Book Recommender System

By Kyle Dhauke

Under the supervision of

Frank C Langbein

Special thanks to my supervisor, Frank C Langbein, for providing invaluable advice and guidance during this dissertation.

1. Introduction

A world without recommender systems is one where users are easily susceptible to information overload. With the rapid growth of the internet, coupled with the exponential expansion of digital content, it is easy for consumers to be continuously bombarded with new information. For instance, take the prototypical use-case of a user, Jack, trying to buy a product at an e-commerce store. The homepage may list current bestsellers as well as suggestions for other products- although whether Jack may find these suggestions more useful than distracting may poise a different scenario entirely.

The main function of a recommender system is to prioritize bridging the gap between consumers and the vast repositories of information available to them. Effectively information-retrieval systems, they process data according to the user's interest and preferences, or the observed behaviour about an item. Going beyond traditional search engines, they can take into account various factors such as past purchases, browsing history, explicit feedback, social connections and demographic characteristics. Typically, this is conducted via leveraging advanced algorithms and data analysis techniques to effectively filter and curate all the option's a user may face.

Consequently, it comes to no surprise how business and digital platforms have realized the immense value of these systems in enhancing user engagement and revenue generation. It is estimated that Amazon's recommender systems drive anywhere from a quarter to a third of the choices their consumers make (Hosanagar, 2015). Similarly, Netflix famously hosted an open competition to see who could create a recommender system better than their current version, with a grand prize of US$1,100,000 (Bennet & Lanning, 2007). This recent popularity trend is largely due to these systems having the ability to improve customer satisfaction, facilitate targeted marketing and predict user-behaviour- making them indispensable tools in today's competitive market landscape.

Nonetheless, these systems are still fairly regarded in their early infancy. For an example, despite the winner of the Netflix competition besting the American media company's own algorithm by 10.06%, it was ultimately not used due to the costly engineering factors it would take to implement the new design (Casey Johnston, 2012). As research continues to be built upon, evidenced by the rapid growth of the annual Association for Computing Machinery (ACM) Recommender Systems conference, newer technologies are being used to solve previous problems and pave the way to a more personalized user experience.

The aim of this project is to develop a range of recommendation algorithms for books, as well as test/train these on existing datasets. By evaluating the strength and weaknesses of each recommendation algorithm, one can therefore draw conclusions on the nature of these systems and provide better context for their usage.

2. Background

Definition of a Recommender System

When briefly defining the purpose of a recommender system, one would not be wrong to define it as a subclass of an information retrieval system that provides personalized suggestions to a user. This idea can be traced back to Resnick and Varian's pioneering article (1997) on recommender systems, which states the following:

> *In a typical recommender system people provide recommendations as inputs, which the system then aggregates and directs to appropriate recipients. In some cases the primary transformation is in the aggregation; in others the system's value lies in its ability to make good matches between the recommenders and those seeking recommendations.*

Whilst setting a strong foundation, this definition fails to identify the highly nuanced and contextual environment a recommender system can be used in, one which can dictate its behaviour entirely. Kartik Hosanagar notes, when separating products in an e-commerce store by utilitarian products (serving some sort of functional purpose) and hedonic products (luxury items), the recommender systems have a far more significant impact on hedonic products (2023). Adomavicius and Tuzhilin (2005) attempted to provide a more mathematical approach, formalizing the definition as follows:

> *More formally, the recommendation problem can be formulated as follows: Let C be the set of all users and let S be the set of all possible items that can be recommended. Let u be a utility function that measures the usefulness of item s to user c, that is, u: C x S fiR, where R is a totally ordered set (for example, nonnegative integers or real numbers within a certain range). Then, for each user c ŒC, we want to choose such item s that maximizes the user's utility.*

This definition now offers a quantifiable optimization objective, where the producer of a recommender system has a clear goal of maximising the user's utility for each user c in C. Furthermore, it provides a clarity of variables and sets, delineating the scope and components of the recommendation system. Unfortunately, its rigidity in formulating the problem via calculating an item's utility completely ignores algorithms which may prioritize a user's profile- while it completely disregards the notion of a system encompassing groups of users. Instead Burke et al (2011) provides a more dynamic definition which this paper will be adopting:

> *A recommender system is personalized. The recommendations it produces are meant to optimize the experience of one user, not to represent group consensus for all.*
>
> *A recommender system is intended to help the user select among discrete options. Generally the items are already known in advance and not generated in a bespoke fashion.*

Personalization serves as a distinction from typical information retrieval systems, allowing the recommender system to tailor to the individual user's preferences, interests, and needs. This definition focuses on the discrete set of options a recommender system operates within, facilitating the decision-making process by

suggesting the most suitable choices from that set. In addition, this paper will slightly tweak Burke et al's original version, changing the following line:

"… The recommendations it produces are meant to optimize the experience of one user **or a group of users sharing similar characteristics,** not to represent group consensus for all."

By including the optimization of user experience for individuals who share similar characteristics, one can encompass a wide range of recommender systems that may not solely target individual end-users, but rather cater to specific groups.

Context of the Recommender System

The context of a recommender system can hugely affect the manner of which the system exerts it behaviour, its effectiveness as well as its relevant to the user. Context can be classified into two key dimensions: the application domain of the system and the knowledge source of the dataset utilized.

Figure 1: Illustration of the context of a recommender system.

Application Domain:

When building a recommendation system, it is pivotal to identify the common problem area one wishes to tackle before picking the technological approach. This is due to the strong influence the application domain exerts over the type of solution that can be successfully applied. There are a plethora of domains a recommendation system can target, but the most popular include E-government, E-learning, E-resource services, E-tourism, E-business and E-commerce (Lu et al., 2015). In deeper analysis, the characteristics of each domain, like the type of product and its persistence, can also have a big impact. For example, books can last for many years and are easily available for consumption, whilst mobile phones and cameras, in a technological domain, may become rapidly obsolete in the span of a few years. As such, where a book recommender may rely on a probabilistic-based model such as Naïve Bayes or a user-item matrix, an E-tourism domain may use a knowledge-based content model instead.

In the application domain, the focus lies on elicting the desired user response.. Should the consumers immediately be confronted with the highest rated products, or perhaps foster a more diverse, niche community of items? During one such research study (Hosanagar, 2015), it was found that users were more likely to elicit a greater response when recommended a low-rated product. Interestingly, this poses the question to the developer on whether they wish to recommended popular items, at the risk of propelling a rich-get-richer effect, else increase serendipity and diversity by suggesting low-rated products- albeit at the cost of potentially subtracting revenue from the most profitable items.

Knowledge Sources:

The second step to building a recommender system would be to examine where the system extracts its data. Knowledge sources can be classified into two types: implicit and explicit feedback (Loy, 2022.). Explicit feedback is considered to be direct and quantitative data collected from users (e.g. Amazon allowing users to rate purchased items on a scale of 1-10). Else, implicit feedback is collected indirectly from the user, acting as a proxy for user preference (e.g. Amazon items you may have clicked on versus items ignored). While explicit feedback is relatively rarer to obtain, it can obtain a mixture of quantitative and qualitative data that makes it ideal for content-based recommender systems. Alternatively, implicit feedback is extremely abundant, allowing one to tailor a system in real time.

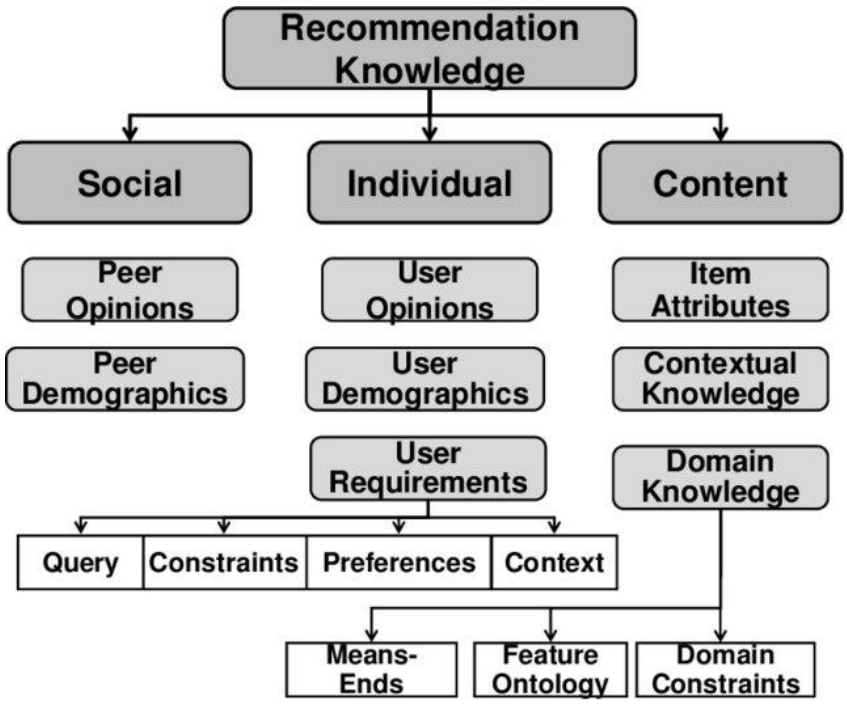

Figure 2: Knowledge Sources in Recommender Systems.

Although, certain types of recommendation systems can be designed to incorporate both implicit and explicit feedback. A hybrid collaborative filtering approach may analyse explicit feedback, such as ratings, reviews or explicit preferences provided by users, while also combining implicit feedback derived from user behaviour. In doing so, the system gains a more comprehensive understanding of user predisposition, helping it generate better choices.

