**Table of Contents**

|  |  |  |
| --- | --- | --- |
| 1. | Introduction | 5 |
| 2. | Requirement Specifications | 5 |
| 3.  4. | Datasets  Technique | 5  5 |
| 5. | Algorithm | 6 |
| 6. | Training algorithm | 6 |
| 7. | Output | 9 |
| 8. | Conclusion | 11 |
|  |  |  |
|  |  |  |

**Introduction**

The aim of this project is to build a system that recommends the top 10 movies that are similar to movies watched by a user and are highly rated by other users as well.

**REQUIREMENT SPECIFICATIONS**

**Hardware**

RAM: 8GB

Hard Disk: 500 GB

Processor: Intel core

**Software**

• Operating System: Windows 11

• Anaconda, Jupyter notebook

**Dataset**

Our dataset (ml-latest-small) describes 5-star rating and free-text tagging activity from Movie Lens, a movie recommendation service. It contains 100836 ratings and 3683 tag applications across 9742 movies. These data were created by 610 users between March 29, 1996, and September 24, 2018. This dataset was generated on September 26, 2018.

**Techniques**

We have employed use of Word2Vec technique to train the model. Instead of using words and creating word embeddings, we used Movies and created movie embeddings vector and trained it over. So, for two movies to be similar to each other, their movie embedding vector will essentially be similar.

**Algorithm**

We are using word2vec algorithm, **Word2vec** is a technique natural processing language published in 2013. The word2vec algorithm uses a **neural network** model to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. As the name implies, word2vec represents each distinct word with a particular list of numbers called a vector. The vectors are chosen carefully such that a simple mathematical function (the cosine similarity between the vectors) indicates the level of semantic similarity between the words represented by those vectors.

**Training Algorithm**

First, we are classifying a particular user like the movie if his /her rating is greater than 4 then we are denoting by 1 and if it is less than 4 then it is denoted by 0. Then we are dropping the rating table. We are creating a pivot using userId and movieId of the rating table by matrix created. converting rating\_co\_occ to matrix with diagonal elements as 0 to help in the loss function. we converted matrix to tensor using a torch

Then we created a reference table with movieId, indexes of movies, movie name, and genre. After that, we are doing word embedding by One Hot Encoder matrix. We are taking input OHE to NN to module movies\_tensor.requires\_grad. We created a model to calculate loss function and optimize using gradient descent where N is batch size D\_in is input dimension, H is a hidden dimension, D\_out is output dimension. Then we create random Tensors to hold inputs and outputs. Use the nn package to define our model as a sequence of layers. nn.Sequential is a Module that contains other Modules, and applies them in sequence to produce its output. Each Linear Module computes output from the input using a linear function and holds internal Tensors for its weight and bias. The nn package also contains definitions of popular loss functions, In case we will use Mean Squared Error (MSE) as our loss function. we are using Forward pass to compute predicted y by passing x to the model.

Before the backward pass, use the optimizer object to zero all of the gradients for the variables it will update (which are the learnable weights of the model). This is because by default, gradients are accumulated in buffers whenever backward called.Backward pass is to compute the gradient of the loss with respect to model parameters.

