Application of Social Media in the Domain of Recommender Systems

Project report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

In

Electronics and Communication Engineering

Submitted by

Pradeep Kumar G (1810110166)

Under supervision of

Prof. Madan Gopal
(Professor, Department of Electrical Engineering)

Department of Electrical Engineering
School of Engineering
Shiv Nadar University Delhi NCR
(Spring 2022)

Candidate Declaration

I hereby declare that the thesis entitled “Application of social media in the domain of recommender systems” submitted for the B Tech Degree program. This thesis has written in my/our own words. I have adequately cited and referenced the original sources.

(Signature)
Pradeep Kumar G
(1810110166)
Date: 28/04/2022

CERTIFICATE

It is certified that the work contained in the project report titled “Application of social media in the domain of recommender systems,” by “Pradeep Kumar G” have been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Signature)
Prof. Madan Gopal
Dept. of Electrical Engineering
School of Engineering
Shiv Nadar University Delhi NCR
Date: 28/04/2022

Abstract

Recommender systems play an important role in helping users find relevant information by suggesting information of potential interest in them. They are applied in variety of fields from recommending movies and songs to online shopping, where items are recommended based on the user's past activity or similarity with other users. The increasing popularity of social media has produced rich information about a user, social data include information that social media users publicly share, which includes user's likes, dislikes, opinions about a specific topic, discussions and debates on topics of social relevance. In this project we build a recommender system that utilizes this social media data to provide users with relevant content.

Table of Contents

List of figure …….. **8**

List of Tables …….. **11**

1. Introduction …….. **12**
    a. Motivation …… **12**
    b. Problem Statement …… **12**
2. Literature Review ……..…… **14**
    a. Collaborative Filtering…..……..... **14**
    b. Memory Based Algorithm ……... **14**
    c. Model Based Algorithms…….. **15**
    d. Content Based Filtering ..…….. **15**
    e. Neural Network.. ……... **15**
    f. Neural Graph Algorithms……... **16**
3. Work done ……..…… **20**
    a. Recommendation System ..……... **20**
        i. Content Based Filtering ……..... **20**
        ii. Collaborative Filtering…….... **21**
        iii. Hybrid Recommender Systems ……... **22**
    b. Data ……...……... **23**
        i. Explicit Data…….…..... **24**
        ii. Implicit Data……..…….... **24**
    c. Social Data ……...……... **25**

i. Data Collection……..…….... **25**

ii. Data Preprocessing...…..……..... **26**

iii. Aspect Based Sentiment Analysis...…..…….……..... **27**

d. Book Recommendation System…….... **28**

i. Goodreads Book Dataset…….... **29**

ii. Data Description…….…….... **29**

iii. Data Preparation…….…….... **30**

iv. Graph Neural Networks…….... **32**

1. Graphs…….…….... **32**

2. Graph Tasks…….…….... **34**

3. Graph Neural Networks..…….…….... **35**

4. GraphSAGE……..…….,…..... **35**

5. Graph Preparation..…….... **38**

v. Model Architecture…….... **39**

1. HinSAGE…….…….. **41**

2. Link Regression Task…….…….... **44**

3. Loss Function…..…….…….... **45**

4. Model Summary…..…….……..... **45**

5. Training…..…….……..... **48**

6. Recommendations…..…….……..... **50**

vi. Results…..…….…….…..... **51**

vii. Implementation ….…….. **54**

4. Conclusion……... ..…....…… . **69**

5. Future Prospects……... ..…....…… . **70**

6. References ……. **71**

List of Figures

2.1 Neural Collaborative filtering ……..…….. **16**

2.2 Neural Graph Collaborative Filtering..…….. **17**

3.1 Summary of content-based filtering models …….. **21**

3.2 Principle behind collaborative filtering models. …….……... **22**

3.3 Principle behind hybrid recommender system …….. **23**

3.4 Data Preprocessing Steps …….…….. **26**

3.5 Average rating and average rating per genre per added to enhance user features……...…….. **31**

3.6 A simple graph with nodes and edges...…….. **32**

3.7 Different type of graphs……...…….. **33**

3.8 Special graphs.……...…….. **33**

3.9 Pseudo code of GraphSAGE algorithm.…….. **36**

3.10 Nodes are books and users, edges only exist between a user and a book, edges have weights and that being the rating a user has given to the book .....…….. **38**

3.11 User-Item graph.……....…….…….. **39**

3.12 Model Flowchart.……..…...…….. **40**

3.13 Graph with nodes and feature vectors..…….. **42**

3.14 Neighbourhood Aggregation..……..…….. **43**

3.15 HinSAGE Model..……..……..…….. **44**

3.16 Inputs are reshaped according to the needs of the model, done by stellargraph……...……..…….. **46**

3.17 The inputs or the neighbors are fed to the mean aggregator function to calculate representations or embeddings of nodes …..…….…….. **47**

3.18 The embeddings are concatenated in the link embedding function, and passed to a neural network to predict the ratings.……..……..…….. **48**

3.19 The training loss of the model……..……..…….. **49**

3.20 Recommendation Flowchart..……..……..…….. **50**

3.21 Predicted vs Actual Rating.……..……..…….. **52**

3.22 User's read history……..……..……..…….. **53**

3.23 Recommended books for the user..…..……..…….. **53**

3.24 Code to download dataset..……..……..…….. **54**

3.25 Code to convert file format……..……..…….. **54**

3.26 Code to observe data…….……..……..…… **55**

3.27 Keeping useful columns…..……..……..…….. **55**

3.28 Keeping useful columns in ratings data……..…….. **55**

3.29 Book genres are found out from another dataset provided by UCSD…… **56**

3.30 Code to find genres..……..……..……. **56**

3.31 Code to merge genres with meta-data of book..…….. **57**

3.32 Cleaning the ratings data……..……..…….. **57**

3.33 Ratings and meta-data are merged to create helpful data for creating user features…….. **58**

3.34 Reduce rows of user……..…….. **58**

3.35 Code to standardize the data……..……..…… **59**

3.36 Code to remove non-numeric column……..…….. **59**

3.37 Book data…… **59**

3.38 Ratings data…… **60**

3.39 User data…… **60**

3.40 Code to add user nodes …… **60**

3.41 Code to add book nodes to the graph…… **61**

3.42 Code to add edges to the graph…… **61**

3.43 Code to add node types…… **62**

3.44 Code to observe number of nodes and edges..…… **62**

3.45 Code to add features to graph..…… **63**

3.46 Code to create subgraphs..…… **64**

3.47 Properties of graph created by stellargraph …… **64**

3.48 Edges split into test train and validation sets.…… **65**

3.49 Code to generate data from graph to mode…… **65**

3.50 HinSAGE model …… **66**

3.51 Model Training…… **66**

3.52 Code to retrieve ideal candidates…… **67**

3.53 Code to filter recommendation…… **67**

3.54 Code for generating recommendations.…… **68**

List of Tables

2.1 Different recommendation system solutions…….. **19**

3.1 Twitter data features…….…….. **25**

3.2 Meta-Data Of books…….…….. **29**

3.3 User-Books interactions data ...…….. **30**

3.4 Genres of books added as features. ..…….. **31**

3.4 Hyperparameters……. ..…….. **52**

Chapter 1
Introduction

Recommendation system are used to recommend relevant items to users. Their use cases are vast they could be used to recommend movies to music, e-commerce products and advertisements as well. Popular recommendation algorithms include recommending items based on similarity of users, similarity of items or a combination of both.

Motivation:

Recommender system has the ability to predict whether a particular user would prefer an item or not based on the user's profile. Recommender systems are beneficial to both service providers and users. They reduce transaction costs of finding and selecting items in an online shopping environment.

Due to drastic rise in accessibility of internet, the number of social media users have increased to over 3.6 Billion in recent years and the number of active users of Facebook and WhatsApp recently crossed 2 billion users. Data collected from these social media sites have vast information stored in them. Some of the common information that can be inferred are the user's likes, dislikes, opinions about a certain topic, location, designation, followings and more. Apart from the users' information we can also mine information about items or products, user opinions about a certain item, item trends etc.

Hence using social media data in the task of recommendation systems could enhance the recommendations.

Problem Statement:

Due to the rapid growth of the internet in conjunction with the information overload problem the use of recommender systems has started to become necessary for both e-businesses and customers. In recent years graph neural networks have been extensively researched, they are capable of capturing unstructured data which many conventional techniques fail to do so. Given the

unstructured nature of social data, they could be used well with graph neural networks to recommend items to users.

Hence in this project we aim to recommend items to users with the help of social data and graph neural networks. The objective are to collect data from social media sites, find meaningful data from it and to represent the social data into a suitable graph and develop a graph neural network model to learn underlying user-item interactions and recommend relevant items to users.

Chapter 2
Literature Review

In this section we look at various implementations and methods proposed by different authors to tackle the problem of recommendation system and social media.

Recommendation algorithms can be divided into collaborative filtering and content-based filtering algorithms.

Collaborative filtering algorithms use user's interaction matrix with the items to construct a user preference vector and predicts items that fits user's tastes and later use it for recommendation [1]. This can be further divided into memory based and model based collaborative filtering algorithms.

Memory based algorithms:

Memory based algorithms can be further divided into user-user and item-item algorithms. In user-user based algorithm recommends item to user if similar users have liked this item before. Similarities here are computed based on rating of items between the users. In item-item algorithm, an item is recommended based on whether the item is similar to the item previously liked by the user [2]. It finds similarity based on number of users common between them. Algorithms like KNN, vector cosine correlation and Pearson correlation are usually used to create similar groups and recommend items to users within the same group.

These algorithms suffer from: *cold-start* problem, that is when a new user enters and no data is available, *scalability*, adding a new user or item would require recomputing all similarities of users and items and is computationally expensive, *data sparsity* is a big issue faced by recommendation models including this, users only interact with very few items and hence don't have enough data to recommend, and *gray sheep* problem, this occurs when the set of users whose preference is similar to the target user is quite low.

Model based algorithms:

Model based algorithms estimates or learns a model based on user-item ratings and tries to predict user's rating. One of the most popular methods is the *matrix factorization* algorithm. The goal of a matrix factorization algorithm is to learn low-dimensional vectors or embeddings for all users and items such that multiplying them gives the rating the user would give to an item.

Mathematically the task is to decompose the ratings matrix into a dot product of two matrices.

$$R_{m\times n} \approx P_{m\times k} \times Q^T_{n\times k}$$

Here *R* represents the rating matrix which is decomposed into *P* and *Q* matrix. *P* represents the user matrix where the rows represent the users and column represent *k* latent factors. Similarly, *Q* represents the item matrix where the rows represent the *k* latent factors and columns represent the number of items [3]. These latent factors represent each user and item in a vector of length *k*, they are also called embeddings and represent the characteristics of a user or item. The task is to find best latent factors to factorize the rating matrix. This is done by minimizing the root mean squared error using techniques like ALS or SGD.

After the best latent factors are found the dot product of these user and item matrices are used to fill the ratings matrix and recommend the best items. This still suffers from data sparsity issue.

Content based filtering is a method to recommend similar items based on information or meta-data of items the user has interacted with in the past. Hence it doesn't require other user's data and can recommend items based only on user's history. It uses techniques like cosine similarity, Naïve Bayes, TF-IDF to mine and identify user preferences. But it suffers from cold-star problem and serendipity problem. Serendipity refers to recommending new or diverse items to users.

Neural networks have been recently used in recommendation systems. One such popular method is the neural collaborative filtering (NCF) method [4]. Matrix factorization (MF) predicts ratings based on the dot product of latent factors, which is quite simple and may not capture complex user interaction data. NCF utilizes a multi-layer perceptron to learn user-item interactions. It uses one-

hot encoded user and item vectors as inputs, which are then mapped to a hidden embedding layer. Which are then concatenated and passed on to MLP layers. NCF uses the MLP layers to capture the non-linearities from the user interactions, it also uses the input vectors for regular matrix factorization and concatenates the outputs and predicts the rating. The idea is to capture both the linearity and non-linearity of the user-interaction matrix.

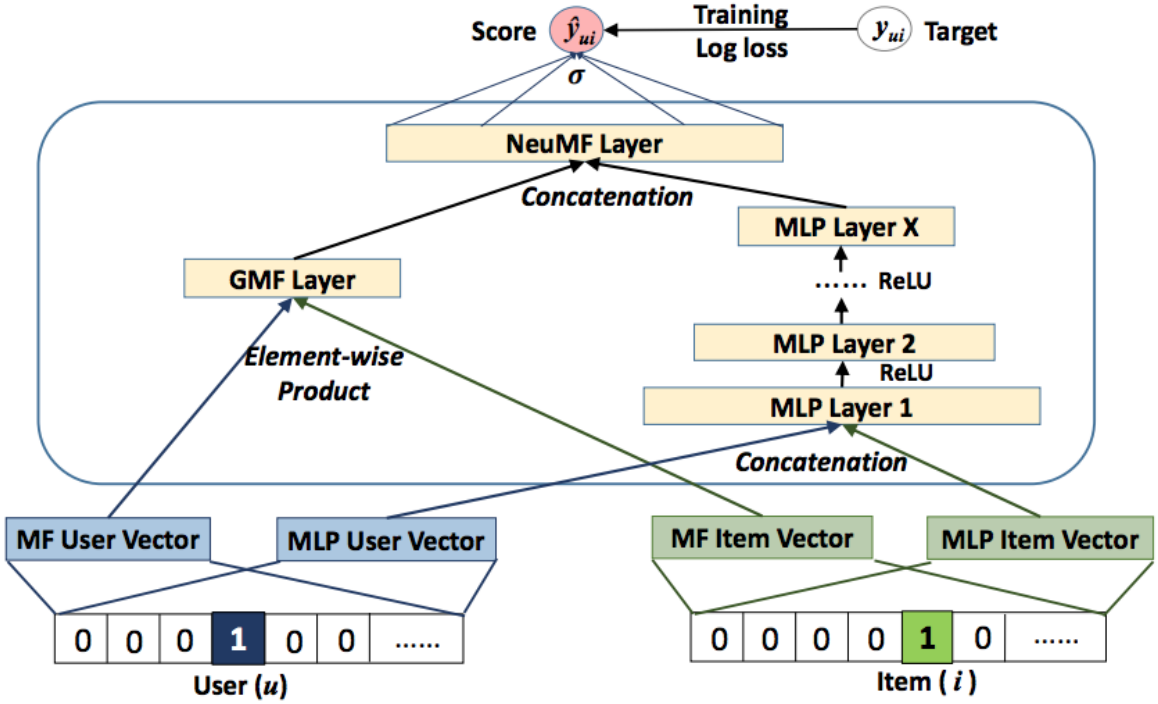

Figure 2.1 Neural Collaborative filtering: User and item embeddings are passed to MLP layer and general Matrix Factorization layer and concatenated to predict user ratings.