Instead, Figure 2 offers another method to categorize knowledge sources- this time by creating a taxonomy of sources divided by the peer users of the system, the user, and from the item data itself (Felfernig et al., 2015, Figure 2). Social knowledge is typically manifested in numeric rating profiles, where the opinions of multiple users is used to weight items by the algorithm. A well-known example can be seen in the PageRank algorithm (Brin & Page, 1998). The relationship between individual and social knowledge can be purely reciprocal, as noted by Felfernig et al, 2015. A user rating is treated as individual knowledge when the system is providing a recommendation, else social when other peer users rely on it.

Lastly, content knowledge not only labels the attributes of the item inside the system, but also the complex relationship shared between each item and all the users. It is just as important to note how each item can

recognize the similarity between its neighbours as well as how a recommender can recognize these features. Overall, this newer taxonomy of knowledge offers several advantages over simply using implicit/explicit terms.

1. Comprehensive Coverage: The user, social and data categorization provides more coverage of the different type of knowledge sources available, going beyond the binary division of implicit and explicit feedback. It acknowledges the diverse range of information that can be leveraged for recommendations.
2. Reduced Reliance on Feedback Types: Implicit and explicit feedback can be valuable sources of user preference, however they may not encompass the entirety of user knowledge or provide a complete understanding of item characteristics. By considering user, social and data knowledge sources, the recommender systems can tap into a wider range of information and provide more accurate and diverse recommendations.
3. Future Expansion and Adaptability: As technology evolves and new types of knowledge becomes available, future-proofing concepts to be more dynamic and versatile can be incredibly vital. This classification framework accommodates the inclusion of additional categories or sources of knowledge without the constraints of the implicit-explicit feedback dichotomy.

Evaluation and Challenges of a Recommender System

Before exploring each recommender system, there must first be a closer look into the evaluation and challenges associated first. Recommender systems play a vital role in various domains by assisting users in finding relevant and personalized recommendations, however, undertaking this task is no easy feat. Certain algorithms fare slightly better than others when given specific conditions, hence it is important to assess their performance when addressing these inherent challenges. Firstly, a brief analysis will take place on the two key dimensions of a recommender; bias and variance.

It is practically impossible for a recommender system to be accurate 100% of the time, after all, humans are fairly unpredictable. This, combined with a large population, means errors will always persist; there will always be a difference between the model predictions and actual predictions. As such, it is up to the developer to attempt to reduce these errors where possible. Fortunately, there are two types of reducible errors: bias and variance.

Bias:

In the context of recommender systems, bias refers to the tendency of the system to consistently deviate from the preferences of users. It is calculated by the difference between prediction values made by the model and expected values. Therefore, it can be defined as an algorithms inability to capture the true relationship between data points.

- Low Bias: A low bias model will make fewer assumptions about the form of a target, accurately striving to capture the user's true preferences.
- High Bias: A model with a high bias will tend to make more assumptions, since it becomes unable to capture the important features represented in the dataset.

Variance:

On the other hand, variance represents the sensitivity of the recommender system to changes in the input data. It can be found from calculating how spread out or diverse a set of prediction values are from the expected value.

- Low variance: This means there is small variation in the prediction of the target function with changes in the training data set. Hence, when given new data to learn from, the model will be able to react appropriately, providing accurate answers.
- High variance: A high-variance system is overly sensitive to training data, resulting in overfitting. Overfitting occurs when a recommender system performs exceptionally well on the training data, but, fails to generalize well to unseen data. This can lead to poor performance on real-world tasks. This can be seen in figure 3, which illustrates the same model, albeit with training data and new data.

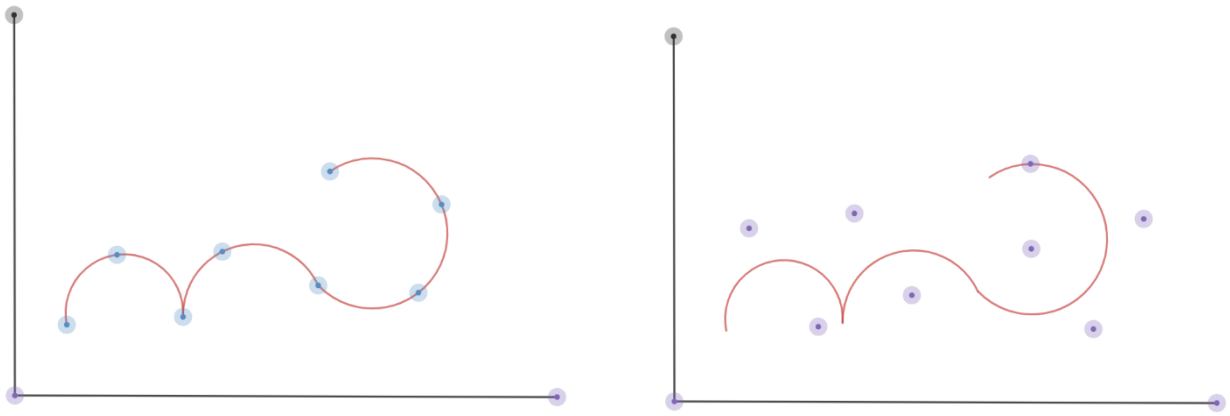

Figure 3: An over-fitted model (red line) where we see model performance on a) training data (blue dots) and b) new data (purple dots).

Bias-Variance Trade-off:

Bias-variance trade-off is a fundamental challenge in recommender systems. First formally introduced by Geman et al. (1992), it refers to the fact that when trying to make a statistical prediction, there is a trade-off between accuracy of the prediction and its precision. Else, more commonly stated as the balance between bias (opposite of accuracy) and variance (opposite of precision), as explained by Doroudi, 2020.

The problem boils down to such: Increasing the complexity of the recommendation model can reduce bias, but, may increase variance. This potentially leads to overfitting. Conversely, reducing complexity may infact decrease variance, albeit at the cost of introducing more bias. Essentially, an optimum must be discovered, which is often highly contextualized and based upon the needs of the recommendation system (context).

Although, achieving an appropriate bias-variance trade-off can be assisted via employing regularization techniques, such as incorporating user/item biases, utilizing ensemble methods, or applying cross-validation to fine-tune model the performance. Later in this paper, regularization techniques will be employed when visiting the final solutions.

Challenges in a recommender system

In this section, we will highlight the main challenges recommender systems typically face when executing their functions. The challenges encountered in the design and implementation of a system tend to be multifaceted, arising from various dimensions. These include, but are not limited to, data, user behaviour, system design, and evaluation. It can be difficult to quantitively assess the impact of these challenges, largely due to the system often grappling with sparse and incomplete data as well as a limited availability of explicit user feedback. Nonetheless, user satisfaction continues to be the most indicative measure of evaluation.

Cold-Start Problem: Perhaps the most well-known problem, the term 'cold-start' originates from the analogy with starting a vehicle engine. When the engine is cold, it requires additional effort and time to reach optimal operating conditions. Similarly, in the context of a recommender system, cold-start occurs when there is insufficient information or metadata available for the recommender to perform optimally. In terms of users, this may mean first-time accounts have little history available to base recommendations- whilst new items may not be appear as frequently as items that have existed longer/have had more user interactions.

Data Sparsity: Data sparsity occurs in large-scale data analysis, where expected values are missing in the dataset. This can be due to active users rating few items (due to the lack of incentives). Consequently, this reduces the recommendation accuracy.

Scalability: Scalability problems have been significantly more prevalent due to the fast growth of e-commerce sites (Fayyaz et al., 2020), as well as an exponential increase in the training data (especially for collaborative filtering methods). Thus, in this era of big data, improving the efficiency of how the system interacts with the database can prove highly beneficial in terms of performance saved.

Serendipity (Diversity): The notion of "serendipity" is used to express the tendency of a model to create a confinement area. Essentially, the recommender has a narrower selection of objects, where highly related niche items may be overlooked. A result of a lack in enforcing diversity, it is important for the system to balance exploration (by recommending lesser known items) against exploitation (making sure to take into account user preference).

Shilling attack problem: This is a malicious problem where a user fakes his identity and enters the system to give false item ratings (Roy & Dutta, 2022). Typically, this occurs when the malicious user wants to either increase or decrease some items' popularity, causing direct bias on the selected items. Unfortunately, shilling attacks can greatly reduce the reliability of the system.

Content-Based Recommendation System

Recommender systems are classified according to the approach/paradigm used for predicting preferences. There exists two major categories of methods: collaborative filtering (CF) methods and content based (CB) methods. To begin with, CB recommender systems will be explored.

In CB recommender systems, all the data items are collected into different item profiles based on their description or features, e.g. for a movie the features will be the movie director, actor, etc. Hence where the *'Content'* part of its name stems from, due to its reliance on item features as a knowledge source.

There are two techniques that have been used to generate recommendations. The first is a more traditional technique that generates recommendations heuristically using traditional information retrieval methods, such as cosine similarity measure. The second technique relies more on statistical learning and machine learning methods, building models that try to explain the observed user-item interactions.

TF-IDF and Cosine Similarity Measure:

When designing a traditional information retrieval method, there are two important sub problems, as outlined by Manjula & Chilambuchelvan, 2016. The first is finding a representation of the items and next is to recommend for unseen items.

An item can be represented by its most "important" words, ones with the highest Term Frequency-Inverse Document Frequency (TF-IDF). A numerical representation, TF (Term Frequency) assigns a value based on how frequently a term appears within a document, whereas IDF (Inverse Document Frequency) measures the rarity

of a term. When these two terms are combined, they can be used to calculate a weighted score that reflects the importance of a term in a specific document.

$\text{tf}(t, d)$ measures the associated of a term $"t"$ with respect to a given document d. A value of 0 can occur when the document does not contain the term, else non-zero otherwise. TF can be calculated by:

$$\text{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \epsilon d} f_{t',d}} \quad (1)$$

where $f_{t,d}$ is the raw count of term. The denominator is the total number of terms in document d (iterating with each occurrence of the same term separately).