**Adam Formula**

**m = beta1\*m + (1-beta1)\*dx**

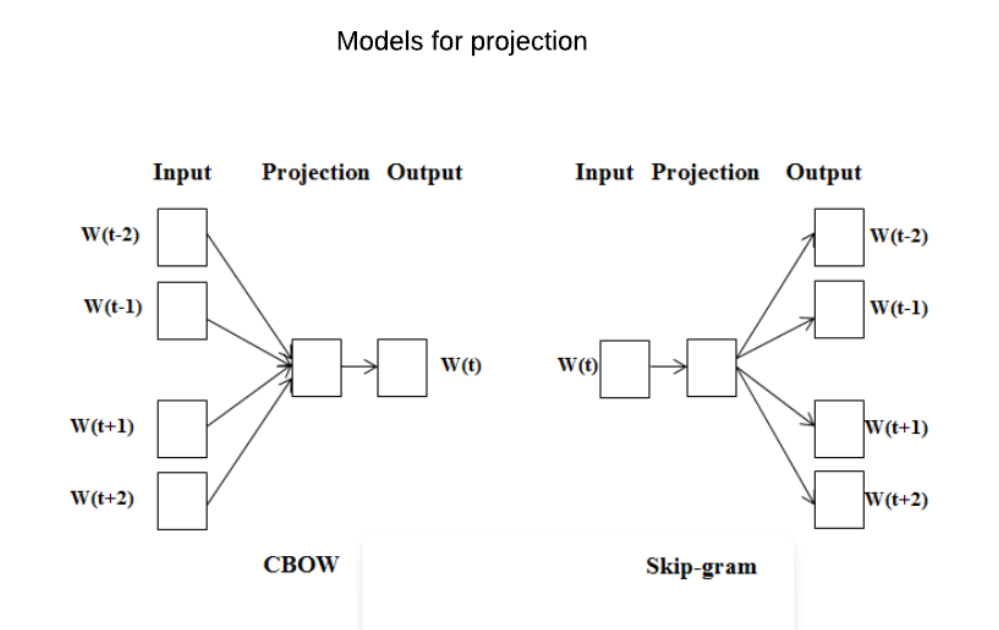
**mt = m / (1-beta1\*\*t)**

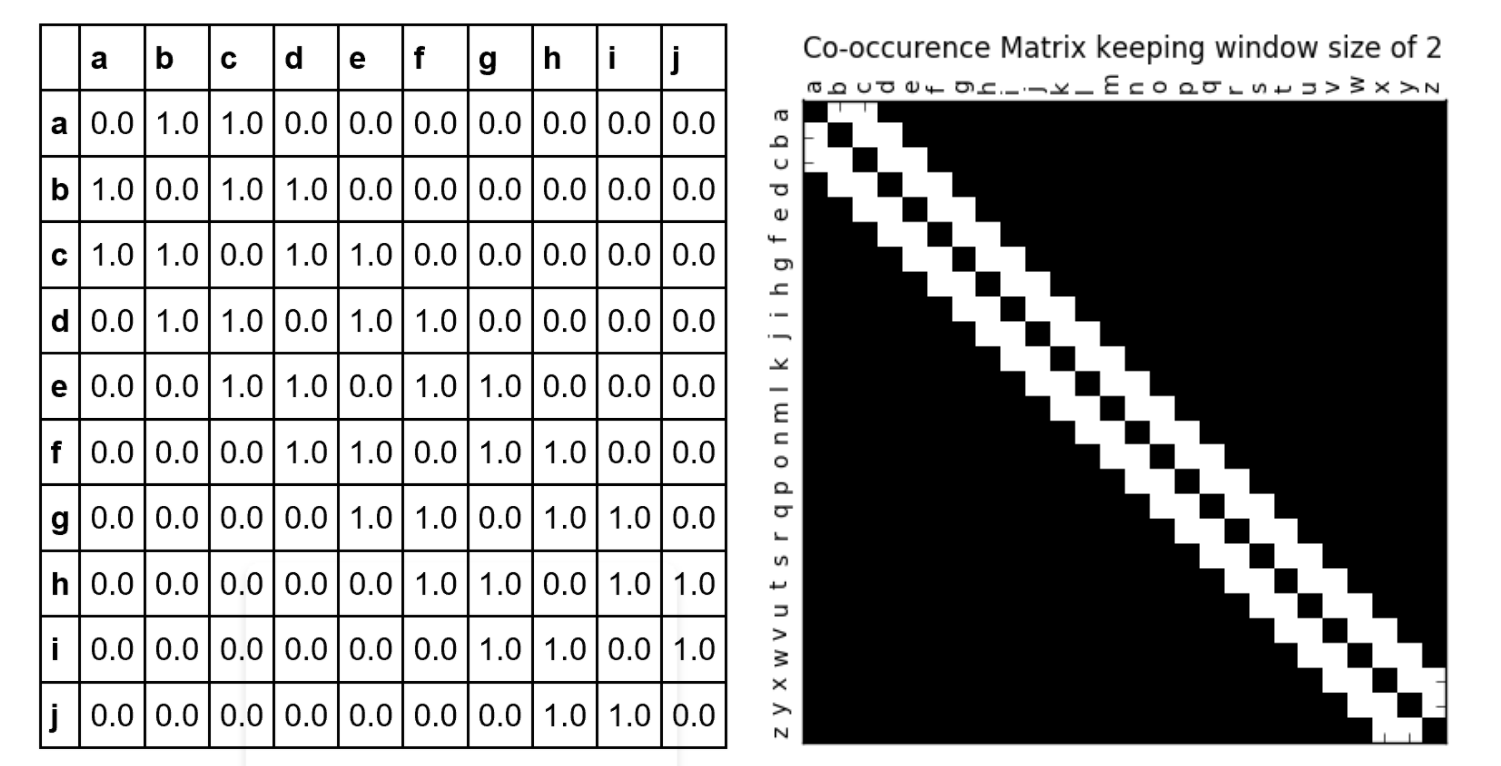
**v = beta2\*v + (1-beta2)\*(dx\*\*2)**

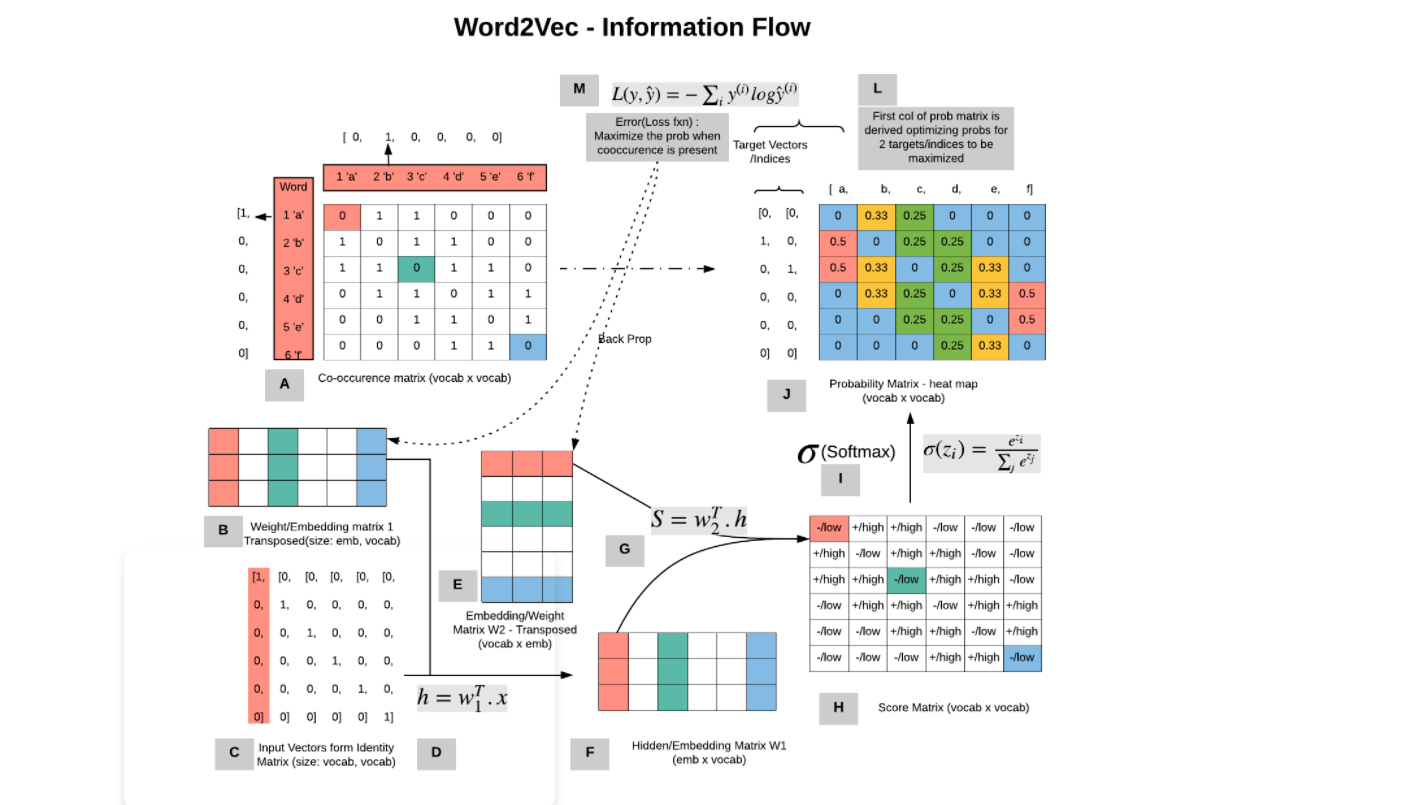
**vt = v / (1-beta2\*\*t)**

**x += - learning\_rate \* mt / (np.sqrt(vt) + eps)**

We are Calling the step function on an Optimizer makes an update to its parameters. We have got a Loss of cost function after 200 iterations. we have plotted loss vs interactions.Then we have recommended top 10 movies.We have created a prediction function to give movie recommendations to getting tensor for movie and storing weight vector from model, movie index value sorted from min to max for predicted score values. we are Getting title of the recommended movie from Reference