Neural graph algorithms:

Recently, graph neural networks are increasingly researched on recommender systems, one of the reasons is because the data collected from online platforms have many forms which include user-item interaction, user profile, item profile, user's social networks and much more [5]. Traditional approaches are unable to capture these multiple forms of data. Graphs represents this data in the form of nodes and edges and are able to learn strong representations.

The main steps involved in GNNs are Message passing or propagation of information, aggregation which is a function to aggregate information received and iteration that determines the extent of the message [6].

Popular models in GNNs include Neural graph collaborative filtering (NGCF) which considers the user-item interaction matrix as bipartite graph where users and items are nodes and positive interactions are the edges [7].

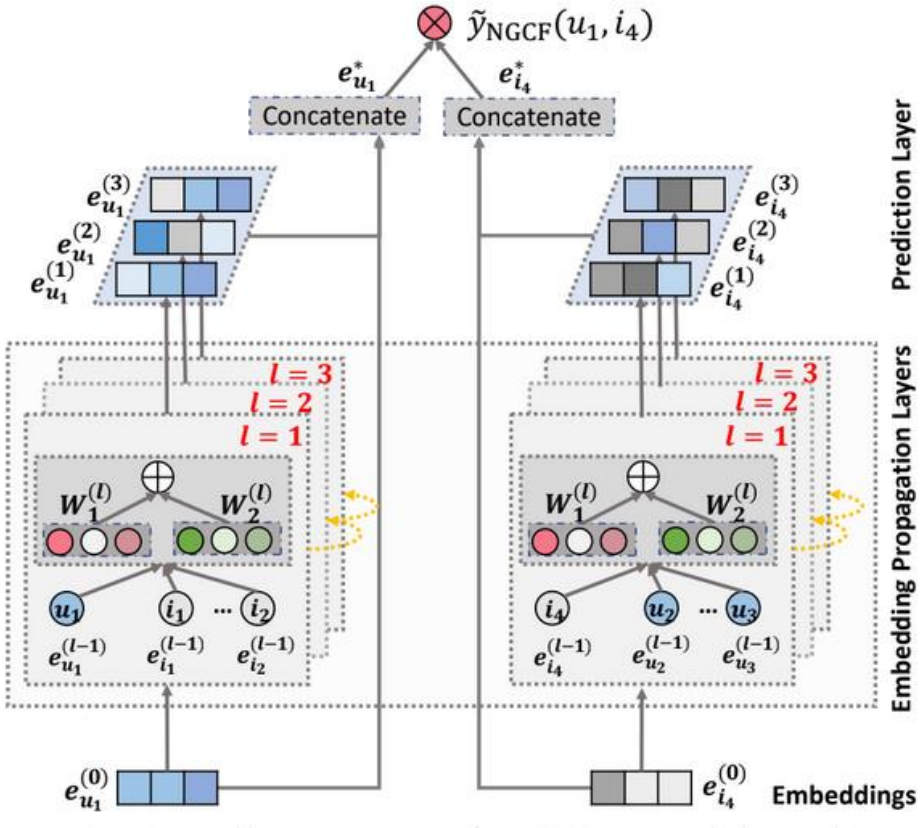

Figure 2.2 Neural Graph Collaborative Filtering, user and item embeddings are created from each embedding propagation layer and concatenated. Rating scores are calculated using dot product of the user and item embedding.

NGCF's initial embedding layer initializes user and item embeddings, which are passed to the propagation layer, where messages are passed from one node to another, they update the nodes with information from neighbour nodes. Multiple layers could be stacked for gathering higher order representations. For the final prediction layer all the embeddings are concatenated for each user and item. Where user's preference over an item is calculated by

$$\hat{y}_{NGCF}(u,i) = e_u^{*T} e_i^*$$

Where e_u and e_i represent embeddings of user and item respectively.

Decathlon Canada, used GNNs to recommend products to their users [8]. Their graph consisted of 3 types of nodes and 5 types of edges. GNNs are used to generate embeddings here. Nodes were updated with their neighbors' message using mean aggregation, learnable weight matrices are then multiplied to it and updated it with initial node message which has been multiplied with its own weight matrix. Which are then summed and passed through non-linearity. Adding more layers causes deeper neighbors to be included.

$$h_u^k = a(W_{self}^k + W_{neigh}^k \Sigma_{v \in N(u)} h_v^{k-1})$$

Where h^k_u is the embedding of node *u* at layer *k*, *v* represents neighbors of *u* in the neighborhood *N(u)*, and *W* are learnable parameters.

The embeddings of user *u* and item *v* are then scored using the cosine similarity function and highest scored items are recommended.

A comparison of various algorithms is tabulated below.

Method	Technique	Advantages	Disadvantages
Content Based	Similarity score of meta-data /mined text with user's history.	Independent of other users, Explainable Recommendations	User cold start problem, no diverse recommendations (no serendipity), Domain knowledge required.
Memory Based	Similar users / items group are found and most popular items are recommended.	Scalable, Diverse recommendations (user-user).	Gray sheep, cold-start problem, user data sparsity, computationally expensive
Matrix Factorization	User-Item interactions matrix completion using latent factors multiplication.	Diverse recommendations, Domain knowledge not required.	Scalability – \ computationally expensive, data sparsity
NCF	User-item interactions learnt using neural networks	Captures complex relationships compared to MF.	Doesn't perform better than MF
GNNs	Node / Edge embeddings are learnt and edge ratings are calculated.	Explainable recommendations, Useful with un-structured data.	High training time, complex graph modelling

Table 2.1 Different recommendation system solutions

Chapter 3
Work Done

In this section we describe the work done over this semester. In this section we give a brief on theory involved in creating a recommendation model and then proceed to report what has been done in this project. After the basic theory we focus on two subsections namely social media data which describes the social media data processing, recommendation system subsection which describes the book recommendation task on goodreads.com's book dataset.

Recommendation System:

Recommendation system is the task of recommending relevant items to users. Examples include, recommending music based on user's listening history, recommending news articles based on user's preferred topics, recommending add-ons while making purchase on e-commerce websites, recommending workouts based on user's session data etc. Recommendation systems are classified into 3 types Collaborative Filtering, Content-based Filtering and Hybrid Recommendation systems. We will look briefly into this in the following sections.

Content-based filtering:

Content-Based Filtering is a method for recommending items with attributes similar to those that users like, and recommends them based on the information of the items. Or, in simpler terms it is a task of recommending similar items based on information on items selected by the user in the past. Content-based filtering models are the most basic recommendation models and were popular in the early years of recommendation systems.

Content-based filtering models recommends items that are closely related to items previously liked by the user, so it has limitations as it is unable to recommend new items to users. Hence, they were mainly used in services that recommend items or text data items that are easy to recommend based on item information and user-profile data. Some practical examples include recommending similar

music based on meta-data information. The interest in content-based filtering models alone has decreased but hybrid recommenders use content-based filtering techniques to enhance or add user/item preferences.

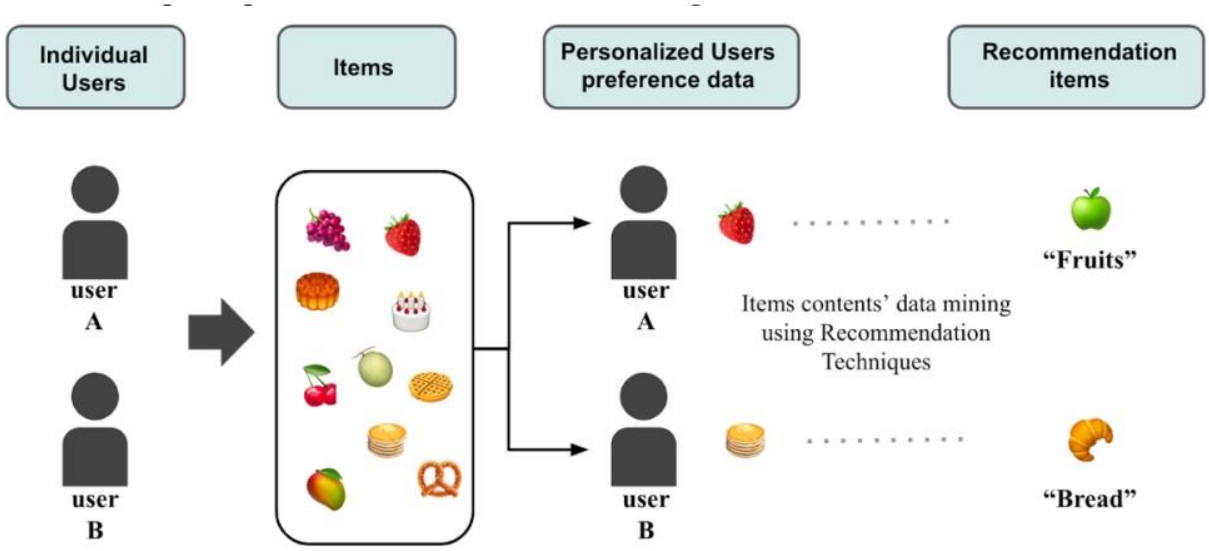

Figure 3.1 Summary of content-based filtering models

Collaborative Filtering:

Collaborative filtering models work on users' ratings or interactions matrix. Collaborative filtering models are one of the most widely used models and they construct user's preference data using the user's ratings data to predict items that fit the user's preference, and then rank them for giving recommendations. Their classification was already discussed in the literature review section.

Collaborative filtering models even though adopted well enough face scalability and sparsity problem and hence the accuracy of the recommendations is lowered. To counter these issues hybrid models have been used in recent times.

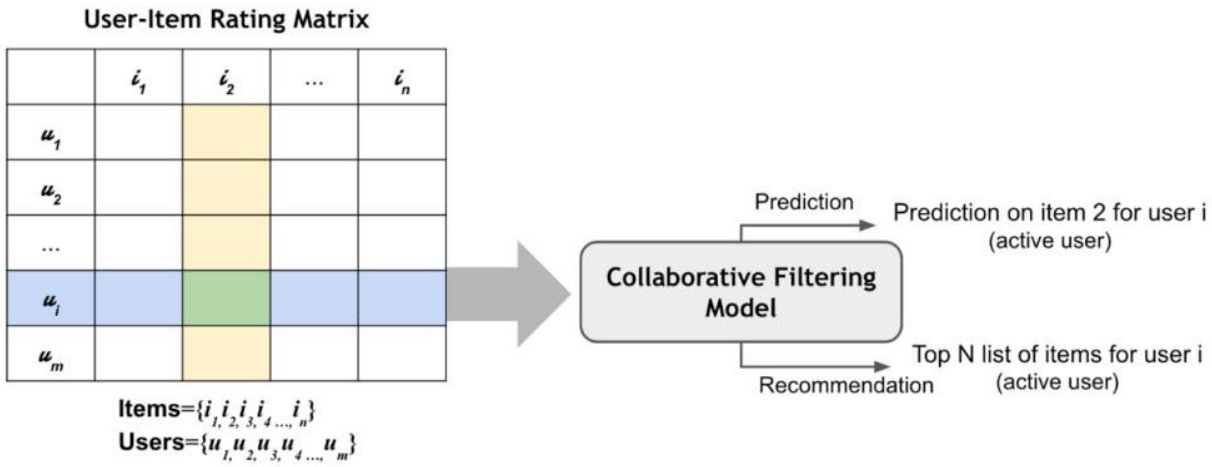

Figure 3.2 Principle behind collaborative filtering models.

Hybrid Recommender Systems:

The main limitation of content-based filtering models are that they rely extensively on meta-data of the user's item while collaborative model rely on user's item rating/interactions data. The main idea behind implementing hybrid models is to solve the sparsity problem, most models try to compensate the lack of rating data by integrating information from content-based filtering models and collaborative filtering models. One popular method is to integrate various side information to explicitly rated data. This side information includes social connections between users and items information data [14].

Hybrid systems also include using multiple models and passing scores between each other, one such method is the feature augmentation hybrid method where on model is used to classify and item's preference score or item and the information generated is passed to the next recommendation model [15].

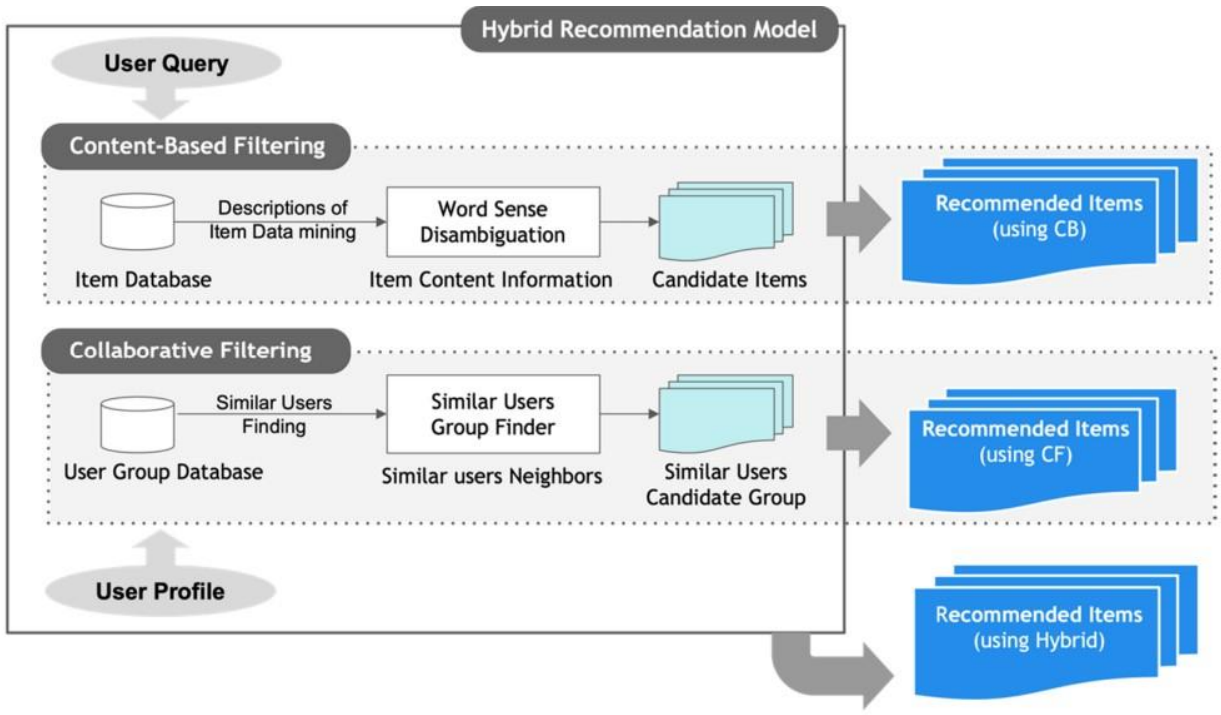

Figure 3.3 Principle behind hybrid recommender system

Recommendation systems rely on various forms of data for recommending items, in the next section we look into different kinds of data recommendation systems use.