The formula for IDF($"t"$) is given in Eq.2. IDF can reduce the scaling factor of term ($"t"$) in Eq.1 depending on how many times it is aggregated in each document.

$$IDF(t) = \frac{\log 1 + |\text{d}|}{|dt|} \quad (2)$$

Where, d is the document collection dt is the set of documents containing term t. Thus, when TF and IDF are combined together, they form the TF-IDF measure which is given in Eq.3:

$$\text{TF-IDF}(d, t) = \text{TF}(d, t) \cdot \text{IDF}(t) \quad (3)$$

Now that a suitable representation for each item has been found, the next step is to determine the similarity between each item. This can be achieved via cosine similarity, which is a measure of similarity between two vectors in a multi-dimensional space. For instance, consider two vectors, A and B, in a multi-dimensional space. Each dimension of the space will correspond to a feature or attribute of the vector.

The formula for cosine similarity can be found in Eq.4:

$$\text{cosine similarity} = \cos(\theta) = \frac{A \cdot B}{||A|| \, ||B||} \quad (4)$$

Here, $(A \cdot B)$ represent the dot product of vectors A and B, i.e. the similarity between the magnitude of vectors. $||A||$ and $||B||$ represent the Euclidian norms (length) of vectors A and B, respectively. Cosine similarity ranges between -1 and 1; -1 indicates completely opposite directions, 0 indicates identical directions, and values closer to 1 indicate higher similarity.

Statistical and Machine Learning Methods:

Content based methods cast recommendations problems into either a classification problem (predicting if a user “likes” an item) or into a regression problem (predicting the rating given by a user to an item).

Classification Problem

For the classification problem, a probabilistic method can be used if based on user features. An item-centred approach, a Bayesian classifier can be built via trying to predict the probability for each user to like an item. To

achieve this task, a Bayesian classifier will be trained that takes user features as inputs and outputs either "like" or "dislike" (Rocca, 2019). As such, Eq.5 is expressed into the Bayes formula:

$$\frac{\mathbb{P}_{item}(like|user\_features)}{\mathbb{P}_{item}(dislike|user\_features)} \quad (5)$$

Eq.5 displays the ration between the probability for a user with given features to like the considered item against its probability to dislike it. Eq.6 expresses such in the Bayes formula:

$$\frac{\mathbb{P}_{item}(like|user\_features)}{\mathbb{P}_{item}(dislike|user\_features)} = \frac{\mathbb{P}_{item}(user\_features|like) \cdot \mathbb{P}_{item}(like)}{\mathbb{P}_{item}(user\_features|dislike) \cdot \mathbb{P}_{item}(dislike)} \quad (6)$$

Where

$$\mathbb{P}_{item}(like) \text{ and } \mathbb{P}_{item}(dislike)(= 1 - \mathbb{P}_{item}(like))$$

are priors computed from the data whereas

$$\mathbb{P}_{item}(.|like) \text{ and } \mathbb{P}_{item}(.|dislike)$$

are likelihoods assumed to follow Gaussian distributions. Subsequently, various hypotheses can be made regarding the covariance matrices of these two likelihood distributions:

1. No assumption about the covariance matrixes: This means that no assumption is made regarding the relationship between the covariance matrices of the likelihood distributions for different classes.
2. Equality of matrices: This assumes that there is an assumption of shared covariance structure across classes.
3. Feature Independence: This hypotheses assumes the features used for classification are independent of each other within each class.

Consequently, various well-known item classification models, including Naïve Bayes, can be derived (such as quadratic discriminant analysis and linear discriminant analysis).

Regression Problem:

In contrast to the classification problem, one can consider a user-centred method when implementing our regression approach. Machine learning can be used to train a linear regression model that takes item features as inputs and outputs the rating for this item.

Let M denote the user-item interaction matrix. A matrix X can be constructed by stacking row vectors representing user coefficients to be learned, and a matrix Y by stacking row vectors representing given item features. For a specific user i, the aim is to learn the coefficients in X_i by solving the following optimisation problem (Rocca, 2019) shown in Eq.7:

$$X_i = \underset{X_i}{argmin} \frac{1}{2} \sum_{(i,j)\in E} [(X_i)(Y_j)^T - M_{ij}]^2 + \frac{\lambda}{2}(\sum_k (X_{ik})^2) \quad (7)$$

It is important to note that in Eq.7, the variable i is fixed. As a result, the first summation only considers (user,item) pairs that pertain to user i.

Collaborative Filtering Recommendation System

CF systems are considered the most prominent technique in recommendation systems (Schafer et al.2007). This algorithm creates a user taste model ("user-item interactions matrix") based on the user's history. By doing so, collaborative methods are able to use these past-item interactions to detect similar users and/or similar items, and subsequently make predictions based on these estimations.

CF filtering algorithms can further be divided into two sub-categories, each based essentially on their usage, or lack thereof, of a model. Memory-based approaches are primarily based on nearest neighbours search, while model based approaches assume an underlying "generative" model that explains user-items interactions.

Memory-based approaches:

Memory-based approaches can further be classified into two types (Omega & Hendry, 2021):

- **User-Based Filtering:** Represent users based on their interactions with items and evaluate similarity based on a user's "behaviour profile".
- **Item-Based Filtering:** Represent items based on positive interactions users had with them, evaluating the distance between those items.

Despite each type focusing on different areas, they both use similar functions to solve the recommendation problem. Manjula & Chilambuchelvan, 2016, state when building a memory-based CF, the following steps are used:

1. *Calculate the similarity or weight, w_{ij}, which reflects distance, correlation, or weight, between two users or 2 items, i and j.*
2. *Generate a prediction for the active user by taking the weighted average of all the ratings of the user or item on a definite item or user, or employing an easy weighted average.*

Therefore, in the context of building a top-N recommendation system, the objective is to identify the k-nearest neighbours. Once the similarities are computed, these neighbours are aggregated to obtain the top-N most frequently occurring items- which are then recommended. While user-based filtering calculates the similarity w_{uv}, between the users u and v, who have rated similar items, item-based filtering computes similarity between item I and j by figuring out users who have had positive experience with said items.

When working out the weight between users or items, one can use correlation-based similarity or vector cosine-based similarity.

Correlation-Based Similarity:

A Pearson correlation can be used to calculated the similarity between two users u and v, or between two items. Pearson correlation is a statistical measure that quantifies the linear relationship between two variables- assessing the degree to which two variables move together in a linear fashion. This type of similarity measure should be considered if the data is subject to grade-inflation (different users possibly using different rating scales).

The coefficient, often denoted as r, ranges from -1 to +1, where -1 indicates a perfect negative linear relationship whereas a +1 points to a positive linear relationship. A 0 indicates no linear relationship. For example, the Pearson correlation between item i and j is given in Eq.8.

$$w_{ij} = \frac{\sum_{u \epsilon I}(r_{ui} - \bar{r}_i)(r_{uj} - \bar{r}_j)}{\sqrt{\sum_{u \epsilon I}(r_{ui} - \bar{r}_i)^2 \sqrt{\sum_{u \epsilon I}(r_{uj} - \bar{r}_j)^2}}} \quad (8)$$

Where r_{ui} is the rating of user u on item i, which is the average rating of the i[th] by those users.

Vector Cosine-Based Similarity:

Vector cosine-based similarity is a measure that quantifies the similarity between two vectors by calculating the cosine of the angle between them. In particular, for items I and j (or users u and v), the cosine similarity is obtained by taking the "·" (dot) product of their respective vectors and dividing it by the product of their magnitudes. The resulting value represents the degree of similarity between the vectors. This type of measure should be considered if the data is sparse.

To compute the vector cosine for similarity for a collection of n items (or users), an n x n similarity matrix is constructed. Each element of the matrix corresponds to the cosine similarity between a pair of items (or users), hence by populating this matrix it becomes possible to obtain a comprehensive representation of the similarity among all items (or users).

For instance, if the vector $A = \{x1, y1\}$, vector $B = \{x2, y2\}$, then the vector cosine similarity between A and B can be given in Eq.9:

$$W_{A,B} = \cos\left(\vec{A} \cdot \overleftarrow{B}\right) = \frac{x1x2 + y1y2}{\sqrt{{x_1}^2 {y_1}^2}\sqrt{{x_2}^2 {y_2}^2}} \quad (9)$$

Model-based approach:

Model based collaborative approaches rely on user-item interactions information, assuming an underlying latent model can supposedly explain these interactions. In other words, it is assumed that users who agreed in the past (via interacting with the same item) will subsequently agree in the future.

Matrix Factorisation

The most famous and widely used model-based approach is Matrix Factorisation. Factorisation is the method of expressing an entity into a product of two smaller factors. Henceforth, matrix factorisation simplifies a big

rating matrix (UD) into smaller factors which retain the dependencies and properties of the original rating matrix (Thandapani, 2022).

The mathematic concept of matrix factorization can be defined as follows :

A set of Users (U), items (D), R size of $|U|$, and $|D|$. The matrix $|U| \cdot |D|$ includes all the ratings given by users, with the goal to discover **K** latent features (Chen, 2020).

Given the input of two matrices $P(|U| \cdot k)$ and $Q(|D| \cdot k)$, it would generate the product result R, as shown in Eq.10.

$$R \approx P \cdot Q^T = \hat{R} \tag{10}$$

Matrix P represents the relationship between a user and the features, while matrix Q represents the relationship between an item and the features. The prediction of an item's rating can be obtained by calculating the dot product of the corresponding vectors for u_i and d_j, as shown in Eq.11 below:

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj} \tag{11}$$

In order to obtain two entities from matrices P and Q, it is necessary to initialize the matrices and compute the product difference, referred to as matrix M. Subsequently, through iterations, the difference is minimized using a method known as gradient descent. The objective of gradient descent is to identify a local minimum for the difference. This can be expressed in Eq.12.

