Charles University

Faculty of Mathematics and Physics

BACHELOR'S THESIS

Ondřej Novák

Framework Supporting Online Evaluation of Recommender Systems

Department of Software Engineering

Supervisor of the master thesis:	Mgr. Ladislav Peška, Ph.D.
Study programme:	Computer Science
Study branch:	IOI

Prague 2019

Title: Framework Supporting Online Evaluation of Recommender Systems

Author: Ondřej Novák

Department: Department of Software Engineering

Supervisor: Mgr. Ladislav Peška, Ph.D., Department of Software Engineering

Abstract: This work aims to highlight the importance of online evaluation for the testing of recommender systems. Firstly, we will look at the methods and the ways modern recommender systems operate. We will also introduce how they are compared both in online and offline settings. With this knowledge, we aim to build a .NET framework capable of tracking the various recommender systems for the purpose of measuring and comparing their performance during online use. To showcase the functionality of this framework, we use it to create a mock-up of an online movie database, where users can rate movies and receive movie recommendations.

Keywords: recommender systems, online evaluation

This work is dedicated to my family and friends both old and new for their support. It is also dedicated to my supervisor for the months of readiness to help and invaluable experience.

Contents

Introduction

In today's highly interconnected world users are faced with an endless number of various products (generally referred to as items). Be it electronics, home appliances, clothing, books, vacation destinations, ... Not only that, but video on demand services, music streaming services, news organizations, and social networks are also competing for the time and attention of users.

While trying to make sense of such a vast landscape of possibilities, it becomes impossible for any person to keep up. Getting familiar with all of the items is too time-consuming and thus, the user might miss some items that would be important or interesting to them. This is where recommender systems (hereafter also referred to as RSs) come in. They aim to address this issue by learning about users and their preferences and then try to find items that match these tastes.

Goals

In this work, we aim to familiarize the reader with basic concepts behind recommender systems with special emphasis on their evaluation. We highlight the importance of online evaluation as the most reliable but often overlooked way of estimating RS performance. We wish to encourage wider adoption of online evaluation especially for academic research by making it easy to compare recommender systems regardless of the used domain.

It is our task to create a .NET framework that allows deployment and concurrent use of different RSs. The ultimate purpose being the comparison of their quality in online use. This framework has to be able to deal with a diverse range of recommendation approaches and different types of feedback data available across domains. The framework will automatically keep track of recommendations provided by each system and will be able to recall them for evaluation. Common evaluation metrics will be provided for system evaluation.

Furthermore, we aim to make this framework lightweight and easy to use. We require the implementer to add only domain-specific functionality that cannot be generalized. This way, they can focus as much of their time and effort as possible on the system development itself. To showcase the usability of this framework, we present a web application built over the MovieLens dataset [1] with recommender systems provided by the MyMediaLite library [2].

Chapters overview

In chapter 1 we introduce recommender systems. We look at different types of recommender systems and how they operate. The feedback that systems receive from users is discussed right after. Lastly, we will show different approaches used to gather feedback and how we can measure performance from this feedback.

In chapter 2 we formulate our task of trying to create a framework capable of tracking recommender systems and users' interactions with items. We discuss the properties this framework should exhibit and how we can achieve them.

Chapter 3 describes which steps need to be taken by someone who wishes to implement our framework. In this part, we also detail the intended way of using this framework.

Chapter 4 describes how we used the framework, MovieLens dataset, and MyMediaLite library to create a mock-up of a movie database web application.

Finally, in chapter 5 we describe how the application from chapter 4 could be reused and adapted to a different (but similar) dataset.

1. About recommender systems

This chapter serves as a brief introduction to recommender systems. It describes issues RSs try to address, how they might be able to address them and how we can compare their performance.

1.1 What are recommender systems?

Before we try to explain how recommender systems function, it will be helpful to define the recommendation task more precisely.