Data:

Recommendation systems use data to recommend relevant user or items. This could be data containing users and items interactions, whether users have purchased, liked, rated a item or it could be data of only items where items and their various features are present, similarly, data of only users and only their features could also be present.

Hence data collection is arguably the most important factor for building or deciding a recommender system. There are mainly two ways of collecting data for recommendations implicit and explicit.

Explicit Data:

It is the explicitly available or user defined data. This includes the ratings given to a movie by the user, or rating of products given by the users on e-commerce sites etc. The likeness or dis-likeness can be easily inferable based on the score or the rating given by the user. Some issues faced in explicit data are:

1. Explicit data require additional input from the user and a user rarely takes their time to give their feedback.
2. The way users give their ratings are different from each other, some may rate only items they like and that too highly while some others might be very critical of the way they rate items which cannot be inferred from explicit data.

Implicit Data:

It is an inferred form of data which means it tries to collect information based on user's actions or behaviour. Examples include browsing history, clicks on links, count of the number of times a song is played, the percentage of a webpage user has scrolled, mouse movements etc. These are data that do not require any extra input from user and are quite abundant given the rise of social media. These kinds of data have some well-known issues some being:

1. Negative interests are difficult to measure unlike explicit ratings where user's dislikes can be found easily by their low score of ratings, there are no direct methods to do so for implicit data.
2. Noisy data, when collecting implicit data users' actions could be misrepresented or might mean something else entirely. Example if a user purchases a product for another, it could be misrepresented as user liking the product.

Social media has vast amounts of user and item data, and implicit data could be mined to construct user or item preferences from social media sites. In the next section we describe how to collect such data from a popular social media site called Twitter.

Social Data:

We focus on the popular social media site twitter where users share their thought, interact with other users etc. via tweets. In the following sections we describe the data collection module, data pre-processing module and aspect-based sentiment analysis module of the social media data collection pipeline.

Data Collection:

A public library called *snscrape* was used to scrape twitter data. It is legal to scrape data from twitter but exploiting their API isn't, *snscrape* doesn't use Twitter API to get their data hence doesn't break their terms and conditions. The steps for data collection are described as follows:

1. **Scraping:** A list of users were chosen to scrape data from, a script ``*snscrape --jsonl --max-results \{2\} twitter-user \{0\} \> \{1\}".format(username, savename, limit)* was used to pass the list of users for scraping most recent 500 tweets. The script returned the user's tweets, mentioned users, retweets, replies, location, hashtags etc. in json format.
2. **Cleaning**: After scraping the data needs to be in relevant format for further preprocessing and in this step, we format the retrieved data for further use. We keep only relevant data Table 3.1 shows the kept data. The data is converted to a data table from the json format using the *pandas* library.

Features	Value
Username	ChrisMedlandF1
Tweet	"Hello"
Hashtag	F1
Description	"F1 Journalist"

Table 3.1 Twitter data features

The data is then stored in a comma separated value (csv) file for easier use. A sample user's twitter data retrieved is shown in Table 3.1.

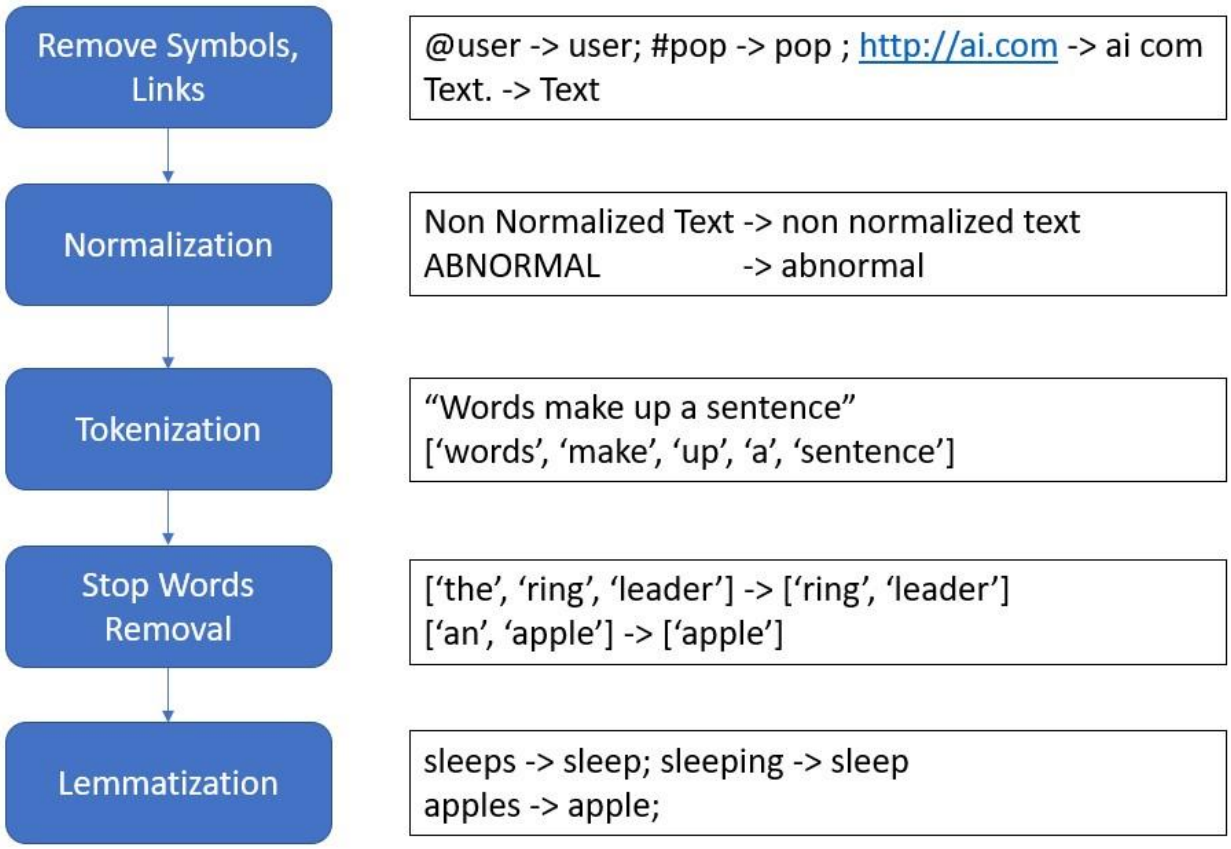

Figure 3.4 Data Preprocessing Steps

Data Preprocessing:

Various preprocessing steps can be summarized as follows:

1. **Cleaning:** As the raw text data contains various symbols, punctuations, web links, they need to be removed and is done using *regex*.
2. **Normalization**: To standardize the cleaned text we use the *to_lower* function provided by

python to chance the case of text to small case.

3. **Tokenization**: Sentences need to be split into tokens or words so that it can be represented better and relations between words can be identified. We use the *word_tokenize* function provided by the open source library *sklearn*.
4. **Stop Words Removal:** Removal of frequently occurring words like pronouns, articles etc. rarely provide any meaning and create noise and hence need to be removed. Common stop words have been collected as a dictionary data provided by the *nltk* library.
5. **Lemmatization:** We convert words to their root forms to keep only their meaning and remove unnecessary extensions. This includes plurals, various tenses etc. We use the *WordNetLemmatizer* function provided by the *nltk* library.

A flow chart of the preprocessing steps with examples has been provided in Fig. 3.4.

Aspect Based Sentiment Analysis:

To identify a user's opinion on an aspect or a topic we need to perform aspect-based sentiment analysis. This isn't limited to users only we could extend it to items as well. Finding the most important aspects of an item and then gauging the sentiments of users on those aspects. The following steps were taken to identify aspects and perform sentiment analysis on those aspects.

1. **Embeddings:** Numeric representation of text data is necessary for the model to compute. We use *CountVectorizer* method of embedding to represent text data into numeric format. It represents the words by the frequency of its occurrence across the text.
2. **Topic Modelling**: To find relevant topics from the text data we use the *Latent Dirichlet Allocation* algorithm. It maps text to a list of topics by assigning each word in the text to different topics, based on conditional probability estimates. The most dominant topics would be used as aspects, we limit the number of aspects to 5.
3. **Sentiment Analysis**: By using *nltk* library's *SentimentIntensityAnalyzer* we gather user's polarity scores for a particular tweet and categorize them into positive, negative and neutral tweets.
4. **Aspect based sentiment analysis:** After finalizing the dominant topics we determine which aspect a keyword belongs to by taking its highest relevance score across the five

aspects. We group by the dominant topics and the sentiment scores to come up with sentiments for these aspects.

The data collected using social media sites mainly consist of text data and user's interaction with other users or a social network. This kind of data is unstructured and can be best modeled using graphs. Due to time consuming nature of this task and various API restrictions we only present the data collection of social media sites. This could be further used as additional information for recommendation task. This could be explored in the future.

In this project due to difficulties in creating our own social dataset, we utilize the help of an open source dataset from a book related social media site called goodreads. Goodreads is a social media site where users track the books they have read, want to read, share books they have read or discover books they would like to read. They can make friends on the site, follow users they know or like, follow authors they like, follow book communities they like. Users can write reviews about the books they have read, engage in discussions with other users regarding books. Mainly they can also rate the books they have read. Goodreads is the most popular site for readers and we utilize their data in recommending books users would like in the next section.

Book Recommendation System:

In this section we describe the task of recommending books to users based on the users' reading history and book features. We describe the dataset that is used, pre-processing techniques and model architecture that has been used. The data we use is based on social media site of goodreads we explore the use of graph neural networks to learn underlying representations between users and books.

Goodreads Book Dataset:

The data was collected till late 2017 from goodreads.com by University of California, San Diego and has been made public for academic use, the data was scraped only from user's public shelves [10][11]. The User IDs and review IDs were anonymized to preserve user's privacy. There were 3

groups of datasets, meta-data of the books, user-book interactions and user's public reviews. The latter wasn't used as its out of scope for the current project. The complete book dataset contains around 2.3 million works, over 400 thousand book series and over 800 thousand authors. The user-book interaction group has close to 900 thousand users and nearly 230 million interactions.

As the scale of dataset is extremely huge, we use only the subset of the complete dataset provided by the same organization, the complete datasets were divided into major genres and for this project the poetry genre was chosen.

Data Description:

The datasets in the poetry genre were divided into meta-data and user-book interactions data, where the meta-data of the books contained various information about of the books like number of users who have: read, rated, reviewed the books, the sub-genre the book belongs to, authors details etc. The user-book interactions contain the ratings, reviews and read status of the users. Total number of books under the poetry genre available are 36,514 and there are over 2.7 million interactions. The ratings given by the user range between 1 and 5, where the rating is not available or missing 0 is provided [10][11]. Table. 3.2 Table. 3.3 provide information of the available data.

Features	Value
Book ID	7327624
Title	"The Unschooled Wizard"
Language Code	US
Average Rating	4.33
Number of Pages	400
Author ID	10333
Genres	"Fantasy, Poetry"

Table 3.2 Meta-Data Of books

Features	Value
User ID	**0**
Book ID	732764
Is Read	1
Is Reviewed	1
Rating	5

Table 3.3 User-Books interactions data

Data Preparation:

The data was already removed of duplicates and important missing data. Even then there were certain discrepancies and needed to be treated. Hence, the following data preparation methods were taken:

1. **Data Reduction:** Due to limitation of computational resources the amount of data needed to be reduced as much as possible. This was done in the following steps: 1. By dropping users who have given rated less than 10 books. 2. By dropping books which have been rated by less than 30 users. 3. Removing books not read by the user. 4. Dropping interactions where user hasn't provided their rating for a book. This brings our data down to 15348 books, 21470 users and over half a million user-book interactions.

2. **Book Features**: To enhance the book dataset with more usable features we include the average rating per genre of the book. This was calculated using the user-book interaction dataset, and new genre columns with the average rating for each genre was added. There are 10 different genres in the data. The list of genres is shown in Table. 3.4 where 1 indicates the genre exists and 0 indicates that the genre is missing.

3. **User Features**: We needed to create a separate user-feature dataset, for this we took the

help of the user-book interaction and the book meta-data dataset, the user dataset now contained, the average rating the user gives to a book, the average rating for each genre the user has read a book from. Fig. 3.5 shows a sample of user features data.

We now have an additional data which include 10 genres and average rating given by each user for books and genres as well. These were done using the open source tools provided by the *pandas* library.

Genres	**Value**
Children	0
Comics	0
Fantasy_Paranormal	0
Fiction	1
Historical	0
Myster_Crime	0
Non-Fiction	0
Poetry	1
Romance	1
Young-Adult	1

Table 3.4 Genres of books added as features.

user_id	user_avg_rating	num_ratings	children	comics	fantasy_paranormal	fiction	historical_biography	myster_crime	non-fiction	poetry	romance
0	3.625000	16	4.0	4.0	3.625000	3.600000	3.666667	4.0	3.818182	3.625000	3.000000
1	4.600000	20	0.0	0.0	5.000000	4.600000	4.923077	5.0	4.562500	4.600000	4.571429
2	4.157895	19	5.0	0.0	4.545455	4.157895	4.142857	0.0	4.384615	4.157895	4.428571
3	2.700000	20	0.0	0.0	2.583333	2.700000	2.750000	3.0	2.500000	2.700000	3.142857
4	3.230769	65	0.0	0.0	3.055556	3.220339	3.300000	3.2	3.333333	3.230769	3.250000

Figure 3.5 Average rating and average rating per genre per added to enhance user features.

Graph Neural Networks:

Graph neural networks are being actively researched on for various purposes, their capabilities to model unstructured data, modeling the dependencies between the nodes in a graph enables the breakthrough in the research related to graph analysis. They are increasingly applied on various domains like social networks, knowledge graphs, recommender system and biological sciences.

Before defining graph neural networks, we need to know about the fundamentals required for developing graph neural networks. As the data used in graph neural networks are graphs, we briefly report them in the next section and mention different type of graphs that are used for graph neural networks.

Graphs:

Graph is a data structure consisting of two components vertices more commonly known as nodes and edges. A graph G can be formally defined by a set of vertices and the set of edges between these vertices, denoted as, $G = G(V, E)$. Vertices or nodes represent items, entities, or objects, which can naturally be described by quantifiable attributes [16]. Edges represent and characterize the relationships that exist between items, entities or objects. A single edge can be defined with respect to two vertices. An edge can be directed or undirected depending on whether the nodes have dependencies between them. Fig. 3.6 shows a simple graph.

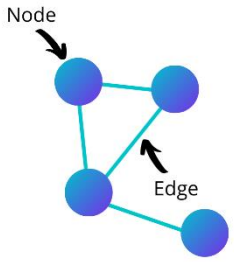

Figure 3.6 A simple graph with nodes and edges

Edges can be weighted or unweighted, in case of unweighted each edge has the same importance.

The relation between entities can have varying degrees of importance, which can be differentiated in graphs using edge weights.