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 \tag{12}$$

The gradient is able to minimize the error, therefore we differentiate the above equation with respect to these two variables separately- as shown in Eq.13.

$$\frac{\vartheta}{\vartheta P_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})\ (q_{kj}) = -2 e_{ij} q_{kj}$$

$$\frac{\vartheta}{\vartheta q_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})\ (p_{kj}) = -2 e_{ij} p_{ik} \tag{13}$$

From the gradient, the mathematic formula can be updated for both p_{ik} and q_{kj}, as shown in Eq.14. α is the step to reach the minimum value while the gradient is calculated, for which it is usually set with a small value.

$$p'_{ik} = p_{ik} + \alpha \frac{\vartheta}{\vartheta P_{ik}} e_{ij}^2 = p_{ik} + 2ae_{ij}q_{kj}$$

$$q'_{ik} = q_{ik} + \alpha \frac{\vartheta}{\vartheta q_{ik}} e_{ij}^2 = q_{ik} + 2ae_{ij}p_{kj} \quad (14)$$

From the above equations, p'_{ik} and q'_{ik} can both be updated through multiple iterations, until the error finally converges to its minimum. This can be shown in the final equation, Eq.15.

$$E = \sum_{(u_i,d_j,r_{ij})\epsilon T} e_{ij} = \sum_{(u_i,d_j,r_{ij})\epsilon T} (r_{ij} - \sum_{k=1}^{K} p_{ik}q_{kj})^2 \quad (15)$$

Once the matrix has been factorized, the initial user-item interaction matrix is no longer required. Instead, an estimation of an items corresponding rating can be computed by using the dot product of the user and item matrix. As a result, this algorithm greatly reduces the space taken by the initial matrix (via reducing the dimensionality of the matrix)- improving the performance.

Rocca, 2019, makes an interesting observation on how the algorithm could be further improved upon with the added usage of memory-based techniques over this newer representation of the user-item interactions matrix. By applying approximate nearest neighbour searches over a reduced dimensionality, the algorithm becomes more tractable. Interestingly, this blend of different techniques allows the developer of the system to explore new ways in which to make the recommender system more personalized, effectively finding a healthy optimum between variance and bias without having to perilously chosen between either metric.

For example, in an article called “Logical Versus Analogical or Symbolic Versus Connectionist or Neat Versus Scruffy,” Minsky (1991) argued that:

> *neither purely connectionist nor purely symbolic systems seem to be able to support the sorts of intellectual performances we take for granted even in young children. . . . I’ll argue that the solution lies somewhere between these two extremes, and our problem will be to find out how to build a suitable bridge. (p. 37)*

This is but one way, out of many, for which a developer could balance bias and variance fluidly while designing a well-made system.

3. Approach

This dissertation will attempt to create three programs which, when given an inputted book title, which will generate a list of useful recommendations. The recommendations provided by each algorithm can then be evaluated against a list of books the same user may wish to read, providing insight on the performance metrics of said algorithm. The general idea is that the program would work as follows:

1. User types in a book title.
2. User picks which algorithm they would like to perform the recommendation.
    a. Additonal information can be inputted, such as the amount of recommendation a user wants returned.
3. The algorithm outputs a list of books which provide an estimate of what the user would like.

As researched, there are many different algorithms one could use to employ the same book recommendation task. In this paper, we will primarily explore a K-Nearest Neighbours algorithm (Item-Based filtering, content-based approach) as well as Matrix Factorisation algorithm (Model-based, collaborative-filtering approach).

Goodbooks – 10k Dataset

As with any data-analysis project, it is important to obtain a large and relevant dataset. The objective of this program is to take an input of a single book title, compare against a user-item matrix or a series of documents, before outputting a recommended list of items. Hence, it is vital the dataset contains a variety of ways to discover a user's specific interest (or lack thereof) in a book.

Fortunately, there was a dataset which had the necessary fundamentals as well as more features which I was able to wholly embrace. The Goodbooks – 10k dataset contains user ratings for 10,000 popular books. Upon conducting statistical analysis, there are around 100 reviews for each books, with ratings varying between 1-5.

This dataset also provided a suitable ID index, making it easier to identify and compare books in a selection. There are 53,424 users, all of which have made at least two ratings. The average number of ratings per user is 8.

Most importantly, is the "to_read" dataset, which contains 53.4k books users have listed as their to_read. This will be highly beneficial during the evaluation of each recommender algorithm, providing a hard-basis to compare whether a recommended item was successful (via checking against this dataset).

Implementation

Implementation will be examined backwards, beginning with the completed program, before then analysing each algorithm respectfully.

Main Function

T he program was implemented as a simple Main.py (Python) file that contains a command line interface. In order for the program to run, certain requirements need to be met. A requirements.txt was also provided to ensure the correct dependencies were installed:

- Goodbooks-10k dataset folder must be in the same directory as the main.py program.
- Pandas must be installed, due to the machine learning libraries used.
- Typer must be installed, due to the command line interface.

That being said, if the requirements are satisfied, the program can be run by executing main.py

Two commands can be given:

- Python main.py recommend-books
- Python main.py evaluate-recommender

The program itself is structed as follows, in listing 1:

```
import pandas as pd
import numpy as np
from generate_user import *
from book_recommender_1 import *
from book_recommender_2 import *
from knn_recommender import *
from evaluate import *
import typer
from typing import Optional

from typing import Optional
import typer

app = typer.Typer()

# Your function definitions here

@app.command()
def recommend_books(
    algorithm: str = typer.Option(None, prompt=True, help="Please choose a
recommender algorithm:"),
    book_title: str = typer.Option(..., prompt=True, help="Please enter a book
title:"),
    n_recommendations: int = typer.Option(None, prompt="Number of
recommendations to return:", show_default=True),
    n_comp: Optional[int] = typer.Option(None, prompt="Number of eigenvalues
to keep:", show_default=True),
):
    if algorithm == "mf_1":
        if n_comp is None or "":
            n_comp = 12
        recommended_books = matrix_factorisation_1(book_title,
n_recommendations=n_recommendations, n_comp=n_comp)
    elif algorithm == "mf_2":
        if n_comp is None or "":
            n_comp = 12
        recommended_books = matrix_factorisation_2(book_title,
n_recommendations=n_recommendations, n_comp=n_comp)
    elif algorithm == "knn":
        recommended_books = knn_popularity_recommender(book_title,
n_recommendations=n_recommendations)
```

```
    else:
        typer.echo("Invalid algorithm! Please select a valid recommender
algorithm from the options below: ")

        recommender_options = ["mf_1", "mf_2", "knn"]
        for option in recommender_options:
            typer.echo(option)

        return

    typer.echo(f"Recommended books based on {algorithm}:")
    for book in recommended_books:
        typer.echo(book)

@app.command()
def evaluate_recommender(
    algorithm: str = typer.Option(..., prompt=True, help="Choose a recommender
algorithm"),
    n_tests: int = typer.Option(..., prompt=True, help="Number of tests to
run:"),
    n_comp: Optional[int] = typer.Option(None, prompt="Number of eigenvalues
to keep:", show_default=False),
):
    if algorithm == "mf_1":
        recommender = 'b1_test'
        if n_comp is None:
            n_comp = 12
    elif algorithm == "knn":
        recommender = 'knn_test'
        n_comp = None

    elif algorithm == "mf_2":
        recommender = 'b2_test'
        if n_comp is None:
            n_comp = 12

    accuracy, precision, recall, f1_score = performance_metrics(n_tests,
recommender, n_comp=n_comp)

    typer.echo(f"Performance metrics for {algorithm}:")
    typer.echo(f"Accuracy: {accuracy}")
    typer.echo(f"Precision: {precision}")
    typer.echo(f"Recall: {recall}")
    typer.echo(f"F1 Score: {f1_score}")

if __name__ == "__main__":
    app()
```

Listing 1: Code used to execute the Typer CLI

If recommend-books command was chosen, the following will occur:

1. User will be prompted on an algorithm: knn, mf_1, mf_2 *(Case sensitive)*
2. User will be prompted for a Book Title
3. User will be prompted for the Number of recommendations to return
4. (Optional) If the user has chosen MF_1 or MF_2, this will decide the number of Eigen Values to keep. Else, for KNN, this option will not do anything.
5. The algorithm will parse the given inputs, running the selected algorithm and then returning a list of books.

KNN Algorithm:

An implementation of the KNN algorithm can be found in Listing 2 below:

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

def knn_popularity_recommender(book_title, n_recommendations=10):
    # Read books and ratings datasets
    books = pd.read_csv('Goodbooks-10k Dataset/books.csv', sep=',')
    books = books.iloc[:, :16]
    books = books.drop(columns=['title', 'best_book_id', 'work_id',
'books_count', 'isbn', 'isbn13','original_publication_year', 'language_code',
'work_ratings_count','work_text_reviews_count'])
    books = books.dropna()
    ratings = pd.read_csv('Goodbooks-10k Dataset/ratings.csv', sep=',')


    df1 = pd.merge(ratings, books, on="book_id")

    df= df1.drop_duplicates(['original_title'])  # Merge books and ratings
datasets


    # Get the original titles of the books
    original_titles = books['original_title'].values
    original_titles = [title for title in original_titles if isinstance(title,
str)]


    vectorizer = TfidfVectorizer(input='content')     # Initializes a TF-IDF
vectorizer!


    tfidf_matrix = vectorizer.fit_transform(original_titles)     # Compute the
TF-IDF vectors for the original book titles
```

```
    cosine_similarity_matrix = cosine_similarity(tfidf_matrix)   # Calculate
the cosine similarity matrix


    title = books['original_title'].values
    title_list = list(title)
    samia = title_list.index(book_title)

    corr_samia = cosine_similarity_matrix[samia]

    book_corr_list = list(zip(title_list,corr_samia)) # Zips the book
correlation coefficients to their respective books
    sorted_book_corr = sorted(book_corr_list, key = lambda x: x[1],reverse =
True) # Sorts the list from descending order

    recommended_books  = [] # Creates an empty list to store the recommended
books.
    for i in sorted_book_corr:
        x = i[0]
        if x != book_title:
            recommended_books.append(x)

    return (recommended_books[:n_recommendations])
```

Listing 2: Code used to execute the KNN Algorithm

Matrix Factorisation Algorithm:

An implementation of the Matrix Factorisation algorithm is displayed in the listing 3 below:

```
import pandas as pd
import numpy as np
import sklearn
from sklearn.decomposition import TruncatedSVD
import warnings
import time

# Function which uses basic matrix factorisation to recommended
n_recommendations amount of books. Optionally, you can choose
# the amount of eigenvalues to keep in our truncatedSVD. The higher the value,
the more biased (personalised).


def matrix_factorisation_1(book_title, n_recommendations = 10, n_comp = 12):
    """
        Uses basic matrix factorisiation to recommend n_recommendations amount
of books.