Dataframe we are getting the Index of Movie from MovieId, Sorting the top 10 movie indexes from Co-Occurrence Matrix, and Getting the movie title from the top 10 movie indexes. We are getting top 10 movie predictions for Apollo 13. Before the backward pass, use the optimizer object to zero all of the gradients for the variables it will update (which are the learnable weights of the model). This is because by default, gradients are accumulated in buffers wheneverbackwardis called. The predictions for Apollo 13 will show that these are much different than the actual predictions and also from the one we have recommended when the learning rate is 0.01.

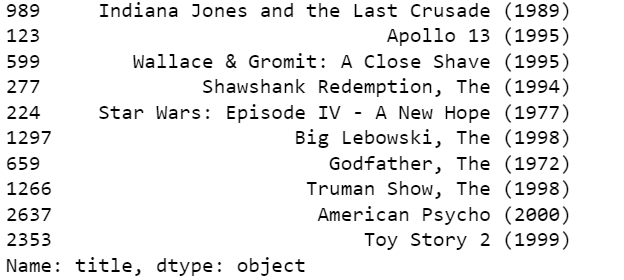






**Output**

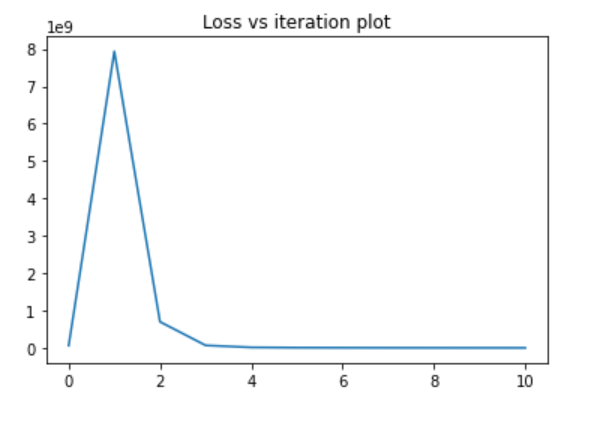
**The predictions for Apollo 13 will show that these are much different than the actual predictions and also from the one we have recommended when the learning rate is 0.01**

****

**Plot of loss vs interations**



**Plot of loss vs interations**



**Conclusion**

learning rate scales the magnitude of our weights in order to minimize the network's loss function. If learning rate is set to a high value it would keep bouncing as it nears the optimal point and may not reach the optimal point and thus give us incorrect recommendations. Whereas, if the learning rate is too small, then training will progress very slowly as we are making very tiny updates to the weight and there is a chance to get stuck in local minima and our gradient descent function will give constant loss with an increase in iterations, resulting in high cost and incorrect recommendations.