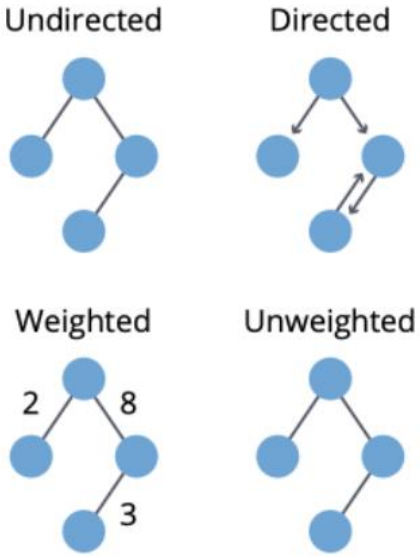

Figure 3.7 Different type of graphs

Graphs can be of different types as well in most recommendation system cases there are two kinds of entities or nodes for example a user and an item node. Graphs where multiple different nodes are present are called heterogenous graphs. Graphs that have same type of nodes are called homogenous graphs.

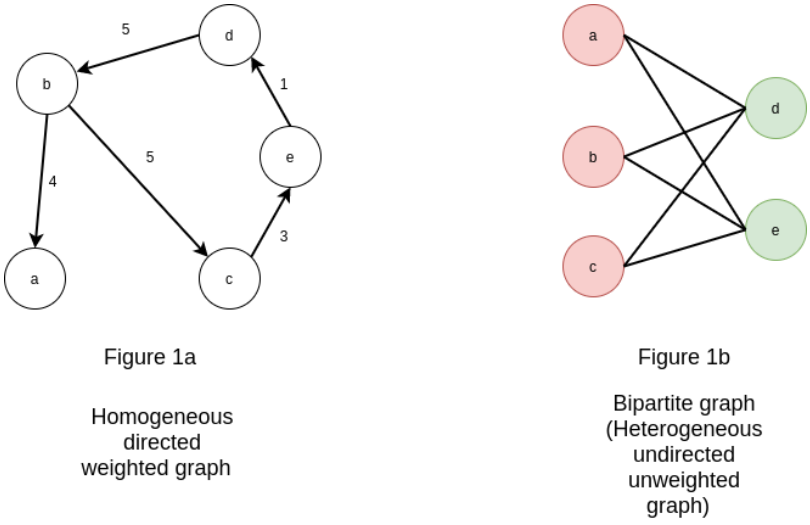

Figure 3.8 Special graphs, same colored nodes represent same kind of nodes

Another commonly used graph is the bipartite graph which is a heterogeneous graph consisting of two disjoint sets of nodes, and edges exist only between nodes from the opposite set of nodes [6].

In our case as our book dataset has 2 kinds of nodes namely book and user node and edges exist only between opposite kind of nodes that is only between book and user nodes, hence we will represent our dataset using bipartite graphs. The edges would be weighted as the ratings between the nodes.
Note, vertices or nodes can have their own set of properties usually stored in vectors and called feature vectors.
Once graphs are known we will look into what kind of tasks are possible using graphs in the next section.

Graph Tasks:

Most of these tasks require embeddings or numerical representation of the entities which are usually calculated or captured in the forward pass of the graph neural network. After embeddings are generated downstream tasks are handled, they are also the ones that produce outputs. They are mainly of three types: Node-level, edge-level and graph-level tasks [16]. The two main types that we concern with are:

1. Node-level tasks or also called node classification is formulated as binary classification or multi-class classification tasks where the goal is to predicting the identity or label of each node in the graph. Inputs here are the node embeddings.

2. Edge-level tasks or also called link predictions are very frequently used in recommendation systems. The task here is to predict whether a link/edge exists between nodes. They can be formulated as binary classification task where simply edges' existence is predicted or they could be also formulated as scoring task where embeddings are learnt using a scoring function where higher scores are given where edges are present and vice versa.

In the case of book recommendation, we need to predict whether a link exists between a user and a book node as well as the rating the user would give the book. This task can be called a link prediction or link regression task as our goal is to predict the rating the user would give the node.

Graph Neural Networks:

Now we are back to graph neural networks, it can be defined as a type of neural network which directly operates on the graph structure. The graph structure as mentioned earlier stores a lot of information which are difficult to capture using tradition techniques.
The goal of graph neural networks is to learn representations or embeddings of nodes/entities, or edges using neighborhood information. It does so in the following steps:

1. It iteratively aggregates feature information from neighbours to update the current representation of nodes/edges.
2. The aggregation function determines how the neighbours are aggregated, it could be mean function, max function etc.
3. The current representation is then updated with the aggregated representation using the update function for time t+1.
4. For each iteration, the current representation or embeddings are used as inputs for further downstream or output tasks that update the trainable parameters based on the loss function of the task.

The task of collecting or passing the feature information from neighbors is called message passing, message referring to feature information. Hence GNNs generally are based on three ideas that is message passing or propagation of information between entities, aggregating the representation and iterating them for learning further or deeper representations.

GraphSAGE:

For the task of book recommendation, we use one such variant of graph neural network called the GraphSAGE algorithm [17]. As mentioned earlier the book recommendation data could be

represented using bipartite graphs, this is because we have a user-book interactions data.

GraphSAGE is a representation learning technique, that operates on a simple assumption that vertices with similar neighborhoods should have similar embeddings. When calculating node's embedding GraphSAGE included the node's neighbors' embeddings as well. This method of learning embeddings without the need of retraining the model is called inductive learning.

Each node is represented as the aggregation of its neighborhood, hence when a new node arrives it can be represented using its neighbor nodes.

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Figure 3.9 Pseudo code of GraphSAGE algorithm

At first all node embeddings are initialized with node features. The outer for loop indicates the depth of the neighborhood, or in other words until what depth should the neighbors' embeddings must be included to calculate the embeddings for the current node and hence indicates the number of update iteration. h_v^k denotes the embedding or representation of node *v* at update iteration *k* or depth *k*.

For each node *v*, until depth *K*, neighborhood embeddings are created with the aggregator function for each node and concatenated with the existing embedding of the node. A non-linearity is used to update the node embedding. Each node embedding is then normalized.
GraphSAGE learns the aggregator weights while training which makes retraining not necessary

when a new node is added as its features can be generated using its neighborhood.

The aggregator function at each stage can be user defined and in our case we have chosen the mean aggregator function for both the neighborhood aggregation and updating of current embeddings of the node.

The aggregator function for mean is as follows:

$$h^k{}_{N(v)} = \frac{1}{|N(v)|} D_p\left[\sum_{u \in N(v)} h_u{}^{k-1}\right]$$

It aggregates all the embedding vectors for all the nodes *u* that are in the immediate neighbourhood of the target node, node *v*.

The updated representation of node *v* based on its neighbourhood aggregated representation and node *v*'s previous representation is given by:

$$h_v{}^k = \sigma\left(concat[W^k{}_{\text{self}} D_p[h_v{}^{k-1}],\ W^k{}_{\text{neigh}} h^k{}_{N(v)}] + b^k\right)$$

Here, $h_v{}^k$ is the representation of the node at layer or depth *k*. *N(v)* are the neighborhood of node *v*. b^k is an optional bias added, σ is a non-linearity added to make the model capture complexity. D_p is an optional dropout applied with probability *p*. Dropout randomly sets input units to 0 with a probability set while training the model, its aim is to reduce overfitting. W_{neigh} and W_{self} are trainable weight matrices in other words they learn weights for aggregators.

GraphSAGE uses unsupervised loss function but in our case, we have labels/ratings and hence a supervised loss function is used, namely the mean squared error.

Now that we know what model we are going to base our model and what kind of graph our model requires; in the next section we describe the conversion of book dataset to bipartite graph.

Graph Preparation:

As mentioned earlier we need to represent our book dataset in a bipartite graph. A graph can be defined as a set of nodes and edges that connect the nodes. In this case the nodes would be the users and books. The edges would represent whether the user has read or not, with the edge being a weighted edge as it would also represent the rating given by the user when the user has read the book Fig. 3.10 represents our data in graph format.

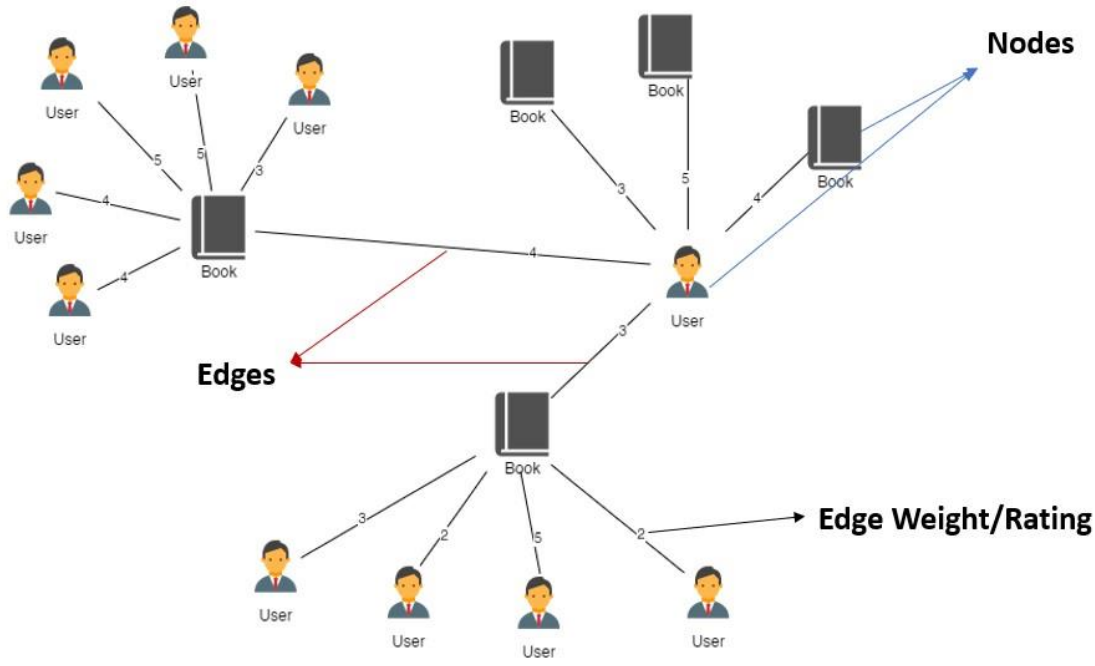

Figure 3.10 Nodes are books and users, edges only exist between a user and a book, edges have weights and that being the rating a user has given to the book.

The following preparation steps were taken:

1. The user nodes and the book nodes are represented by the uniquely identifiable number called the user_id and book_id respectively from the dataset.
2. We add node features to the user and book nodes, the user feature we use is the average rating given by the user. To book nodes we include the genres for each book, the average rating for each book and the title of the book.
3. Edges are connected between the book and user nodes. The edge connecting the books read by the users have weights which are the rating given by the users for the books, ranging

between 1 and 5.

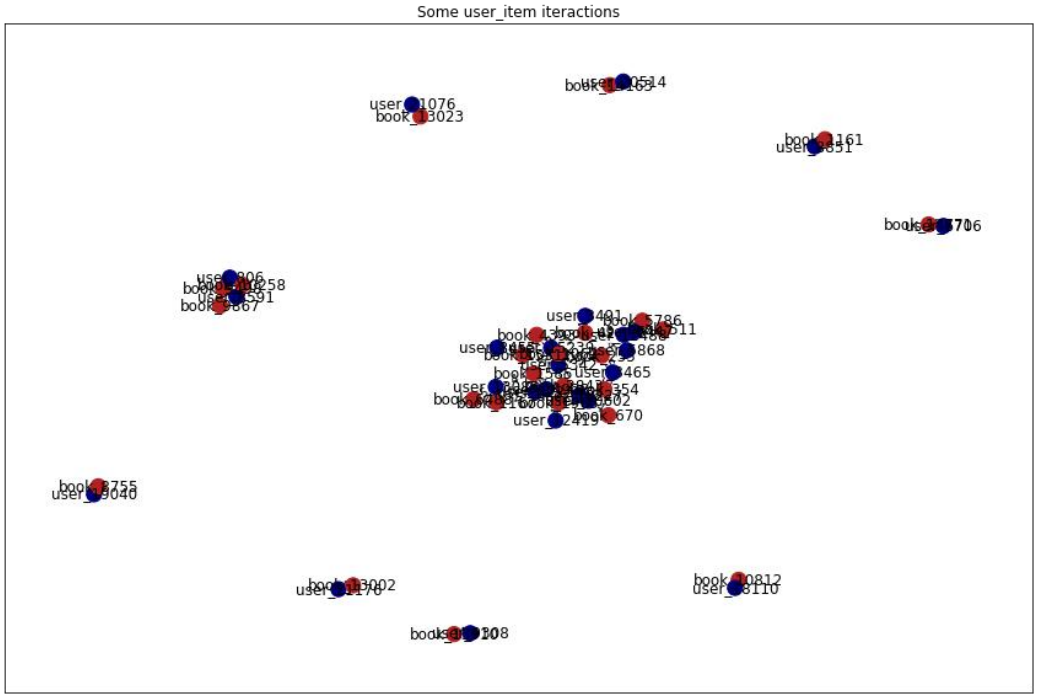

Figure 3.11 User-Item graph, where red nodes are the books and blue nodes are user nodes, edges are the ratings.

The total number of nodes and edges are 36,818 and 540,841 respectively. Fig 3.11 shows a sample of nodes in the graph format where red nodes are the books and blue nodes being the users. The numbers represent the rating the particular has given to a particular book. As the graph contains different types of nodes, this kind of graph is called a heterogenous graph which has multiple type of nodes or edges. The graph creation was done using the *networkx* library.

Model Architecture:

This section describes how the recommendation system is designed to recommend books to user from the goodreads book dataset, as mentioned earlier we will work on graph neural networks which require graph database, we have converted our book data into suitable bipartite graphs in the previous section. The below flowchart gives a glimpse of the architecture.

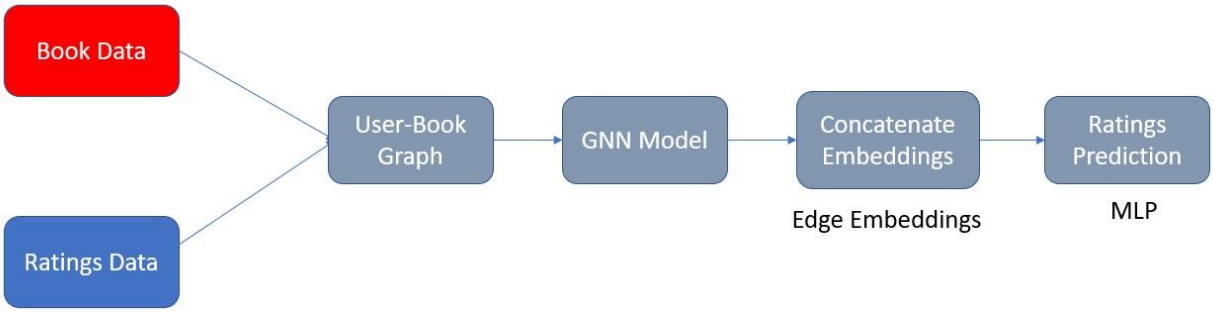

Figure 3.12 Model Flowchart

Our graph database is fed to the graph neural network model, we chose a variant of GraphSAGE as our graph neural network model as mentioned in later sections. The goal of our model is to predict the rating the user would give to a book. The method to do is as follows:

1. Use the graph database generated from previous section to train the model. The graph data contains node types and edges, nodes can be identified using the edges connecting them as mentioned in previous sections. As edges contain edge weights which contain the ratings we need to predict, graph edges are passed as input to the model.