        Parameters:
```

```
            book_title (string): A string which contains a book title from the
Goodbooks-10k Dataset.
            n_recommendations (int): Integer which is the amount of
recommendations that are returned. Default is 10.
            n_comp (int): Integer value denoting the amount of eigenvalues to
keep in our truncatedSVD. The higher the value, the more
            biased the recommendations (personalised). Default is 12.

            Returns:
            recommended_books (list): A list of strings containing an
n_recommendations amount of books that were recommended.
    """
    books = pd.read_csv('Goodbooks-10k Dataset/books.csv', sep=',')
    books = books.iloc[:, :16] # Splices the first 16 columns.
    books = books.drop(columns=['title', 'best_book_id', 'work_id',
'books_count', 'isbn', 'isbn13',
'original_publication_year','language_code','work_ratings_count','work_text_re
views_count'])

    ratings = pd.read_csv('Goodbooks-10k Dataset/ratings.csv', sep=',')
    df = pd.merge(ratings, books, on="book_id")


    #Before duplicates were removed: (5976479, 8)
    df1= df.drop_duplicates(['original_title'])
    #After duplicates were removed: (5859358, 8)
    # 117,121 duplicates removed


    ### Matrix Factorisation ###
    books_matrix = df1.pivot_table(index = 'user_id', columns =
'original_title', values = 'rating', fill_value = 0.0)

    # Creating a training data set
    X = books_matrix.values.T # (9274, 3821). Transposes the books_matrix.

    #Fitting the Model
    SVD = TruncatedSVD(n_components=n_comp, random_state=0) # Variable to
decide the n_components used in our truncated SVD.
    matrix = SVD.fit_transform(X)

    var_explained = SVD.explained_variance_ratio_.sum() # Stores the
percentage of the variance between eigenvalues. Used in evaluation.
    # print(matrix.shape) #(9274, 12) for n_components = 12
    print("the current var= ", round(var_explained * 100,3)) #Percentage of
variance. 6.839% for n_components = 12


    import warnings
```

```
    warnings.filterwarnings("ignore",category =RuntimeWarning)#avoids
RuntimeWarning #Base class for warnings about dubious runtime behavior.
    corr = np.corrcoef(matrix)
    corr.shape


    title = books_matrix.columns
    title_list = list(title)
    samia = title_list.index(book_title)
    corr_samia  = corr[samia]

    book_corr_list = list(zip(title_list,corr_samia)) # Zips the book
correlation coefficients to their respective books
    sorted_book_corr = sorted(book_corr_list, key = lambda x: x[1],reverse =
True) # Sorts the list from descending order

    recommended_books  = [] # Creates an empty list to store the recommended
books.
    for i in sorted_book_corr:
        x = i[0]
        if x != book_title:
            recommended_books.append(x)

    return (recommended_books[:n_recommendations]) # Returns the
n_recommendations amount of books.
```

Listing 3: Code used to execute the Matrix Factorisation 1 algorithm.

Singular Value Decomposition (SVD) was used to decompose the matrix into three separate matrices: U, Σ, and V^T, where the U matrix represents the user-feature relationship, the Σ matrix containing the singular values, and the V^T matrix representing the item-feature relationship.

Although, in our case, the SVD was computationally expensive and memory-intensive, hence truncated SVD was used to additionally approximate the SVD. Truncated SVD generates the matrices with the specified number of columns, whereas SVD outputs n columns of matrices. Therefore, truncated DVD decreases the output, which works better for sparse matrices when trying to detect features output.

The amount of eigenvalues play a crucial role in truncated SVD because they determine the importance of each singular value in the decomposition. Eigenvalues represent the variance or energy captured by each singular value. Larger eigenvalues correspond to more significant features or latent factors, at the cost of increasing the dimensionality of the matrix.

In the code provided, the eigenvalue was able to be customized by user input.

Matrix Factorisation 2:

In 2nd matrix factorisation program was implemented, albeit, this time it would provide more advanced features. In particular, the algorithm will be upgraded via taking into account regularisation terms as well as user biases.

The regularization term was added to the objective function during the fitting of the matrix factorization model. It aims to prevent overfitting by discouraging overly complex or extreme solutions. In turn, this

promotes a more balanced representation of the data. In this case, L2 regularisation was used, encouraging smaller parameter values by penalizing their squared magnitudes.

Biases in the matrix factorization models capture the inherent biases or tendencies associated with users and items in a recommendation system. These biases typically represent systematic deviations in user/item preferences that are usually not accounted for by the latent factors alone.

User and item biases were included in the model, enabling matrix factorization to better account for the personalized preferences and global popularity effects.

An implementation of the Matrix Factorisation 2 algorithm is displayed in the listing 4 below:

```
import pandas as pd
import numpy as np
import sklearn
from sklearn.decomposition import TruncatedSVD
import warnings
from scipy.sparse import linalg
from sklearn.utils.extmath import randomized_svd

# Function which uses basic matrix factorisation to recommended
n_recommendations amount of books. Optionally, you can choose
# the amount of eigenvalues to keep in our truncatedSVD. The higher the value,
the more biased (personalised).


def matrix_factorisation_2(book_title, n_recommendations = 10, n_comp = 12):
    """
        Uses advanced matrix factorisiation to recommend n_recommendations
amount of books.
        Introduces L2 Regularisation and biases.

        Parameters:
            book_title (string): A string which contains a book title from the
Goodbooks-10k Dataset.
            n_recommendations (int): Integer which is the amount of
recommendations that are returned. Default is 10.
            n_comp (int): Integer value denoting the amount of eigenvalues to
keep in our truncatedSVD. The higher the value, the more
            biased the recommendations (personalised). Default is 12.

            Returns:
            recommended_books (list): A list of strings containing an
n_recommendations amount of books that were recommended.
    """
    books = pd.read_csv('Goodbooks-10k Dataset/books.csv', sep=',')
    books = books.iloc[:, :16] # Splices the first 16 columns.
    books = books.drop(columns=['title', 'best_book_id', 'work_id',
'books_count', 'isbn', 'isbn13',
```

```
'original_publication_year','language_code','work_ratings_count','work_text_re
views_count'])


    ratings = pd.read_csv('Goodbooks-10k Dataset/ratings.csv', sep=',')
    df = pd.merge(ratings, books, on="book_id")


    #Before duplicates were removed: (5976479, 8)
    df1= df.drop_duplicates(['original_title'])
    #After duplicates were removed: (5859358, 8)
    # 117,121 duplicates removed


    ### Matrix Factorisation ###
    books_matrix = df1.pivot_table(index = 'user_id', columns =
'original_title', values = 'rating', fill_value = 0.0)


   # Creating a training data set
    X = books_matrix.values.T

    # Mean and standard deviation of each column
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)

    # Applying z-score normalization to reduce the CPU load.
    X_normalized = (X - mean) / std

    # Calculate the truncated SVD with regularization
    U, Sigma, Vt = randomized_svd(X_normalized, n_components=n_comp)
    Sigma_reg = np.diag(Sigma)  # Diagonal matrix of singular values

    # Apply L2 regularization to the singular values
    lambda_reg = 0.01  # Regularization parameter
    Sigma_reg = np.sqrt(Sigma_reg ** 2 + lambda_reg)

    # Computes the factor matrices with regularization
    matrix = U.dot(Sigma_reg)
    Vt = Sigma_reg.dot(Vt)


    # Initialize biases
    user_biases = np.zeros(X_normalized.shape[0])
    item_biases = np.zeros(X_normalized.shape[1])

    # Fit the regularized model
    n_iterations = 12  # Number of iterations for optimization
    learning_rate = 0.001  # Learning rate for optimization
```

```
    for _ in range(n_iterations):
        for i in range(X_normalized.shape[0]):
            for j in range(X_normalized.shape[1]):
                if X_normalized[i, j] > 0:
                    prediction = np.dot(matrix[i, :], Vt[:, j]) + mean[j] +
user_biases[i] + item_biases[j]
                    error = X_normalized[i, j] - prediction

                    # Update factor matrices
                    matrix[i, :] += learning_rate * (error * Vt[:, j] -
lambda_reg * matrix[i, :])
                    Vt[:, j] += learning_rate * (error * matrix[i, :] -
lambda_reg * Vt[:, j])

