibleBias: prevVisibleBias})

scoredMovies = movies
scoredMovies["Recommendation Score"] = rec[0]
scoredMovies.sort_values(["Recommendation Score"], ascending=False).head(20)

movrat.iloc[100]
```

```
movies100 = movrat[movrat['UserID']==301]
movies100.head()

#Merging movies_df with ratings_df by MovieID
movrat100 = scoredMovies.merge(movies100, on = 'MovieID', how = 'outer')

#Dropping unecessary columns
merged100 = merged100.drop('List Index_y', axis=1).drop('UserID', axis=1)

#Displaying top 20 films based on Recommendation Score to the user
merged100.sort_values(["Recommendation Score"], ascending=False).head(20)
```

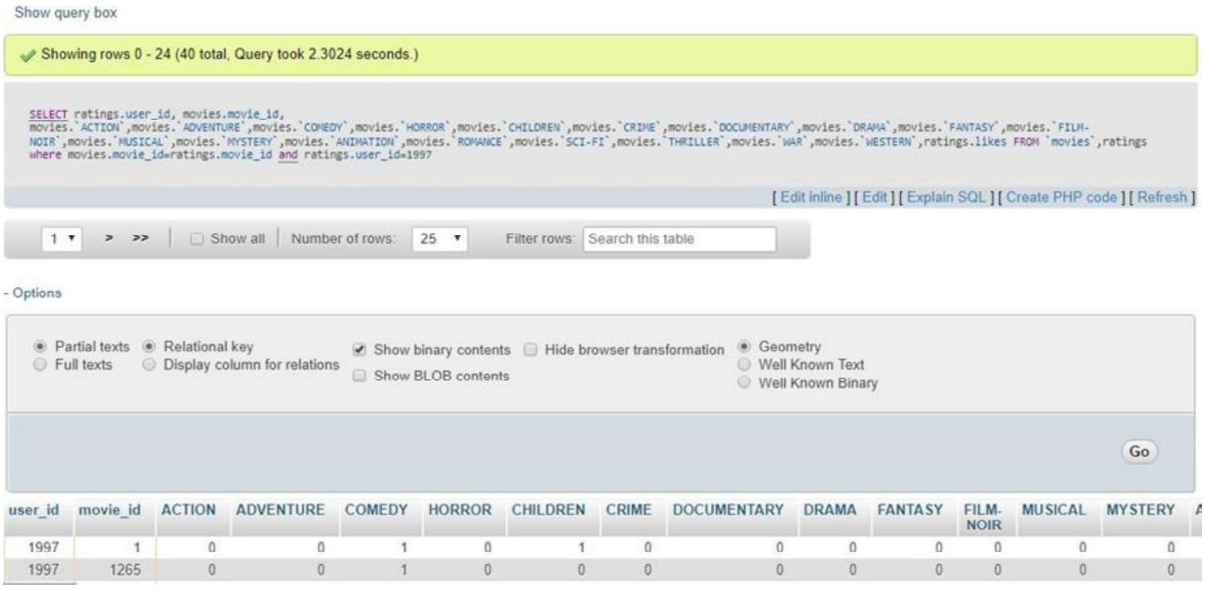
AUTO-ENCODER BASED RECOMMENDER (PYTHON)

```
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
ratings = pd.read_csv('ml-1m/ratings.dat',\
        sep="::", header = None, engine='python')
ratings_pivot = pd.pivot_table(ratings[[0,1,2]],\
        values=2, index=0, columns=1 ).fillna(0)
X_train, X_test = train_test_split(ratings_pivot, train_size=0.8)
n_nodes_inpl = 3706 #input layer nodes
n_nodes_hl1  = 256  #hidden layer nodes
n_nodes_outl = 3706 #output layer nodes
hidden_1_layer_vals = {'weights':tf.Variable(tf.random_normal([n_nodes_inpl+1,n_nodes_hl1]))}
output_layer_vals = {'weights':tf.Variable(tf.random_normal([n_nodes_hl1+1,n_nodes_outl]))}
input_layer = tf.placeholder('float', [None, 3706])
input_layer_const = tf.fill( [tf.shape(input_layer)[0], 1] ,1.0  )
input_layer_concat =  tf.concat([input_layer, input_layer_const], 1)
layer_1 = tf.nn.sigmoid(tf.matmul(input_layer_concat, hidden_1_layer_vals['weights']))
layer1_const = tf.fill([tf.shape(layer_1)[0], 1] ,1.0)
layer_concat =  tf.concat([layer_1, layer1_const], 1)
output_layer = tf.matmul( layer_concat,output_layer_vals['weights'])
output_true = tf.placeholder('float', [None, 3706])
meansq = tf.reduce_mean(tf.square(output_layer - output_true))
learn_rate = 0.1 #Assumed for now
optimizer = tf.train.AdagradOptimizer(learn_rate).minimize(meansq)
init = tf.global_variables_initializer()
sess = tf.Session()
```