2. Hence our task of learning user-book ratings can be defined as a supervised Link Attribute Inference: given a graph of user-movie ratings, we train a model for rating prediction using the ratings created for training, and evaluate it using the test ratings dataset. Edges are the inputs and ratings are the labels. The model requires the user-book graph structure to do neighborhood sampling which is the key role of the graph model.

3. The graph neural network model's role is to generate edge embeddings. This is created by concatenation of the user node embeddings and book node embeddings.

4. The graph task can be classified as link regression task as we generate edge embeddings and find out the rating the user would give this edge and rank them to generate recommendations. The next section describes the model and its inputs.

As we had mentioned earlier, we use GraphSAGE model to recommend books. There is issue with GraphSAGE that it hasn't been generalized with heterogenous graphs which the user-book graph is part of as it has different kind of nodes namely the user node and the book node, hence we use a implementation of GraphSAGE by *stellargraph* library called HinSAGE which is generalized for heterogenous graphs.

HinSAGE:

To support heterogeneity of nodes and edges *stellargraph* propose to extend the GraphSAGE model by having separate neighbourhood weight matrices (W_{neigh}) for every unique ordered tuple of *(N1, E, N2)* where *N1, N2* are node types, and *E* is an edge type. In addition, the heterogeneous model will have separate self-feature matrices W_{self} for every node type (this is equivalent to having a unique self-edge type for every node type).

Every edge is only associated with a single type for the starting and ending nodes which means *N1* and *N2* nodes can be known if edge *E* is specified this is equivalent to having separate neighbourhood weight matrices (W_{neigh}) for every edge type *E* [12].

The resulting feature update rules on heterogeneous graphs, for mean aggregation are as follows

$$h^{k}{}_{N_r(v)} = \frac{1}{|N_r(v)|} \sum_{u \in N_r(v)} D_p\left[{h_u}^{k-1}\right]$$

$${h_v}^{k} = \sigma\left(\text{concat}\left[{W^{k}}_{t_v,\text{self}} D_p\left[{h_v}^{k-1}\right], {W^{k}}_{r,\text{neigh}} {h^{k}}_{N_r(v)}\right] + b^{k}\right)$$

Here, *r* denotes the edge type from node *v* to node *u*. *r* is defined as unique tuple (t_v, t_e, t_u), where t_v and t_u denotes type of node *v* and node *u*, and t_e denotes the relation type. In our case there is only one type of edge relation but as HinSAGE requires edge types the work around is to have a reverse edge or mention undirected edge. That is, we use two tuples to define two edge types, one being an edge between user *(N1)* and the book node *(N2)* the other edge type being the edge

between the node book *(N1)* and the user node (*N2)*. We call these edge types user edge and book edge respectively.

$N_r(v)$ denotes the neighbourhood of node *v* via edge type *r*.

The HinSAGE model is used to generate embeddings and has been described below. The graph below shows the information nodes have that is the node feature in vector format. The node features in this case would be book genres, average rating, user average rating etc. These features are used to initialize the embeddings.

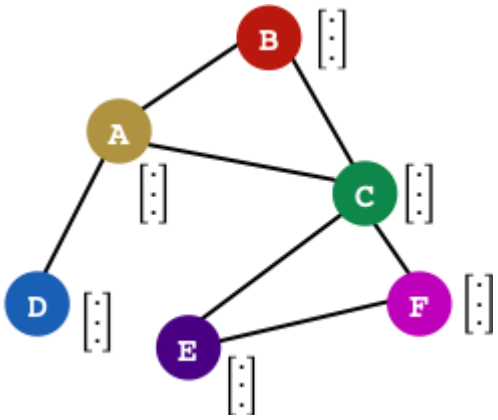

Figure 3.13 Graph with nodes and feature vectors.

Steps to generate embeddings:

1. The initial embedding vectors are initialized with the node's feature vectors. The representations are parameterized by *h*. The equation could be written as,

$$h_v^{k-1} = h_v^0 = x_v$$

where *h* is the representation of the node, *k* is the iteration, *v* represents the node and *x* the feature vector of that node.

2. Aggregate functions are used to aggregate all the embedding vectors for all the nodes *u*

$$a_v = f_{aggregate}(\{h_u | u \in N(v)\})$$

that are in the immediate neighbourhood of the target node *v*. The aggregate function in our case is the mean. The aggregated node representation is given by:

Where a_v is the aggregate representation of the node, *u* represents immediate neighbours of node *v*, $f_{aggregate}$ represents the aggregate function used.

3. Then the target node is updated using a combination of its previous representation and aggregated representation. In our case it's the mean operation. This can be represented by

$$h_v^k = f_{update}(a_v, h_v^{k-1})$$

4. The *k* parameter decides how many hops or how many neighbourhoods to compute representations from. By using $k = 2$ we include A's neighbour B and B's direct neighbours as well. Fig. 3.14 shows this process. The *k* iterates from 1 to *K* where *K* is the depth of the graph.

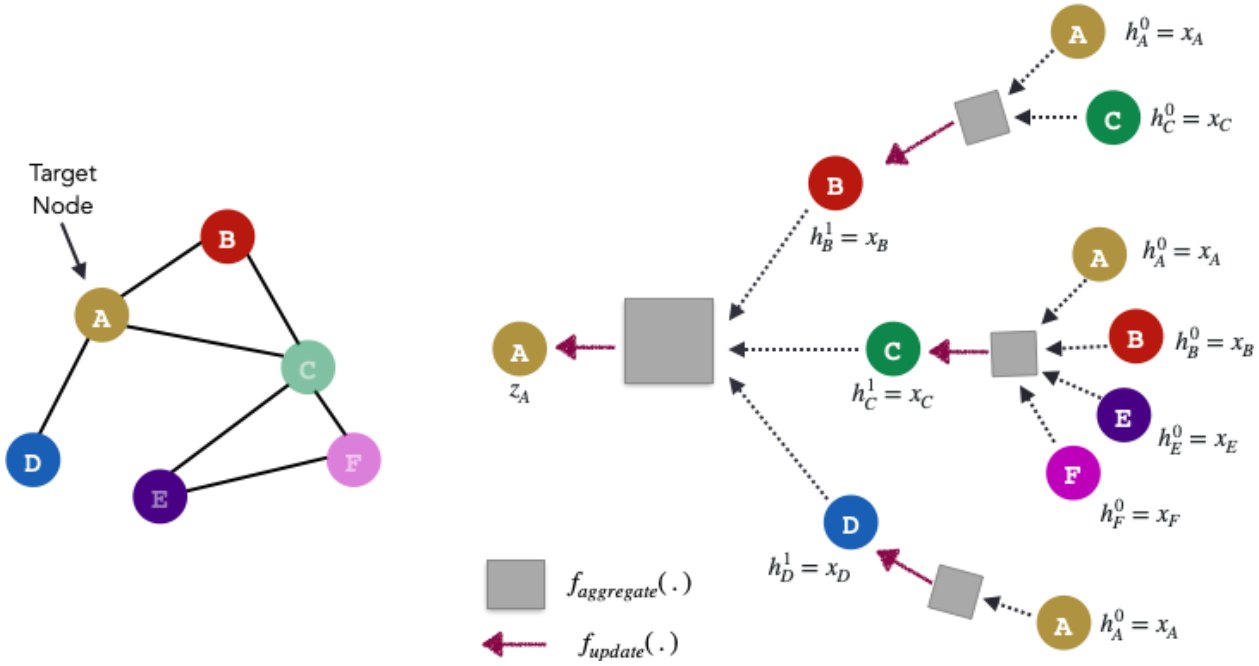

Figure 3.14 Neighbourhood Aggregation

Through one round of the HinSAGE algorithm, we will obtain a new representation for node A. The same process is followed for all the nodes in the original graph. Fig. 3.15 describes the

HinSAGE model.

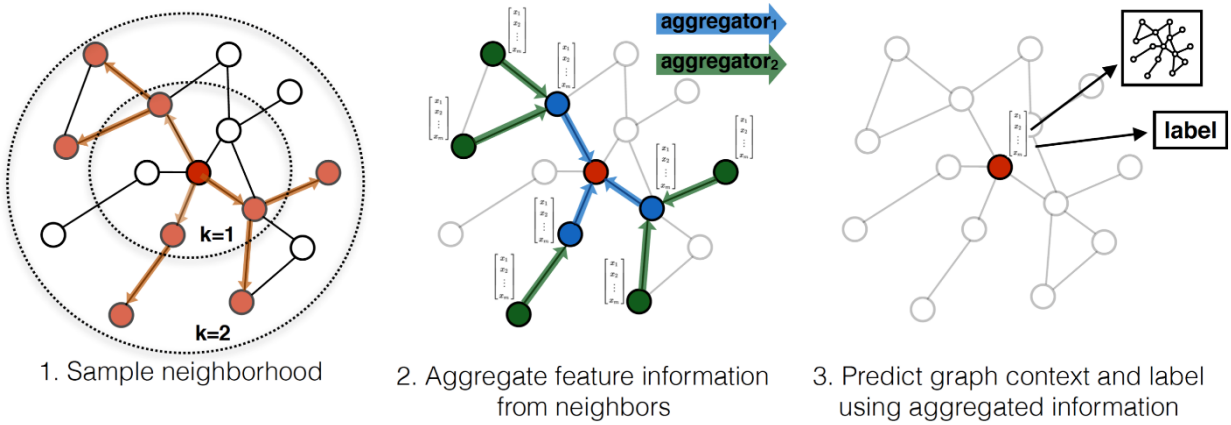

Figure 3.15 HinSAGE Model

Link Regression Task:

Once edge embeddings have been generated their labels are their edge weights or book rating given by that particular user. That is to first we need to create link representations or edge embedding between nodes of books and users, for that we concatenate the nodes embeddings of book nodes and user nodes created earlier, as link represents the user-movie ratings which are pairs of (user, book) nodes. Once link or edge embeddings have been generated the next task is to predict the rating or the labels. This task is called the link regression task. This is also called the down stream task which determines the output of the model that is the rating, wherein the loss function is minimized, embeddings are improved for each iteration of training. We predict ratings using a link regression layer provided by the *stellargraph* library. Which contains a simple multi-layer perceptron and *concat* function to concatenate the node embeddings. The node embeddings is passed to the multi-layer perceptron to predict the rating.

Loss Function:

To predict accurate ratings the model needs accurate representations of edges which in turn are determined by the embedding generation step. In neural networks this improvement takes place by minimizing a loss function. In this book recommendation system, we have gone with the mean squared error as our loss function. It is the most commonly used loss function usually paired with the root mean squared error which minimizes the square root of the mean squared error.

To calculate the mean squared error, the difference between the HinSAGE model's predictions and the ground truth or true rating, is squared, and is then averages it out across the whole dataset. The mean squared error is never negative as we square the errors. The equation is given by:

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Where y is the true rating or the label of the edge, $\hat{y}$ is the predicted rating predicted by the HinSAGE model, which is then averaged over the number of samples present in the training set.

The mean squared error is great for ensuring that the trained model has no outlier predictions with huge errors, since the mean squared error puts larger weight on these errors due to the squaring part of the function.

Model Summary:

As our inputs are directly fed to *stellargaph's* library functions, it takes care of converting, reshaping our models to its own suitability. The model summary generated by *stellargraph* is shown in the figures Fig 3.16.

The model uses various functions or layers that can be summarized below:

1. **Input layers**: The model requires edges as inputs. Stellargraph creates something called link mappers for sampling and streaming training and testing data to the model. The link mappers essentially map user-book edges or links to the input of HinSAGE: they take

minibatches of user-book edges, sample 2-hop subgraphs of our input graph with (user, book) head nodes extracted from those user-book edges, and feeds them, together with the corresponding user-book ratings, to the input layer of the HinSAGE model.

2. **Reshape:** It converts the shape of the input to desired output shape.
3. The input shapes are determined by the sizes of 1 and 2-hop neighbour samples for HinSAGE. That is the number of nodes and the number of their respective neighbors' nodes and in turn their edges. We have used 8 and 4 as the respective neighbour samples and feed it as a list to *num_samples* input for *stellargraph.*

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	[(None, 8, 1)]	0	[]
input_5 (InputLayer)	[(None, 32, 1)]	0	[]
input_6 (InputLayer)	[(None, 32, 1)]	0	[]
input_1 (InputLayer)	[(None, 1, 1)]	0	[]
reshape (Reshape)	(None, 1, 8, 1)	0	['input_3[0][0]']
reshape_2 (Reshape)	(None, 8, 4, 1)	0	['input_5[0][0]']
input_4 (InputLayer)	[(None, 8, 1)]	0	[]
reshape_3 (Reshape)	(None, 8, 4, 1)	0	['input_6[0][0]']
dropout_1 (Dropout)	(None, 1, 1)	0	['input_1[0][0]']
dropout (Dropout)	(None, 1, 8, 1)	0	['reshape[0][0]']
dropout_5 (Dropout)	(None, 8, 1)	0	['input_3[0][0]']
dropout_4 (Dropout)	(None, 8, 4, 1)	0	['reshape_2[0][0]']
input_2 (InputLayer)	[(None, 1, 1)]	0	[]
reshape_1 (Reshape)	(None, 1, 8, 1)	0	['input_4[0][0]']
dropout_7 (Dropout)	(None, 8, 1)	0	['input_4[0][0]']
dropout_6 (Dropout)	(None, 8, 4, 1)	0	['reshape_3[0][0]']

Figure 3.16 Inputs are reshaped according to the needs of the model, done by stellargraph.

4. The layers of HinSAGE model are determined by the length of num_samples in our case that would mean 2 layers of HinSAGE are present hence in Fig. 3.17 we can see two mean aggregator layers. These 2 layers are stacked and, in a sense, give a feel of deep networks.

5. Dropouts as mentioned earlier determines which inputs to be allowed to pass using probability threshold set during training time. In this case there is no dropout but could be applied to improve recommendations.