                    # Update biases
                    user_biases[i] += learning_rate * (error - lambda_reg *
user_biases[i])
                    item_biases[j] += learning_rate * (error - lambda_reg *
item_biases[j])

    import warnings
    warnings.filterwarnings("ignore",category =RuntimeWarning) #avoids
RuntimeWarning #Base class for warnings about dubious runtime behavior.
    corr = np.corrcoef(matrix)
    corr.shape

    title = books_matrix.columns
    title_list = list(title)
    samia = title_list.index(book_title)
    corr_samia  = corr[samia]

    book_corr_list = list(zip(title_list,corr_samia)) # Zips the book
correlation coefficients to their respective books
    sorted_book_corr = sorted(book_corr_list, key = lambda x: x[1],reverse =
True) # Sorts the list from descending order

    recommended_books  = [] # Creates an empty list to store the recommended
books.
    for i in sorted_book_corr:
        x = i[0]
        if x != book_title:
            recommended_books.append(x)


    return (recommended_books[:n_recommendations]) # Returns the
n_recommendations amount of books.
```

Listing 4: Code used to execute the Matrix Factorisation 2 algorithm.

4. Results

Setting up the Testing Environment

Evaluation of the recommender system's effectiveness was conducted via creating a confusion matrix. A confusion matrix is an N x N matrix used for evaluating the performance of a classification model. Although, since the evaluated recommender systems were not strictly classification models, there falls the question of how one can obtain the necessary metrics in order to provide meaningful testing. It can be difficult to classify metrics within the problem setting due to the large amount of uncertainty within the given dataset.

For an example, if the evaluation was based upon positive/negative interactions with the user, how would this information be collected? User ratings may seem like a clear answer, although it cannot be assumed that a user's rating can appropriately identify whether an item is a satisfactory recommendation. Furthermore, due to being a type of explicit feedback, it is unlikely there will be enough user ratings in the dataset to accommodate such a broad test.

Fortunately, the Goodbooks 10k dataset also includes a to_read dataset, which outlines a list of books each user would like to read- thus providing a suitable data to compare each recommender against.

In this paper, each recommender system was tested 20 times. The eigenvalue truncation, the process of selecting a subset of eigenvalues from the larger user-interactions matrix, was set to a fixed rate of 436. This number was specifically chosen from a combination of CPU performance as well as maintaining a controlled amount of variance *(will be explained in more detail further on in the paper)*.

Each time a test was ran on a recommender system, the following statistical information was noted:

True Positive (TN): A book that the system recommended that is subsequently in the user's to read list. For example, if the system recommended "To Kill a Mockingbird" and the user has expressed interest in that book, then it would be a true positive. Each system aims to maximize the number of true positive predictions, as these are the books that the user is most likely to read and enjoy.

False positive (FP): This would count as a book that the system recommended, but the user is unfortunately not interested in. For example, if the system recommended "Pride and Prejudice" but the user did not express interest in that book, then it would be a false positive. Many false positive predictions may surface if it tends to recommend books that are not a good fit for the user.

False negative (FN): In this case, books that the system did not recommend but the user is interested in. For example, if the system did not recommend "The Hunger Games" but the user has expressed interest in that book, then it would be a false negative.

True negative (TN): A book that the system did not recommend and the user is not interested in. For example, if the system did not recommend "The Da Vinci Code" and the user has never expressed interest in that book, then it would be a true negative. For all intents and purposes, this metric will always be 0, due to the fact that each system will recommend far fewer books than they will not recommend.

From these four outcomes produced by the confusion matrix, the precision and recall of the system can be calculated.

Precision represents the fraction of relevant items among all the recommended items to a user, and can be calculated via Eq.16.

$$Precision = \frac{TP}{TP+FP} \quad (16)$$

Recall represents the number of relevant recommended items to the total number of items that should be recommended, as calculated in Eq.17.

$$Recall = \frac{TP}{TP+FN} \quad (17)$$

Additionally, using these values, we can estimate the accuracy of each recommender system. Accuracy represents the number of correctly classified data instances of the total number of data instances. Although, accuracy may not be a good measure if the dataset is not balanced (they're being an uneven distribution of positive-to-negative ratings), hence F-measure can also be utilized.

According to Fayyaz et al., 2020, F-measure can reflect the metric (precision or recall) that excels more than its counterpart. Essentially, it acts as the harmonic mean of precision and recall. It can computed in the following formula (Eq.18):

$$F1\ Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

The code used to obtain the results can be seen in listing 5.

```
import pandas as pd
import numpy as np
from generate_user import *
from book_recommender_1 import *
from book_recommender_2 import *
from knn_recommender import *

books = pd.read_csv('Goodbooks-10k Dataset/books.csv', sep = ',')
books = books.iloc[:, :16] # Splices the first 16 columns.
books = books.drop(columns=['title', 'best_book_id', 'work_id', 'books_count',
'isbn', 'isbn13',
'original_publication_year','language_code','work_ratings_count','work_text_re
views_count'])
books = books.drop_duplicates(['original_title'])

to_read = pd.read_csv('Goodbooks-10k Dataset/to_read.csv', sep = ',')

ratings = pd.read_csv('Goodbooks-10k Dataset/ratings.csv', sep = ',')
```

```
def performance_metrics(n_tests,test_type, n_recommendations = 10, n_comp = 12):
    """
    Picks a recommendation algorithm to test and returns performance metrics based on a calculated confusion matrix.

    Parameters:
        n_tests (int): The number of tests to be ran on the chosen algorithm.
        test_type (string): The type of recommendation algorithm that will be tested. Pick between [b1_test,]
        n_recommendations (int): The number of recommendations the algorithm will return. Default is 10.
        n_comp (int): Integer value denoting the amount of eigenvalues to keep when using an SVD method. Default is 12.

    Returns:
        accuracy [float]: A performance metric used to assess the accuracy of the recommendation algorithm.
        precision [float]: A performance metric used to assess the precision of the recommendation algorithm.
        recall [float]: A performance metric used to tell how many of the actual positive cases the algorithm could predict.
        f1_score [float]: A value denoting the harmonic mean of Precision and Recall.

    """

    performance_results = _confusion_matrix(n_tests, test_type, n_comp,n_recommendations)

    tp_total = performance_results[0]
    fp_total = performance_results[1]
    fn_total = performance_results[2]
    tn_total = performance_results[3]

    try:
        accuracy = (tp_total + tn_total)/((tp_total + fp_total + fn_total + tn_total))
    except ZeroDivisionError: #Deal with the error if we end up dividing with 0's.
        accuracy = 0

    try:
        precision = (tp_total) / (tp_total + fp_total)
    except ZeroDivisionError: #Deal with the error if we end up dividing with 0's.
        precision = 0
```

```
    try:
        recall = (tp_total) / (tp_total + fn_total)
    except ZeroDivisionError: #Deal with the error if we end up dividing with
0's.
        recall = 0

    try:
        f1_score = (2 / ((1/recall) + (1/precision)))
    except ZeroDivisionError: #Deal with the error if we end up dividing with
0's.
        f1_score = 0

    return accuracy, precision, recall, f1_score




def _confusion_matrix(n_tests, test_type, n_comp = 12, n_recommendations =
10):
    """Takes in the selected test type and runs it n_tests amount of times,
making sure the same user isn't picked more than once.
        Returns four confusion matrix values used to determine values in
performance_metrics().
    """
    valid_test = ["b1_test","b2_test","knn_test"]

    tp_total = 0
    fp_total = 0
    fn_total = 0
    tn_total = 0

    if n_tests < 1:
        raise ValueError("Minimum of 1 test required.")
    if test_type not in valid_test:
        raise ValueError("Test type must be one of %r." % valid_test)

    list_of_users = [] # Stores the list of users to be tested.
    i = 0
    while i < n_tests: #Creates x amount of unique users based on the amount
of tests neccessary.
        curr_user = _check_user_list(list_of_users)
        list_of_users.append(curr_user)
        i += 1

    if test_type == valid_test[0]: # Determines the recommendation algorithm
to be tested.
        for j in list_of_users: # For each user in the list of users, conduct
a test and sum up their confusion matrix values.
            test_results = _b1_test(j[0],j[1],n_comp,n_recommendations,j[2])
```

```
            tp_total += test_results[0]
            fp_total += test_results[1]
            fn_total += test_results[2]
            tn_total += test_results[3]

    elif test_type == valid_test[1]: # Determines the recommendation algorithm
to be tested.
        for j in list_of_users: # For each user in the list of users, conduct
a test and sum up their confusion matrix values.
            test_results = _b2_test(j[0],j[1],n_comp,n_recommendations,j[2])
            tp_total += test_results[0]
            fp_total += test_results[1]
            fn_total += test_results[2]
            tn_total += test_results[3]

    elif test_type == valid_test[2]: # Determines the recommendation algorithm
to be tested.
        for j in list_of_users: # For each user in the list of users, conduct
a test and sum up their confusion matrix values.
            test_results = _knn_test(j[0],j[1],n_comp,n_recommendations,j[2])
            tp_total += test_results[0]
            fp_total += test_results[1]
            fn_total += test_results[2]
            tn_total += test_results[3]


    return tp_total,fp_total,fn_total,tn_total


def _check_user_list(list_of_users): # Checks if a user is unique before
returning them.
    """Takes in a list of users and generates a new user id. Returns a user id
who is not in the given list."""
    curr_user = pick_test_user()
    if curr_user in list_of_users:
        _check_user_list(list_of_users)

    else:
        return curr_user

    return list_of_users, curr_user



# This will take in a random test user, test book title and id and run an
individual test.
# Outputs singular test values
def _b1_test(test_user, test_book_title, n_comp, n_recommendations = 10,
test_book_id = None):
```

```
    """Takes in a random test user and runs an individual book_recommender_1
algorithm. Returns the calculated confusion matrix value results."""

    recommended_books =
matrix_factorisation_1(test_book_title,n_recommendations,n_comp) #Stores the
results of the recommendation algorithm into a list.

    df = pd.DataFrame() # Creates an empty dataframe. This is used to just to
recover the book id's of the recommended books. More reliable to compare book
id's (int) than book titles (string).
    for i in recommended_books:
        curr_book_row = books[books['original_title'] == i]
        df = pd.concat([df,curr_book_row],ignore_index=True) #Stores the books
in a new dataframe which contains the book titles and book id's.