```
sess.run(init)
batch_size = 100
hm_epochs =200
tot_users = X_train.shape[0]
for epoch in range(hm_epochs):
    epoch_loss = 0    # initializing error as 0
    for i in range(int(tot_users/batch_size)):
        epoch_x = X_train[ i*batch_size : (i+1)*batch_size ]
        _, c = sess.run([optimizer, meansq], feed_dict={input_layer: epoch_x, output_true:
epoch_x})
        epoch_loss += c
    output_train = sess.run(output_layer, feed_dict={input_layer:X_train})
    output_test = sess.run(output_layer, feed_dict={input_layer:X_test})
    print('MSE train', MSE(output_train, X_train),'MSE test', MSE(output_test, X_test))
    print('Epoch', epoch, '/', hm_epochs, 'loss:',epoch_loss)
sample_user = X_test.iloc[99,:]
sample_user_pred = sess.run(output_layer, feed_dict={input_layer:[sample_user]})
print(sample_user_pred)
sample_user = X_test.iloc[25,:]
sample_user_pred = sess.run(output_layer, feed_dict={input_layer:[sample_user]})
print(sample_user_pred)
```

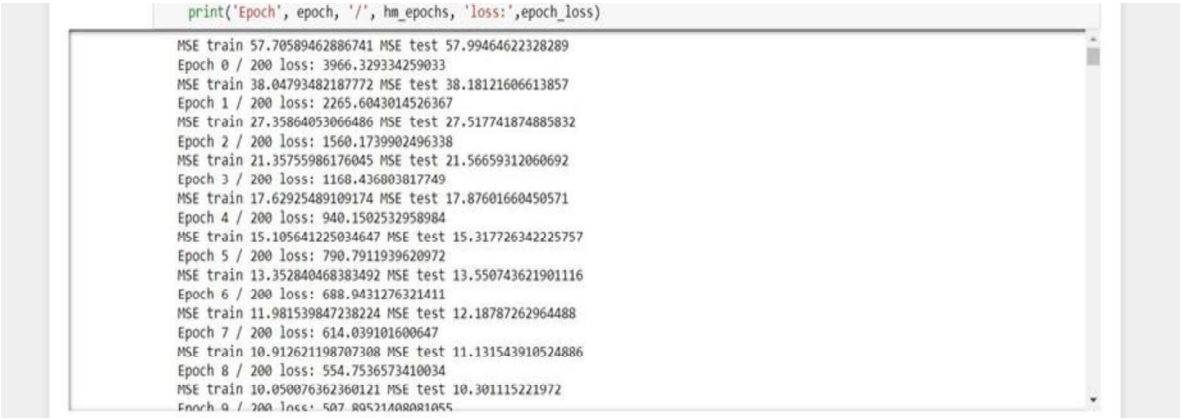
APPENDIX B - SCREENSHOTS

SQL MANIPULATION



user_id	movie_id	ACTION	ADVENTURE	COMEDY	HORROR	CHILDREN	CRIME	DOCUMENTARY	DRAMA	FANTASY	FILM-NOIR	MUSICAL	MYSTERY
1997	1	0	0	1	0	1	0	0	0	0	0	0	0
1997	1265	0	0	1	0	0	0	0	0	0	0	0	0

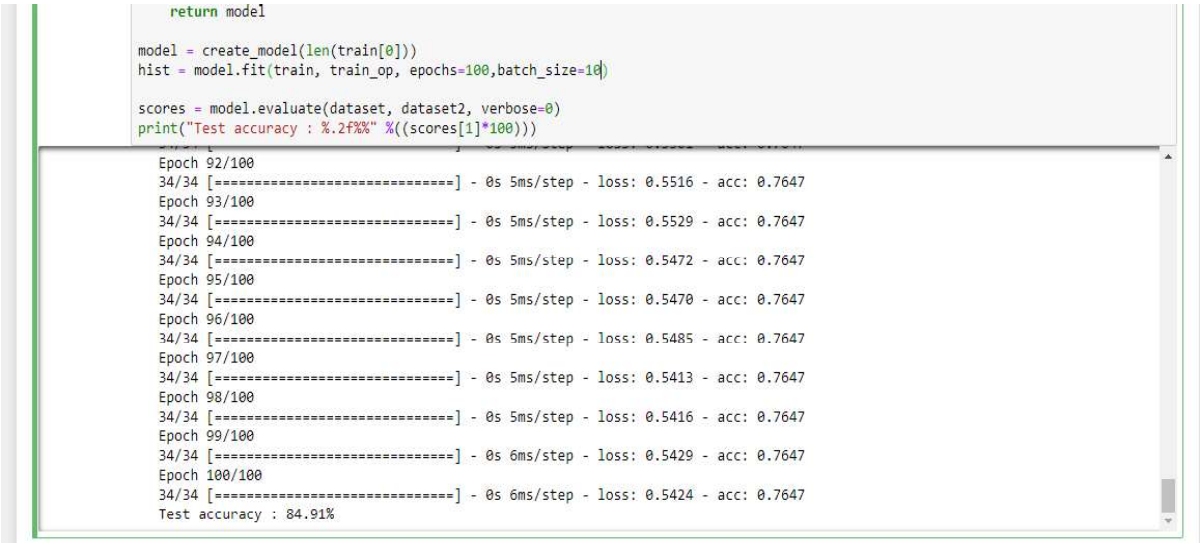
AUTOENCODER BASED RECOMMENDER OUTPUT