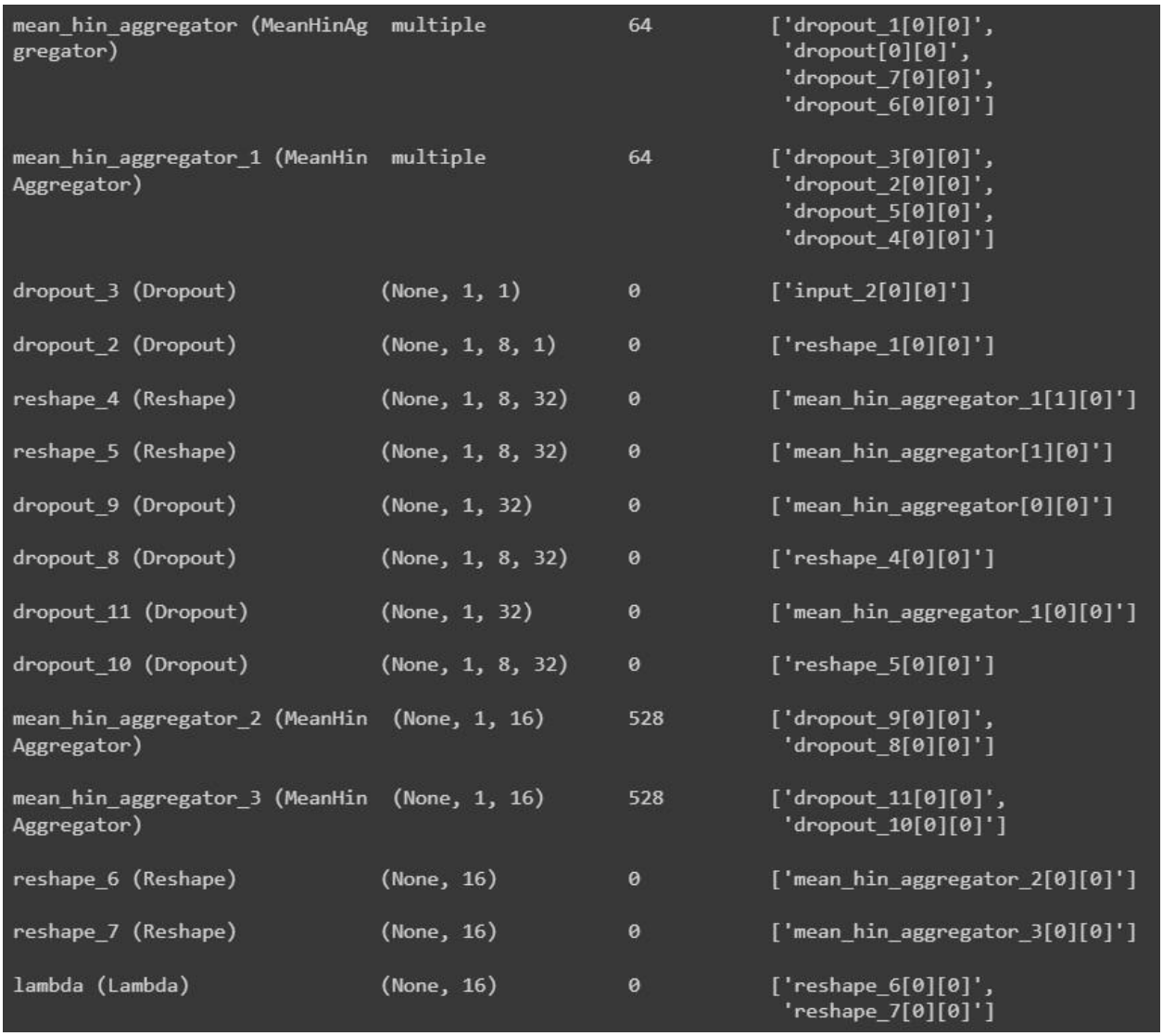

Figure 3.17 The inputs or the neighbors are fed to the mean aggregator function to calculate representations or embeddings of nodes

6. The mean aggregators take the input nodes and then the neighbors of the input nodes, as observed in reshape layer where the shape has changed from 2D to 3D to accommodate the neighbors of input nodes. This was set earlier in num_samples.

7. The aggregated nodes are then passed to another HinSAGE layer, to further aggregate them and learn useful representation for the target node. The lambda layer consists of the user and book embeddings as observed in Fig 3.18.

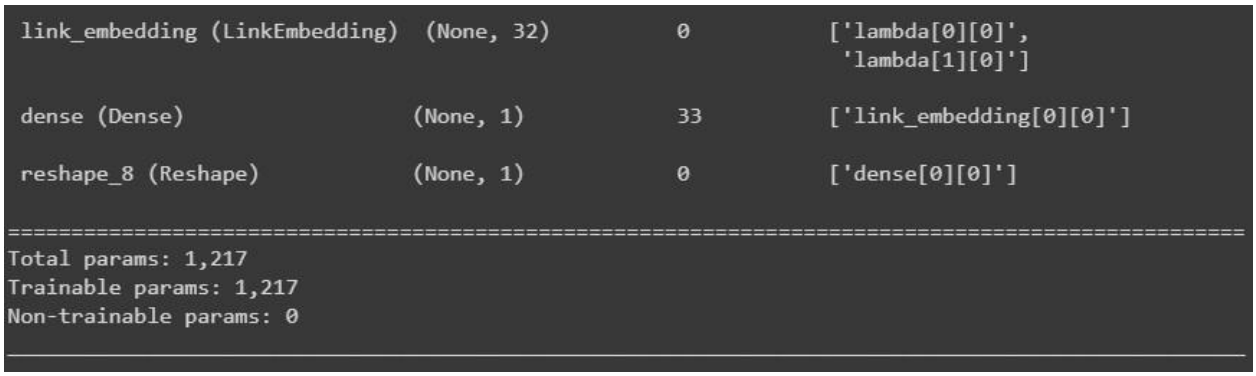

Figure 3.18 The embeddings are concatenated in the link embedding function, and passed to a neural network to predict the ratings.

8. In the *link_embedding* layer the lambda inputs or the embeddings of user nodes and book nodes are concatenated. The concatenated embeddings are passed to a dense layer.

9. **Dense Layer**: It is a simple layer of neurons in which each neuron receives input from all the neurons of previous layer. Its as simple neural network that predicts the rating of the given edge embeddings as input.

Training:

The edges from our graph dataset are divided intro training, testing and validation sets. Training set as the name suggests is for training the model, validation sets are sets to test on unseen edges that are not present in training or test set to tune the parameters of the model. This process is called

hyperparameter tuning. Test set is our evaluation set which we will look into in the results section.

The number of edges in graph equal the number of ratings in the dataset, but due to limitations from *stellargraph* it trains only on a subgraph of the overall graph. In which case the number of nodes are 15000, and the number of edges is 88,946.

In the training process, inputs are trained for 30 epochs, which means training the neural network with all the training data for 30 cycles. As the training progresses the model tries to minimize the loss function with each epoch, this way the embeddings and various learnable parameters keep improving to reduce the loss function in the downstream task of predicting accurate readings. As mentioned earlier the loss function chosen is the mean squared error. Once the model has been trained it can be used to evaluate and make recommendations. The process of making recommendations are ranking the predictions for a user has been reported in the next section.

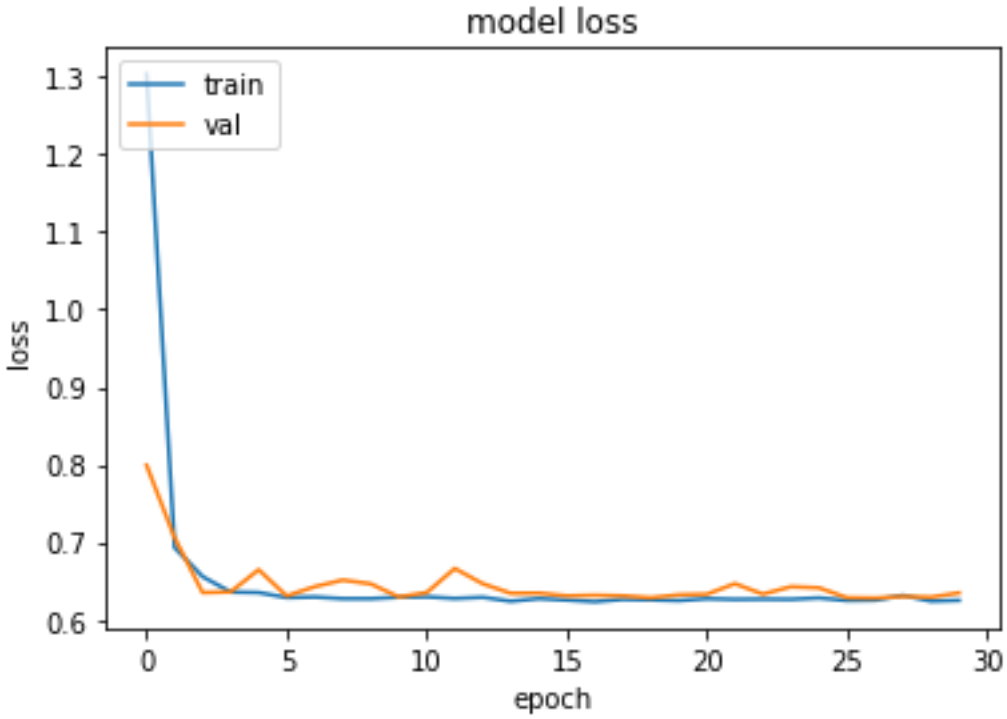

Figure 3.19 The training loss of the model.

Recommendations:

After the model is trained and link ratings have been predicted the next task is to rank the books so that they can be recommended. The recommendation is done in the following way:

1. Retrieve a list of possible edges. This is done using breadth first search (BFS) with a depth of 4 considering the user as root.
2. Use the trained model to predict link ratings.
3. Remove ratings that are below a certain threshold in our case 3.
4. Remove recommendations if any that are watched, return top-10 items.

Fig. 3.20 shows a flowchart of the process taking place.

Figure 3.20 Recommendation Flowchart

Results:

This section describes the model evaluation metric and the results observed.

To evaluate our model, we choose the following metrics, as out task is to predict the rating it comes under the regression task and hence, we use the classic regression metric mean squared error (MSE) has been chosen as the loss function to minimize. To measure the model performance, we measure the mean absolute error (MAE). The equations are described below:

$$MSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

$$MAE = \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Here *n* is the total number of samples considered, *y* is the true value or rating, $\hat{y}$ is the predicted value or rating.

We split the graph edges into 70:20:10 into train, test and validation set. The hyperparameters of the model are given in the Table 3.4. The model was trained for 30 epochs, with $Adam$ optimizer, learning rate of 0.01 and a batch size of 200 and achieved a **MAE** of **0.62**, compared with the baseline untrained model achieving a MAE score of 3.81 on the test set.

Hyperparameter	Description	Value
Epochs	Number of passes made by the algorithm on the training data.	30
Batch Size	Number of samples used in one iteration	200
Dropout	Probability of examples left out	0.0
Num Samples	Number of 1-hop 2-hop neighbour samples, it also determines the layers / iterations in the HinSAGE model.	8,4
HinSAGE Layer Sizes	Hidden layers for HinSage Layers	32, 16
Learning Rate	Determines the step size at each iteration while moving toward a minimum of a loss function.	0.01

Table 3.4 Hyperparameters

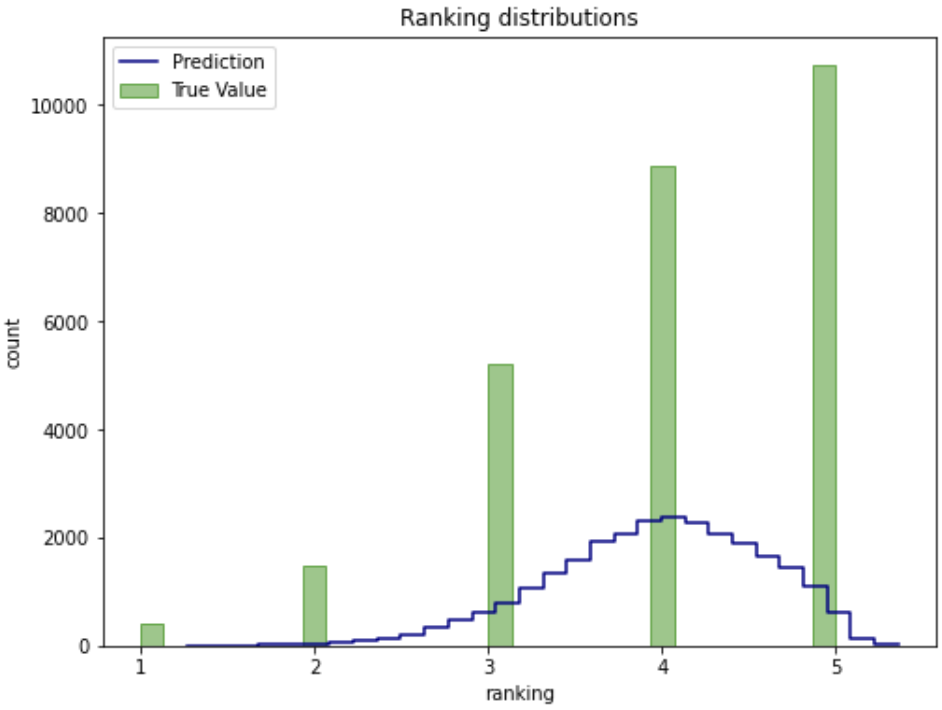

Figure 3.21 Predicted vs Actual Rating

A comparison of the predicted ratings and true ratings have been plotted in Fig. 3.21.. It shows that the model rarely predicts a 1,2 or a 5 rating, enhancements could be looked into by performing better hyperparameter tuning and more improvements have been discussed in the future prospects

section.

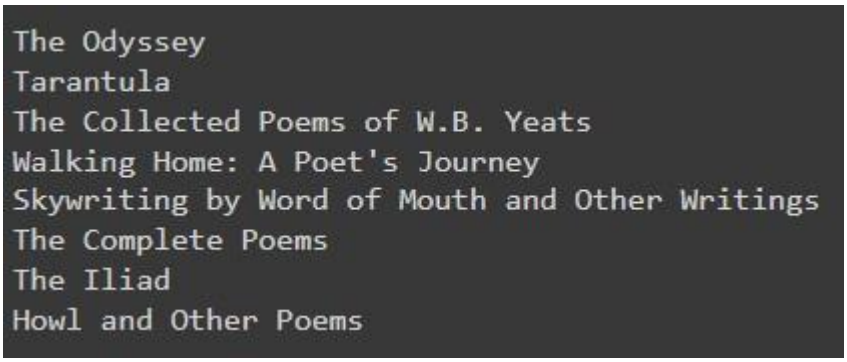

Figure 3.22 User's read history

Some of the recommended books by the model is shown in Fig. 3.33 for a user. Comparing with user's history of reading books in Fig. 3.22 can observe that user likes historical books as observed by the user reading ``The Iliad" and ``The Odyssey" ancient Greek epic poems by Homer. The model predicts the user would very much like historical or classical poems as evident by recommendations like ``Dante's Inferno" and ``Sir Gawain and the Green Knight" both very much historical and classical poems as well.