    recommended_books_list = df["book_id"].values.tolist() # Creates a new
list which solely stores the book id's.

    expected_books = to_read.loc[to_read['user_id'] == test_user].head(10) #
Stores the first 10 books test_user wants to read.
    expected_books_list = expected_books["book_id"].values.tolist()

    results =
_matrix_calc(expected_books_list,recommended_books_list,test_user) #Compares
the matrix calculations against
    tp = results[0]
    fp = results[1]
    fn= results[2]
    tn = results[3]

    return tp,fp,fn,tn


# This will take in a random test user, test book title and id and run an
individual test.
# Outputs singular test values
def _b2_test(test_user, test_book_title, n_comp, n_recommendations = 10,
test_book_id = None):
    """Takes in a random test user and runs an individual book_recommender_1
algorithm. Returns the calculated confusion matrix value results."""

    recommended_books =
matrix_factorisation_2(test_book_title,n_recommendations,n_comp) #Stores the
results of the recommendation algorithm into a list.

    df = pd.DataFrame() # Creates an empty dataframe. This is used to just to
recover the book id's of the recommended books. More reliable to compare book
id's (int) than book titles (string).
    for i in recommended_books:
```

```
        curr_book_row = books[books['original_title'] == i]
        df = pd.concat([df,curr_book_row],ignore_index=True) #Stores the books
in a new dataframe which contains the book titles and book id's.

    recommended_books_list = df["book_id"].values.tolist() # Creates a new
list which solely stores the book id's.

    expected_books = to_read.loc[to_read['user_id'] == test_user].head(10) #
Stores the first 10 books test_user wants to read.
    expected_books_list = expected_books["book_id"].values.tolist()

    results =
_matrix_calc(expected_books_list,recommended_books_list,test_user) #Compares
the matrix calculations against
    tp = results[0]
    fp = results[1]
    fn= results[2]
    tn = results[3]

    return tp,fp,fn,tn

def _knn_test(test_user, test_book_title, n_comp, n_recommendations = 10,
test_book_id = None):
    """Takes in a random test user and runs an individual book_recommender_1
algorithm. Returns the calculated confusion matrix value results."""

    recommended_books =
knn_popularity_recommender(test_book_title,n_recommendations) #Stores the
results of the recommendation algorithm into a list.

    df = pd.DataFrame() # Creates an empty dataframe. This is used to just to
recover the book id's of the recommended books. More reliable to compare book
id's (int) than book titles (string).
    for i in recommended_books:
        curr_book_row = books[books['original_title'] == i]
        df = pd.concat([df,curr_book_row],ignore_index=True) #Stores the books
in a new dataframe which contains the book titles and book id's.

    recommended_books_list = df["book_id"].values.tolist() # Creates a new
list which solely stores the book id's.

    expected_books = to_read.loc[to_read['user_id'] == test_user].head(10) #
Stores the first 10 books test_user wants to read.
    expected_books_list = expected_books["book_id"].values.tolist()

    results =
_matrix_calc(expected_books_list,recommended_books_list,test_user) #Compares
the matrix calculations against
```

```
    tp = results[0]
    fp = results[1]
    fn= results[2]
    tn = results[3]


    return tp,fp,fn,tn


def _matrix_calc (expected_books_list, recommended_books_list,test_user):
    """Takes in the expected_books list and the recommended_books list.
Iterates through each, comparing both lists."""
    tp = 0
    fp = 0
    fn = 0
    tn = 0

    for recommended_book in recommended_books_list: # For each book in the
recommended_book_list
        for expected_book in expected_books_list: # For each book in the
recommended_book_list is compared against every book in the expected_books
list.
            if expected_book == recommended_book: # If they match, we get a
true positive.
                tp += 1
                continue

            if expected_books_list.index(expected_book) ==
len(expected_books_list) - 1: #If we go through every book in the
expected_books_list and there is no match, that means the recommendation was
wrong.
                # fp += _fp_test(recommended_book, test_user) # [OLD] We
consider a false positive if we recommended a book that they expressedly
showed disinterest.
                fp += 1 # Always a false positive if our recommendation does
not match any of the expected books.

    fn = len(expected_books_list) - tp # These are items that the user wants
to read but were missed by the recommender
    return tp,fp,fn,tn
```

Listing 5: Code used to evaluate the recommender systems.

In order to generate a meaningful test, a user had to be selected from the dataset. The user would have to meet certain requirements in order to be used in the evaluation testing:

1. They must have at least 10 books in the to_read dataset.
2. Said user must have rated at least one book, which will then be passed as input into the selected recommender algorithm.

The generate user code can be found implanted in the listing 6 below:

```
import pandas as pd
import numpy as np
import random
from math import isnan


to_read_with_nan = pd.read_csv('Goodbooks-10k Dataset/to_read.csv', sep = ',')
to_read = to_read_with_nan.dropna()
max_users = to_read.max()[0] # 53424 users, however, in the to_read dataset
only 48871 users rated books.




def _check_user(x = 0): # Set default to 0 to avoid argument confusion.
    """ Recursively picks a random user_id (int) and checks if they want to
read at least 10 books from the
        Goodbooks-10k to_read dataset.

        Returns chosen_user (int), an id of a user.
    """
    random_user = random.randint(0, max_users) #Generates a random number,
capped at the maximum users.
    duplicate_count = to_read.pivot_table(index = ['user_id'], aggfunc
='size') #Creates a pivot table around users who have read multiple books.
    # On average, most users want to_read at least 18 books.
    chosen_user = x #Recursively stores the chosen_user

    if random_user in duplicate_count.index:
        if duplicate_count[random_user] <= 9:
            return _check_user(random_user) # Fail: If user has not read more
than 10 books, repeat.
        if duplicate_count[random_user] >= 10:
            chosen_user = random_user # Success: If user has read more than 10
books
    else:
        return _check_user(random_user) # Fail: If the user is not in the
index, repeat.

    return chosen_user
```

```
def pick_test_user():
    """
    Retrieves a suitable test_user id and returns data neccessary to test the
recommendation system.

    Returns:
        test_user (int): id of a chosen test user.
        test_book_title (string): A string containing the title one of the
test_user highest rated books.
        test_book_id (id): An integer id of one of the books which the
test_user has rated the highest.
        test_book_rating (id): An integer value between 1-5 denoting the
rating of the test book. Typically it is 5.
    """
    books = pd.read_csv('Goodbooks-10k Dataset/books.csv', sep=',')
    books = books.iloc[:, :16] # Splices the first 16 columns.
    books = books.drop(columns=['title', 'best_book_id', 'work_id',
'books_count', 'isbn', 'isbn13',
'original_publication_year','language_code','work_ratings_count','work_text_re
views_count'])
    books = books.dropna(subset=['original_title'])
    ratings = pd.read_csv('Goodbooks-10k Dataset/ratings.csv', sep=',')

    test_user = _check_user() # Picks a random user
    a = ratings.loc[ratings['user_id'] == test_user] # Isolates all the books
the test_user has rated.
    a = a.sort_values(by=['rating'], ascending=False)
    list_a = list(a.iloc[0]) # Stores the highest rated books into a list.
    test_book_id = list_a[1]

    book_info = books.loc[books['book_id'] == test_book_id] # Stores the
information of the tested book.

    try:
        test_book_title = book_info.values[0][3] # Stores the title of the
tested book.
        if pd.isna(test_book_title):
            raise IndexError("Book title is out of bounds")  # Raise an
exception if the book title is NaN or out of bounds
    except (IndexError, KeyError) as e:
        print("Exception:", str(e))
        return pick_test_user()  # Recursive call to pick a new test user

    test_book_rating = list_a[2]