```
print('Epoch', epoch, '/', hm_epochs, 'loss:',epoch_loss)

MSE train 57.70589462886741 MSE test 57.99464622328289
Epoch 0 / 200 loss: 3966.329334259033
MSE train 38.04793482187772 MSE test 38.18121606613857
Epoch 1 / 200 loss: 2265.6043014526367
MSE train 27.35864053066486 MSE test 27.517741874885832
Epoch 2 / 200 loss: 1560.1739902496338
MSE train 21.35755986176045 MSE test 21.56659312060692
Epoch 3 / 200 loss: 1168.436803817749
MSE train 17.62925489109174 MSE test 17.87601660450571
Epoch 4 / 200 loss: 940.1502532958984
MSE train 15.105641225034647 MSE test 15.317726342225757
Epoch 5 / 200 loss: 790.7911939620972
MSE train 13.352840468383492 MSE test 13.550743621901116
Epoch 6 / 200 loss: 688.9431276321411
MSE train 11.981539847238224 MSE test 12.18787262964488
Epoch 7 / 200 loss: 614.039101600647
MSE train 10.912621198707308 MSE test 11.131543910524886
Epoch 8 / 200 loss: 554.7536573410034
MSE train 10.050076362360121 MSE test 10.301115221972
Epoch 9 / 200 loss: 507.89521408081055
```

LSTM BASED RECOMMENDER OUTPUT



```
    return model

model = create_model(len(train[0]))
hist = model.fit(train, train_op, epochs=100,batch_size=10)

scores = model.evaluate(dataset, dataset2, verbose=0)
print("Test accuracy : %.2f%%" %((scores[1]*100)))
```

```
Epoch 92/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5516 - acc: 0.7647
Epoch 93/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5529 - acc: 0.7647
Epoch 94/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5472 - acc: 0.7647
Epoch 95/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5470 - acc: 0.7647
Epoch 96/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5485 - acc: 0.7647
Epoch 97/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5413 - acc: 0.7647
Epoch 98/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5416 - acc: 0.7647
Epoch 99/100
34/34 [==============================] - 0s 6ms/step - loss: 0.5429 - acc: 0.7647
Epoch 100/100
34/34 [==============================] - 0s 6ms/step - loss: 0.5424 - acc: 0.7647
Test accuracy : 84.91%
```

MODIFIED MOVIELENS DATASET

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	1193	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
1	661	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0
1	914	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
1	3408	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
1	2355	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1
1	1197	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	1287	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
1	2804	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
1	594	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1
1	919	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1
1	595	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1
1	938	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
1	2398	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
1	2918	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1035	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
1	2791	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	2687	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	2018	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1
1	3105	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
1	2797	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
1	2321	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	720	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	1270	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
1	527	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1

MODIFIED LSTM RECOMMENDER OUTPUT

```
scores = model.evaluate(dataset, dataset2, verbose=0)
print("Test accuracy : %.2f%%" %((scores[1]*100)))
```

```
Epoch 92/100
34/34 [==============================] - 0s 4ms/step - loss: 0.5862 - acc: 0.7647
Epoch 93/100
34/34 [==============================] - 0s 4ms/step - loss: 0.5523 - acc: 0.7647
Epoch 94/100
34/34 [==============================] - 0s 4ms/step - loss: 0.4902 - acc: 0.7647
Epoch 95/100
34/34 [==============================] - 0s 4ms/step - loss: 0.5277 - acc: 0.7647
Epoch 96/100
34/34 [==============================] - 0s 4ms/step - loss: 0.5062 - acc: 0.7647
Epoch 97/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5346 - acc: 0.7647
Epoch 98/100
34/34 [==============================] - 0s 5ms/step - loss: 0.5327 - acc: 0.7647
Epoch 99/100
34/34 [==============================] - 0s 4ms/step - loss: 0.4989 - acc: 0.7647
Epoch 100/100
34/34 [==============================] - 0s 5ms/step - loss: 0.6075 - acc: 0.7647
Test accuracy : 86.91%
```

INPUT VECTOR FORM

```
print(test)
print(train_op)
print(test_op)
```

```
34 19
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

RBM BASED RECOMMENDER OUTPUT

```
plt.plot(errors)
plt.ylabel('Error Rate')
plt.xlabel('Number of Epochs')
plt.show()
```

```
0.027124168
0.021825418
0.020945786
0.020423517
0.020211646
0.020089017
0.020006616
0.01990658
0.019823827
0.019748472
```