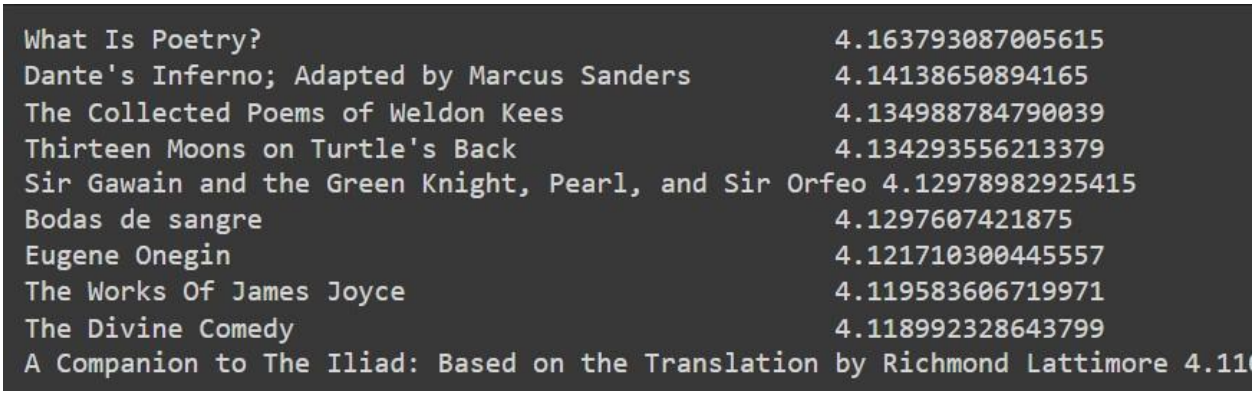

Figure 3.23 Recommended books for the user.

In the next section we briefly discuss where the implementation steps to make the recommendation system model.

Implementation:

In this section we briefly look into the coding segments of the recommendation system model. The codes were written using python language in Google colab. Google colab provide colab notebooks based on jupyter notebooks which allow execution of python codes remotely. It lessens the hassle of installing libraries, etc. which are done by colab itself.

We start with downloading the dataset, the dataset provided by UCSD were of large size and hence use google drive to store them. File ids are provided and the below code uses those file ids to download the required dataset.

```
file_id_map = dict(zip(file_ids['name'].values, file_ids['id'].values))

def download_by_name(fname, output=None, quiet=False):
    if fname in file_id_map:
        url = 'https://drive.google.com/uc?id='+file_id_map[fname]
        gdown.download(url, output=output, quiet=quiet)
    else:
        print('The file', fname, 'can not be found!')
```

Figure 3.24 Code to download dataset

We download the poetry books dataset provided by the UCSD book dataset, and use its id to retrieve, meta-data, user-book interactions and review dataset.

As the files are of large size, they are in compressed json format and they need to be converted to a more suitable format for python to effectively work. One such format is the parquet file format which are used to reduce large sizes and are very efficient as well. The below code converts the file into the required format. The command is part of the *pandas* library.

```
save_filepath = 'goodreads_books_poetry.snappy.parquet'
poetry_books.to_parquet(save_filepath, compression='snappy', index=False)
del poetry_books
```

Figure 3.25 Code to convert file format

Once converted we can efficiently work with the data, the below codes are used for observation of said converted data.

```
save_filepath = 'goodreads_interactions_poetry.snappy.parquet'

df_ratings = pd.read_parquet(save_filepath)
print(f'Number of Records: {len(df_ratings):,}\nUnique Books: {df_ratings.book_id.nunique()}')
df_ratings.head(5)
```

```
Number of Records: 2,734,350
Unique Books: 36514
```

	user_id	book_id	review_id	is_read	rating
0	8842281e1d1347389f2ab93d60773d4d	1384	1bad0122cebb4aa9213f9fe1aa281f66	True	4
1	8842281e1d1347389f2ab93d60773d4d	1376	eb6e502d0c04d57b43a5a02c21b64ab4	True	4
2	8842281e1d1347389f2ab93d60773d4d	30119	787564bef16cb1f43e0f641ab59d25b7	True	5
3	72fb0d0087d28c832f15776b0d936598	24769928	8c80ee74743d4b3b123dd1a2e0c0bcac	False	0
4	72fb0d0087d28c832f15776b0d936598	30119	2a83589fb597309934ec9b1db5876aaf	True	3

Figure 3.26 Code to observe data

The following codes assist in cleaning the data

Removing unnecessary columns from books dataset and keeping only useful ones the following code helps in that case:

```
cols = ['isbn', 'text_reviews_count', 'country_code', 'language_code', 'popular_shelves', 'is_ebook',
        'average_rating', 'similar_books', 'description', 'authors',  'num_pages', 'publication_year',
        'url', 'book_id', 'ratings_count', 'title']

poetry_books = poetry_books[cols]

save_filepath = 'goodreads_books_poetry.snappy.parquet'
poetry_books.to_parquet(save_filepath, compression='snappy', index=False)
del poetry_books
```

Figure 3.27 Keeping useful columns

To keep only necessary columns in the ratings dataset.

```
poetry_cols = ['user_id', 'book_id', 'review_id', 'is_read', 'rating']
poetry_ratings = poetry_ratings[poetry_cols]
```

Figure 3.28 Keeping useful columns in ratings data

To include additional features for both the user and books data, we use the following codes. To find and include genres of books the following codes are used

```
save_filepath = 'goodreads_books_genre.snappy.parquet'

df_genres = pd.read_parquet(save_filepath)
print(f'Number of Records: {len(df_genres):,}\nUnique Books: {df_genres.book_id.nunique()}')
df_genres.head()
```

```
Number of Records: 2,360,655
Unique Books: 2360655
```

	book_id	genres
0	5333265	{'children': None, 'comics, graphic': None, 'f...
1	1333909	{'children': None, 'comics, graphic': None, 'f...
2	7327624	{'children': None, 'comics, graphic': None, 'f...
3	6066819	{'children': None, 'comics, graphic': None, 'f...
4	287140	{'children': None, 'comics, graphic': None, 'f...

Figure 3.29 Book genres are found out from another dataset provided by UCSD

The genres in the above code cannot be added to books meta-data as it is and require certain improvement, which the below code does. It explodes the dictionary of genres into columns and fills missing genres with 0 and genres present with 1.

```
df_genres = df_genres.join(pd.json_normalize(df_genres['genres']))
df_genres.drop(columns=['genres'], inplace=True)

df_genres.fillna(0, inplace=True)
df_genres.head()
```

	book_id	children	comics, graphic	fantasy, paranormal	fiction	history, historical fiction, biography	mystery, thriller, crime	non-fiction	poetry	romance	young-adult
0	5333265	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
1	1333909	0.0	0.0	0.0	219.0	5.0	0.0	0.0	0.0	0.0	0.0
2	7327624	0.0	0.0	31.0	8.0	0.0	1.0	0.0	1.0	0.0	0.0
3	6066819	0.0	0.0	0.0	555.0	0.0	10.0	0.0	0.0	23.0	0.0
4	287140	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0

Figure 3.30 Code to find genres

The above genres need to be merged with the main dataset, we do so by using *pandas merge* function and merging it on book_id. The below code shows the process.

```
df_poetry = pd.merge(df_books, df_genres, on="book_id", how="left")
```

```
df_poetry = pd.read_parquet(save_filepath)
print(f'Number of Records: {len(df_poetry):,}\nUnique Books: {df_poetry.book_id.nunique()}')
df_poetry.head()
```

code	language_code	popular_shelves	is_ebook	average_rating	similar_books	description	authors	...	comics, graphic	fantasy, paranormal	fiction	history, historical fiction, biography	mystery, thriller, crime	non-fiction	poetry	romance	young-adult	author_
US	eng	[{'count': '8', 'name': 'to-read'}, {'count': ...	false	3.83	[]	Number 30 in a series of literary pamphlets pu...	[{'author_id': '15585', 'role': ''}]	...	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	155
US		[{'count': '100', 'name': 'to-read'}, {'count'...	false	3.83	[]	Fairy Tales gathers the unconventional verse d...	[{'author_id': '16073', 'role': ''}, {'author_...	...	0.0	0.0	2.0	0.0	0.0	0.0	2.0	0.0	0.0	160
US		[{'count': '32', 'name': 'to-read'}, {'count':...	false	4.38	[]	Three poems describe the nighttime adventures ...	[{'author_id': '18540', 'role': ''}, {'author_...	...	1.0	1.0	1.0	0.0	0.0	1.0	10.0	0.0	0.0	185
US		[{'count': '554', 'name': 'to-read'}, {'count'...	false	3.71	[1230072, 315167, 676169, 18522, 124335.	A modern verse play about the search for	[{'author_id': '18540', 'role': ''}]	...	0.0	0.0	30.0	0.0	0.0	0.0	29.0	0.0	0.0	185

Figure 3.31 Code to merge genres with meta-data of books

Now we make additional features for the dataset, this includes, average rating given by user, average rating per genre given by the user etc. The below codes do that.

The ratings data is cleaned, redundant columns and rows are removed:

```
## Ratings

ratings = ratings.merge(book_map, on='book_id')
ratings.drop(columns='book_id', inplace=True)
ratings.rename(columns={'book_id2':'book_id'}, inplace=True)

unique_book_ids = book_info['book_id'].unique()
ratings = ratings[ratings['book_id'].isin(unique_book_ids)]
print(f'Ratings left: {len(ratings)}')

ratings = ratings[ratings['is_read']==True]
ratings = ratings[ratings['rating']!=0]

ratings[['user_id']] = ratings[['user_id']].apply(lambda x: pd.factorize(x)[0])
```

Figure 3.32 Cleaning the ratings data.

To calculate user average ratings and average ratings per genre the above ratings data is used to create a user only data consisting of number of ratings, average ratings and average rating per

genre. The below codes do that.

```
## USER INFO: User ratings per genre

lens = pd.merge(ratings, book_info, on='book_id' )
lens['user_avg_rating'] = lens.groupby('user_id')['rating'].transform('mean')

lens['num_ratings']=lens.groupby('user_id')['user_id'].transform('count')

genres = ['children', 'comics','fantasy_paranormal','fiction','historical_biography','myster_crime','non-fiction','poetry','romance','young-adult']

mdf = pd.melt(lens[['user_id','user_avg_rating','num_ratings', 'rating'] + genres], id_vars=['user_id','user_avg_rating','num_ratings', 'rating'], var_name='genre')

mdf = mdf[mdf['value']!=0][['user_id','user_avg_rating', 'num_ratings', 'rating', 'genre']]

# RUN PIVOTED AGGREGATION
users = pd.pivot_table(mdf, columns = ['genre'], index = ['user_id', 'user_avg_rating', 'num_ratings'], values = ['rating'], aggfunc = np.mean)

users.columns = users.columns.droplevel(0)
users.columns.name = None
users = users.reset_index()

users.fillna(0.0, inplace=True)
```

Figure 3.33 Ratings and meta-data are merged to create helpful data for creating user features

Due to limitations in computational resources we use the following codes to remove rows of data based on heuristically determined thresholds.

```
#len(users[users['num_ratings']>20])
user_info = users[users['num_ratings']>30]

unique_users = user_info['user_id'].unique()
ratings = ratings[ratings['user_id'].isin(unique_users)]

save_filepath = 'goodreads_poetry_users.snappy.parquet'
user_info = pd.read_parquet(save_filepath)
user_info = user_info[user_info['num_ratings']>10]

unique_users = user_info['user_id'].unique()
ratings = ratings[ratings['user_id'].isin(unique_users)]
```

Figure 3.34 Reduce rows of users

To convert the current data, certain columns need to be standardized. For example, the user_id and book_id are unsorted and do not start from 0, the graph creating libraries require the ids to start from 0 and be in ascending order. The below codes do that:

```python
# Adjust USER_INFO
# user_id starts from 1 and goes uniterrupted up to max(user_id), so we just need to move the column one number down
#ratings['user_id'] = ratings['user_id'].apply(lambda x: x-1)
user_info['user_id'] = user_info[['user_id']].apply(lambda x: pd.factorize(x)[0])
ratings['user_id'] = ratings[['user_id']].apply(lambda x: pd.factorize(x)[0])

ratings = ratings.sort_values('user_id')
#adjust book_INFO
# Create a dict for bookId's - we will use an index for books
book_index_dict = {}
for book_id in book_info['book_id']:
    if book_id not in book_index_dict.keys():
        book_index_dict[book_id] = len(book_index_dict)
    else:
        print('Error - duplicate book ids')

# Add book_graph_id to ratings and book_info
ratings['book_graph_id'] = ratings['book_id'].apply(lambda x: book_index_dict[x])
book_info['book_graph_id'] = book_info['book_id'].apply(lambda x: book_index_dict[x])

# Sort books by 'book_graph_id' (unnecesary but better safe than sorry)
book_info.sort_values(by=['book_graph_id'], inplace=True)
```

Figure 3.35 Code to standardize the data

Non numerical columns are removed to avoid complication of task in the below code:

```python
# Remove non-numeric columns from book_info
book_info_reccomend = book_info.copy() #later for recommendations (we need to have title to display to user)
book_info = book_info.select_dtypes(['number'])
print(f'Number of numeric columns: {len(book_info.columns)}')
```

```
Number of numeric columns: 14
```

Figure 3.36 Code to remove non-numeric columns

After doing the above we get the following processed data:

```
book_info.head(3)
```

	average_rating	ratings_count	children	comics	fantasy_paranormal	fiction	historical_biography	myster_crime	non-fiction	poetry	romance	young-adult	book_graph_id	book_id
2862	4.17	865	33.0	0.0	0.0	10.0	0.0	0.0	3.0	90.0	0.0	0.0	0	0
2863	4.03	321	44.0	0.0	2.0	20.0	0.0	0.0	2.0	89.0	0.0	0.0	1	1
21970	3.95	734	0.0	0.0	0.0	5.0	17.0	0.0	12.0	14.0	0.0	0.0	2	2

Figure 3.37 Book data

```
ratings.head(3)
```

	user_id	review_id	is_read	rating	book_graph_id	book_id
0	0	5db66c197f8f5bd85633cf9076790282	True	3	19	19
3645	0	042389fc8b72ab90fb9918bb8cab2c0d	True	2	1940	1940
3528	0	836a35ec6750464371124db32c9211aa	True	4	1507	1507

Figure 3.38 Ratings data

```
user_info.head(3)
```

	user_id	user_avg_rating	num_ratings	children	comics	fantasy_paranormal	fiction	historical_biography	myster_crime	non-fiction	poetry	romance	young-adult
0	0	3.369565	46	0.0000	0.000000	3.266667	3.340909	3.407407	3.666667	3.423077	3.369565	3.230769	5.000000
6	1	4.938865	229	4.9375	4.833333	5.000000	4.943925	4.939655	4.916667	4.925620	4.938865	4.945946	4.944444
9	2	3.339416	274	2.5000	4.000000	3.322581	3.348548	3.408451	3.000000	3.300000	3.339416	3.567164	2.700000