    return test_user, test_book_title, test_book_id, test_book_rating
```

Listing 6: Code used to generate a user in order to provide reliable testing.

Results

The evaluated results were as follows:

KNN = K-Nearest Neighbours Popularity Recommender

MF 1 = Matrix Factorisation Algorithm 1

MF 2 = Matrix Factorisation Algorithm with Advanced Features

	KNN	MF 1	MF 2
Precision	0.00498	0.0196	0.0175
Recall	0.005	0.02	0.035
Accuracy	0.0025	0.01	0.0175
F-Measure	0.00499	0.0198	0.0345

Immediately, it becomes clear that this model was not very accurate. Although, for the simple purposes of recommended a book, it performs reasonably well. There are several key insights I was able to gather from my data:

- MF 1 has the highest precision, followed by MF 2 and KNN finishing last. This seems to indicate that the matrix factorisation method were better at capturing the personalized preferences of the characteristics of users and items, rather than relying on simpler information retrieval.
- For each recommender system, recall was always greater than precision. Hence, this seems to indicate that the algorithms were bias towards producing more false positives than false negatives. There are several reasons to explain this, such as the recommendation diversity for each algorithm being too wide, leading to a higher recall for relevant items, however less precision overall. Personally, I believe this is to do with the manner in which the test was conducted. By checking against a user's to read list, rather than creating a threshold for the books they rated, a scenario was created where the algorithm would prioritize capturing as many relevant items as possible rather than increasing its own precision.
- Similarly, the accuracy was shown to be more favourable towards the matrix factorisation algorithms, with the KNN algorithm having the lowest accuracy with a score of 0.0025.

Based on my results, I was able to conclude that the MF 2 (advanced matrix factorisation algorithm) performed the best among the three recommender systems. It continually held the highest precision, recall, accuracy, and F-Measure. When comparing the MF 2 algorithm against the MF 1, there was an approximately 75% increase in both precision and recall. These results can certainly be attributed to the regularization terms having been added, allowing the algorithm to capture more relevant items while preventing overfitting. The inclusion of bias terms also helps capture user and item biases, enhancing the ability of the recommender system to align to a user's preferences.

Fine-Tuning the Performance

Earlier in the evaluation, it was noted that the eigenvalue truncated was set to a fixed rate of 436. This number was not randomly generated, instead stemming from a series of tests which measured the algorithm's performance against my own computational performance. Unfortunately, the lack of a power CPU meant I was limited by my own equipment in being able to produce a higher dimensionality user-interactions matrix. In order to overcome this hurdle, I tested a range of eigenvalues in order to find a value which could compromise the amount of CPU necessary while also maintain a satisfactory amount of variance in the constructed user-item matrix.

Figure 5 presents a line graph of Eigen Values vs Mean Time Taken (for my laptop). Figure 6 similarly represents the same graph, albeit this time portraying the amount of variance each eigen value holds against mean time taken. A logarithmic scale was chosen to represent the eigen values, since they tend to follow a distribution where most values are concentrated towards one end (i.e., a power-law distribution). Therefore, a logarithmic scale was able to compress the values, emphasizing the difference between smaller values while still accommodating larger ones.

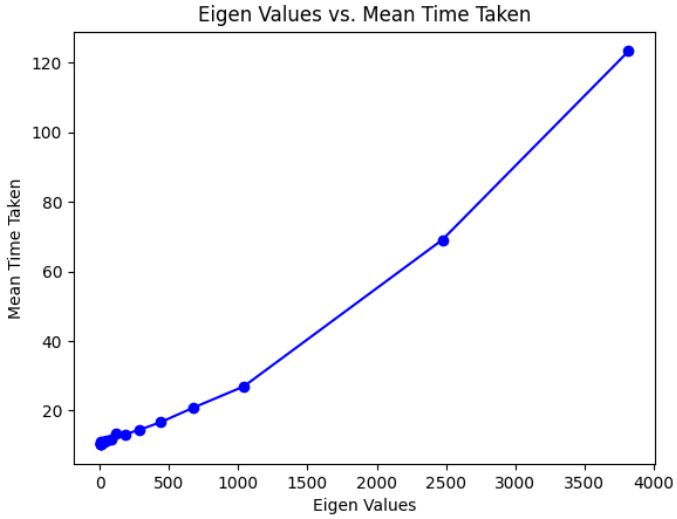

Figure 5: A line graph portraying 'Eigen Values vs. Mean Time Taken'.

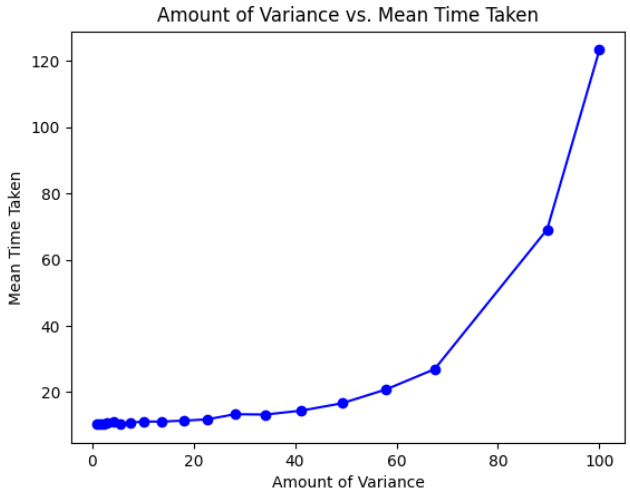

Figure 6: A line graph portraying' Amount of Variance vs Mean Time Taken'.

It is understood that the mean time taken is highly conditional on the user's equipment, albeit this was just a personal method to determine which eigen value I would use for my evaluation. The value 436 was chosen due to its seemingly lower computational cost (average: 16.626s) while holding a fairly large percentage of variance (49.253%).

Listing 7 provides the code (graph.py) that was used to calculate these values.

```
import matplotlib.pyplot as plt
from book_recommender_1 import *
import numpy as np


eigen_values = [1,2,3,4,6,9,14,21,32,50,77,119,183,282,436,673,1039,2475,3821]

var_value = [0.872,1.617, 2.315,2.988,4.163,5.578,
7.623,10.183,13.62,18.025,22.75,28.161,34.143,41.286,49.253,
57.951,67.52,89.685,100.0]

mean_times =
[10.265,10.238,10.471,10.582,10.97,10.493,10.781,11.117,11.099,11.369,11.776,1
3.329,13.217, 14.415,16.626,20.808,26.944,69.142,123.4]
# Plot the eigen values vs. mean time taken
plt.plot(var_value, mean_times, marker='o', linestyle='-', color='blue')

# Add labels and title to the graph
plt.xlabel('Amount of Variance')
plt.ylabel('Mean Time Taken')
plt.title('Amount of Variance vs. Mean Time Taken')

# Show the plot
plt.show()



num_values = 20  # Number of values desired
min_eigen = 1  # Minimum eigen value
max_eigen = 3821  # Maximum eigen value



def eigen_values():
    eigen_values = np.logspace(np.log10(min_eigen), np.log10(max_eigen),
num=num_values + 1).round()
    return eigen_values
```

Listing 7: Code used to generate matplotlib graphs of eigen values against mean time-taken.

References

Adomavicius, G. and Tuzhilin, A. (2005) 'Toward the next generation of Recommender Systems: A survey of the state-of-the-art and possible extensions', IEEE Transactions on Knowledge and Data Engineering, 17(6), pp. 734–749. doi:10.1109/tkde.2005.99.

Bennett, J. and Lanning, S. (2007) The Netflix Prize. Proceedings of KDD Cup and Workshop 2007

Brin, S. and Page, L. (1998) 'The anatomy of a large-scale hypertextual web search engine', Computer Networks and ISDN Systems, 30(1–7), pp. 107–117. doi:10.1016/s0169-7552(98)00110-x.

Burke, R., Felfernig, A. and Göker, M.H. (2011) 'Recommender Systems: An overview', AI Magazine, 32(3), pp. 13–18. doi:10.1609/aimag.v32i3.2361.

Casey Johnston, A.T. (2012) Netflix never used its $1 million algorithm due to engineering costs, Wired. Available at: https://www.wired.com/2012/04/netflix-prize-costs/#:~:text=8%3A20%20AM-,Netflix%20Never%20Used%20Its%20%241%20Million%20Algorithm%20Due%20To%20Engineering,recommendation%20engine%20by%2010%20percent. (Accessed: 10 March 2023).

Chen, D. (2020) Recommendation system - matrix factorization, Medium. Available at: https://towardsdatascience.com/recommendation-system-matrix-factorization-d61978660b4b (Accessed: 01 April 2023).

Doroudi, S. (2020) 'The bias-variance tradeoff: How data science can inform educational debates', AERA Open, 6(4), p. 233285842097720. doi:10.1177/2332858420977208.

Fayyaz, Z. et al. (2020) 'Recommendation systems: Algorithms, challenges, metrics, and business opportunities', Applied Sciences, 10(21), p. 7748. doi:10.3390/app10217748.

Felfernig, A. et al. (2015) 'Constraint-based Recommender Systems', Recommender Systems Handbook, pp. 161–190. doi:10.1007/978-1-4899-7637-6_5.

Geman, S., Bienenstock, E. and Doursat, R. (1992) 'Neural networks and the bias/variance dilemma', Neural Computation, 4(1), pp. 1–58. doi:10.1162/neco.1992.4.1.1.

Hosanagar, K. (2015) 'Recommended for you': How well does personalized marketing work?, Knowledge at Wharton. Available at: https://knowledge.wharton.upenn.edu/article/recommended-for-you-how-well-does-personalized-marketing-work/ (Accessed: 10 March 2023).

Loy, J. (2022) Deep Learning based Recommender Systems, Medium. Available at: https://towardsdatascience.com/deep-learning-based-recommender-systems-3d120201db7e (Accessed: 15 March 2023).

Lu, J. et al. (2015) 'Recommender System Application Developments: A survey', Decision Support Systems, 74, pp. 9–24. doi:10.1016/j.dss.2015.03.008.

Manjula, R. and Chilambuchelvan, A. (2016) 'Content Based Filtering Techniques in Recommendation System using user preferences.', International Journal of Innovations in Engineering and Technology (IJIET), 7(4), pp. 149–154.

Minsky, M. L. (1991). Logical Versus Analogical or Symbolic Versus Connectionist or Neat Versus Scruffy. AI Magazine, 12(2), 34. https://doi.org/10.1609/aimag.v12i2.894

Omega, C.Z. and Hendry (2021) 'Movie recommendation system using weighted average approach', 2021 2nd International Conference on Innovative and Creative Information Technology (ICITech), pp. 105–109. doi:10.1109/icitech50181.2021.9590147.

Resnick, P., and Varian, H. R. (1997). Recommender Systems. Communications of the ACM 40(3): pp. 56–58.

Rocca, B. (2019) Introduction to recommender systems, Medium. Available at: https://towardsdatascience.com/introduction-to-recommender-systems-6c66cf15ada (Accessed: 30 March 2023).

Roy, D. and Dutta, M. (2022) 'A systematic review and research perspective on Recommender Systems', Journal of Big Data, 9(1). doi:10.1186/s40537-022-00592-5.

Schafer, J.; Frankowski, D.; Herlocker, J.; and Sen, S. (2007) Collaborative Filtering Recommender Systems. In The Adaptive Web, ed. P. Brusilovsky, A. Kobsa, and W. Nejdl, 291–324. Lecture Notes in Computer Science 4321. Berlin: Springer.

Thandapani, S.P. (2022) Recommendation systems: Collaborative filtering using matrix factorization - simplified, Medium. Available at: https://medium.com/sfu-cspmp/recommendation-systems-collaborative-filtering-using-matrix-factorization-simplified-2118f4ef2cd3 (Accessed: 30 March 2023).