Figure 3.39 User data

The data we have has to be converted to graph which is taken care of the in the below code. We use *network*x's library to convert our data into a representation of an undirected heterogenous graph.

```
# Add user nodes
for i in progress_bar(ratings.user_id.unique()):
  G.add_nodes_from([
    ('user_' + str(i), {'node_type':'user'})
  ])
```

Figure 3.40 Code to add user nodes

```
# Add book nodes
for i in progress_bar(ratings.book_id.unique()):
  name = book_info[book_info['book_id']==i].title.to_string(index=False)[0:]
  genre = book_info[book_info['book_id']==i].genre.to_string(index=False)[0:].split(',')

  rating = book_info[book_info['book_id']==i].average_rating.to_string(index=False)
  if rating != 'NaN':
    rating = float(rating[0:])
  else:
    rating = 0.0


  G.add_nodes_from([
    ('book_' + str(i), {'node_type':'book',
                        'name':name,
                        'genre':genre,
                        'rating':rating})
  ])
```

Figure 3.41 Code to add book nodes to the graph

```
# Add user-item interactions -> Edges
for i in progress_bar(range(ratings.shape[0])):
  user = 'user_' + str(ratings.iloc[i,0])
  book = 'book_' + str(ratings.iloc[i,4])
  rating = ratings.iloc[i,3]

  G.add_edges_from([(user, book, {'weight':rating})])
```

Figure 3.42 Code to add edges to the graph

After the graph is made from *networkx* library we have to prepare the graph to be used for the model which is done using the libraries of *stellargraph* in the below codes.

```
from stellargraph import StellarGraph
import stellargraph as sg
from stellargraph.mapper import HinSAGELinkGenerator
from stellargraph.layer import HinSAGE, link_regression
from tensorflow.keras import Model, optimizers, losses, metrics, layers, regularizers

G = nx.readwrite.edgelist.read_weighted_edgelist('graph.edgelist')

# Add user node type
for i in ratings.user_id.unique():
  attrs = {'user_' + str(i): {'node_type':'user'}}
  nx.set_node_attributes(G, attrs)

# Add book node type
for i in ratings.book_id.unique():
  attrs = {'book_' + str(i): {'node_type':'book'}}
  nx.set_node_attributes(G, attrs)
```

Figure 3.43 Code to add node types

```
num_users = len(ratings['user_id'].unique())
num_items = len(ratings['book_id'].unique())
num_nodes = num_users + num_items

print('Number of nodes: ', num_nodes)
print('Number of edges: ', ratings.shape[0])
```

```
Number of nodes:  36818
Number of edges:  540841
```

Figure 3.44 Code to observe number of nodes and edges

Now that node and node types have been created, node features are next to be included which is done in the below code.

```
data_user = defaultdict()
data_book = defaultdict()


# --------- Users features ------------
# -------------------------------------
for i in userList:
  avg_rating = np.mean([e[2]['weight'] for e in G.edges(i, data=True)])
  data_user[i] = avg_rating

user_features = pd.DataFrame.from_dict(data_user, orient='index', columns=['avg_rating'])


# --------- book features ------------
# -------------------------------------
for i in bookList:
  # RATING
  code = int(i[5:])
  rating = book_info[book_info['book_id']==code].average_rating.to_string(index=False)
  if rating != 'NaN':
    rating = float(rating[0:])
  else:
    rating = 0.0


  # FEATURES
  data_book[i] = [rating]


book_features = pd.DataFrame.from_dict(data_book, orient='index', columns=['rating'])
```

Figure 3.45 Code to add features to graph

Stellargraph uses subgraphs to train the model and hence subgraphs are created and their properties are shown in the below codes.

```
random.seed = seed

k_nodes = rd.sample(list(G.nodes()), k=k)
subG = G.subgraph(k_nodes)

userList = [n for n,d in subG.nodes(data=True) if d['node_type'] == 'user']
bookList = [n for n,d in subG.nodes(data=True) if d['node_type'] == 'book']

num_users = len(set(userList))
num_items = len(set(bookList))

print('Number of users: ', num_users)
print('Number of books: ', num_items)
print()
print('[Info] Number of NODES: ', num_users + num_items)
print('[Info] Number of EDGES: ', len(subG.edges()))
```

Figure 3.46 Code to create subgraphs

```
g = StellarGraph.from_networkx(subG, node_type_attr='node_type', edge_weight_attr='weight', edge_type_default='rating',
                               node_features={'user': user_features, 'book': book_features})
print(g.info())
```

```
StellarGraph: Undirected multigraph
Nodes: 15000, Edges: 88946

Node types:
 user: [8747]
   Features: float32 vector, length 1
   Edge types: user-rating->book
 book: [6253]
   Features: float32 vector, length 1
   Edge types: book-rating->user

Edge types:
   book-rating->user: [88946]
       Weights: range=[1, 5], mean=4.04695, std=0.976513
       Features: none
```

Figure 3.47 Properties of graph created by stellargraph

Edges of the graphs are used as inputs which are passed to the model with their weights or ratings as the label. They are hence split into train, test and validation sets in the below code:

```
edges_train_val, edges_test = model_selection.train_test_split(
    E, train_size=train_size, test_size=test_size, random_state=seed
)

edges_train, edges_val = model_selection.train_test_split(
    edges_train_val, test_size=val_size, random_state=seed
)


edgelist_train = edges_train.iloc[:, :2].to_numpy()
edgelist_val = edges_val.iloc[:, :2].to_numpy()
edgelist_test = edges_test.iloc[:, :2].to_numpy()

labels_train = edges_train['rating'].to_numpy()
labels_val= edges_val['rating'].to_numpy()
labels_test = edges_test['rating'].to_numpy()
```

Figure 3.48 Edges split into test train and validation sets.

To feed the graph to the model, generators are used, to feed the data properly. They sample the required with user and book head nodes extracted from the user-book edges, and feed them together with the ratings found in user-book edges.

```
generator = HinSAGELinkGenerator(
    g, batch_size, num_samples, head_node_types=["user", "book"]
)
train_gen = generator.flow(edgelist_train, labels_train, shuffle=True)
val_gen = generator.flow(edgelist_val, labels_val)
```

Figure 3.49 Code to generate data from graph to model

Once the above are done, we need to connect the above with the HinSAGE model, where necessary inputs are mentioned, parameters that are required and finally connect the output of HinSAGE model to *link_regression* layer to predict the ratings.

```
assert len(hinsage_layer_sizes) == len(num_samples)

hinsage = HinSAGE(
    layer_sizes=hinsage_layer_sizes, generator=generator, bias=True, dropout=dropout
)

x_inp, x_out = hinsage.in_out_tensors()

output_dim = 1 # default = 1, others = 10
score_prediction = link_regression(output_dim=output_dim, edge_embedding_method="concat")(x_out)
```

```
link_regression: using 'concat' method to combine node embeddings into edge embeddings
```

Figure 3.50 HinSAGE model

The model now needs to be trained and is done in the below code we also set the loss function that needs to be minimized here.

```
# Model creation
model = Model(inputs=x_inp, outputs=score_prediction)
model.compile(
    optimizer=optimizers.Adam(learning_rate=1e-2),
    loss=losses.mean_squared_error,
    metrics=[metrics.RootMeanSquaredError(), metrics.mae],
)
```

Figure 3.51 Model Training

Now that the model has been trained and is ready to make predictions of ratings, we can start recommending. Recommendation involves certain steps as mentioned earlier and the below code implements them.

The following code implements a breadth first search with a depth of four, finding ideal candidates of books and making sure no unnecessary nodes like user nodes are found.

```
user = rd.choice(userList)
num_top = 10
d = 4        # maximum search depth

# ----------------------------------------------
# ------------- candidate edges ---------------
#
# An oriented tree constructed from of a breadth-first-search starting at source
edges = []
node_attributes = nx.get_node_attributes(subG, 'node_type')
for u in list(nx.algorithms.traversal.breadth_first_search.bfs_tree(subG, user, depth_limit=d).nodes()):
  if node_attributes[u] == 'book':
    edges.append((user, u))
```

Figure 3.52 Code to retrieve ideal candidates

The candidate edges are then used to make predictions with our trained model, but once its done, only relevant recommendations are needed to be made, a threshold is set for the ratings above which books are recommended.

```
# filtering all ratings that don't reach a certain threshold
th = 3
user_pred = [r for r in user_pred if r >= th]
edges = [edges[i] for i in range(len(user_pred)) if user_pred[i] >= th]

temp_user_pred = []
temp_edges = []

# filtering all suggestions already read
for i in range(len(edges)):
  for a in subG.neighbors(user):
    if a != edges[i][1]:
      temp_user_pred.append(user_pred[i])
      temp_edges.append(edges[i])
```

Figure 3.53 Code to filter recommendations

After filtering the ratings, we are now ready to make recommendations which are handled by the below code.

```
suggestions = np.flip(np.argsort(user_pred))[:num_top]

for s in suggestions:
  book_id = edges[s][1][6:]
  i = book_info[book_info['book_id']==int(book_id)].index[0]
  name = book_info.at[i, 'title']

  for j in range(50 - len(name)):
    name += ' '
  print('{} {}'.format(name, user_pred[s]))
```

```
What Is Poetry?                                        4.163793087005615
Dante's Inferno; Adapted by Marcus Sanders             4.14138650894165
The Collected Poems of Weldon Kees                     4.134988784790039
Thirteen Moons on Turtle's Back                        4.134293556213379
Sir Gawain and the Green Knight, Pearl, and Sir Orfeo 4.12978982925415
Bodas de sangre                                        4.1297607421875
Eugene Onegin                                          4.121710300445557
The Works Of James Joyce                               4.119583606719971
The Divine Comedy                                      4.118992328643799
A Companion to The Iliad: Based on the Translation by Richmond Lattimore 4.11078
```

Figure 3.54 Code for generating recommendations

The above codes give a brief on how the recommendation system works from technical/code perspective. The coding can be refactored and certain sections can be modified to recommend better.

Chapter 4
Conclusion

In this project looked into creating a recommendation system leveraging social data. We created a pipeline to collect social data from social media sites was developed. Open source dataset from social media site goodreads.com for tracking, planning and sharing books read with other users was used to build a recommendation system.

The recommendation system was built around graph neural networks, a technique that is being actively researched for solving various deep learning problems. We converted our dataset into graphs and fed it to a GNN algorithm called HinSAGE for link regression task. To predict what rating a user would give to an unread book and then recommend best predicted books to the user.

Chapter 5

Future Prospects

- In this project we have implemented a recommendation system that leverages social data provided by goodreads.com to recommend books to the user. We used a HinSAGE graph neural network model to recommend users. In this current project we do not use the completely available data to train the model, when its known that deep learning performs better with more data. This is due to limited computational capabilities available and technical difficulties and could be looked into later with more efficient coding and tools.

- We don't use the additional features available like number of pages, author details etc. for the books and also do not use and user features apart from ratings even though average rating per sub-genre was created during the data preparation phase. This could possibly improve the recommendations by the model.

- The recommendation system currently deals with cold-start problem which means when a new user enters there won't be any good recommendations for the user. To deal with popularly read books could be recommended and once the user has read enough the model could learn and improve the recommendations.

The future prospects could be summarized as follows:

- Include more data for the model.
- Increase the user and book features to enhance recommendations.
- Solve cold-start problems to provide new users with good recommendations.

References

[1] H. Ko, S. Lee, Y. Park, and A. Choi, "A survey of recommendation systems: Recommendation models, techniques, and application fields," Electronics, vol. 11, no. 1, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/1/141

[2] A. Roy. Introduction to recommender systems- 1: Content-based filtering and collaborative filtering. [Online]. Available: https://towardsdatascience.com/introduction-to-recommender-systems-1-971bd274f421.

[3] D. Chen. Recommender system — matrix factorization. [Online]. Available: https://towardsdatascience.com/recommendation-system-matrix-factorization-d61978660b4b.

[4] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, Neural Collaborative Filtering, ser. WWW '17. International World Wide Web Conferences Steering Committee, 2017.

[5] C. Gao, X. Wang, X. He, and Y. Li, Graph Neural Networks for Recommender System, ser. WSDM '22. Association for Computing Machinery, 2022.

[6] S. Rajamanickam. Graph neural network (gnn) architectures for recommendation systems. [Online]. Available: https://towardsdatascience.com/graph-neural-network-gnn-architectures-for-recommendation-systems-7b9dd0de0856

[7] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua, Neural Graph Collaborative Filtering. ACM, jul 2019.

[8] J. DeBlois-Beaucage. Building a recommender system using graph neural networks. [Online]. Available: https://medium.com/decathlontechnology/buildinga-recommender-system-using-graph-neural-networks-2ee5fc4e706

[9] S. Rendle, W. Krichene, L. Zhang, and J. Anderson, "Neural collaborative filtering vs. matrix factorization revisited," 2020. [Online]. Available: https://arxiv.org/abs/2005.09683

[10] M. Wan and J. J. McAuley, Item recommendation on monotonic behavior chains, S. Pera, M. D. Ekstrand, X. Amatriain, and J. O'Donovan, Eds. ACM, 2018. [Online]. Available: https://doi.org/10.1145/3240323.3240369

[11] M. Wan, R. Misra, N. Nakashole, and J. J. McAuley, Fine-Grained Spoiler

Detection from Large-Scale Review Corpora, A. Korhonen, D. R. Traum, and 25 L. M`arquez, Eds. Association for Computational Linguistics, 2019. [Online]. Available: https://doi.org/10.18653/v1/p19-1248

[12] S. Graph. Link prediction with heterogeneous graphsage (hinsage). [Online]. Available: https://stellargraph.readthedocs.io/en/stable/demos/link-prediction/hinsage-link-prediction.html

[13] N. Abraham. Ohmygraphs: Graphsage and inductive representation learning. [Online]. Available: https://medium.com/analytics-vidhya/ohmygraphs-graphsage-and-inductive-representation-learning-ea26d2835331

[14] Zhao, H.; Yao, Q.; Song, Y.; Kwok, J.T.; Lee, D.L. Side Information Fusion for Recommender Systems over Heterogeneous Information Network. ACM Trans. Knowl. Discov. Data 2021, 15, 1–32.

[15] Burke, R. Hybrid recommender systems: Survey and experiments. User Model. User-Adapt. Interact. 2002, 12, 331–370.

[16] I. R. Ward, J. Joyner, C. Lickfold, Y. Guo, and M. Bennamoun, "A practical tutorial on graph neural networks," 2020. [Online]. Available: https://arxiv.org/abs/2010.05234

[17] Hamilton, William L., Rex Ying, and Jure Leskovec, "Inductive representation learning on large graphs," (2017), Proceedings of the 31st International Conference on Neural Information Processing Systems.