We can imagine that we have a matrix M that represents preferences. More specifically, the matrix cell $M_{u,i}$ contains user u's preference of item i. This preference might be know from the user's previous interactions with this item (i.e. the user has rated the item). But since there is usually a low number of interactions between users and items, this value is most likely unknown and the resulting matrix M tends to be sparse.

The task of an RS is therefore clear: to fill in the missing matrix cells and predict users' preferences for each item. Once we can predict the ratings at least for some items, we can take the items with the highest predicted rating and present them to the user as potential items of interest.

If done correctly, these systems can introduce to users a wider range of items than the users might initially consider. They make it easier to find items a user is looking for, but cannot exactly identify. Most importantly of all, they can show items, which the user would find interesting, but isn't aware of their existence and therefore would never try to actively look for them.

Utilization of recommender systems can also be beneficial for companies and various service providers (most commonly e-commerce, social networks, news organizations, ...). If a user sees an interesting item, they are more likely to make a purchase. It is also because providing users with interesting items increases their trust towards the system and the service as a whole. If they find more value in the system, they are more likely to use it again.

For a comprehensive introduction recommender systems see [3].

1.1.1 Desirable traits of recommender systems

Let us have a closer look at some specific characteristics we would like a given RS to exhibit. It is important to note that in real-world situations not all of the listed traits are equally important (some might not be needed at all) [4]. Therefore the RS has to be chosen and adapted for a given domain.

Apart from prediction accuracy (described in section 1.5), the following properties might also be of interest to the RS provider [5] [6].

- **Coverage** is the basic ability to provide a recommendation. When we speak of item-space coverage, we mean the proportion of all items that can be recommended (to any user). User-space coverage, on the other hand, tells us for what proportion of users can the system provide a recommendation.

- **Novelty/Serendipity:** Novelty is the ability to recommend items that the user has not seen before. As serendipitous are considered those recommendations that are both novel and surprising (if the user is a fan of a certain director, recommending a movie by the same director is hardly surprising).
- **Risk management** can be an important factor with recommendations. One area where we can see this clearly is, if we're recommending a possible avenue of investment. Although to a lesser degree, we are undertaking risk with each recommendation we provide. If a user is exposed to items they don't like, it might lead to loss of trust in the system and user's alienation.
- **Diversity** is the ability to recommend a set of items that are not too similar (if someone likes the first Harry Potter book, recommending the other books in the series might be a safe bet, but exposing them to something different could lead to them discovering new interesting books). Besides, we can draw more useful information if the user interacts with dissimilar items.
- **Robustness** means that the system is not vulnerable towards outside attacks that try to influence which items get recommended (i.e. review bombing).
- **Adaptivity** is the ability to take in new information and use it for future recommendations. This is predominantly important in dynamic domains, for instance, the recommendation of news articles, where there are new items with short "shelf life" being added daily.
- **Scalability** means that the RS is able to deal well with growing datasets.
- **Privacy**: Users' data privacy plays a major role in determining their level of trust towards the system. Even after the user data has been anonymized, it might be possible to link a set of preferences to the real person they belong to. This famously led to the cancellation of the second Netflix Prize [7].
- **Fast response** is important since recommender systems have to be able to provide recommendations in real time (while the user is browsing the available catalog of items).
- **Interpretability** of the recommendation can help users understand how the system works and why it is that these particular items are being recommended to them. This can change how users feel about the recommended items and the system as a whole [8]. In reality, this can look like a note next to the recommendation saying: "Because you liked books in the genre of fantasy, you may like . . . " or "Because your friend Terrance likes . . . ".
- **Contextual awareness** is the ability to take the current context of the user into consideration. This could mean recommending only restaurants that are nearby user's location, only stores that are currently open, recommending based on user's search history, based on whether the user is alone or with a group of friends, ... This feature is becoming increasingly important as users spend more and more time on mobile devices.

1.1.2 Common challenges

Cold start

Among the most widely encountered obstacles when building recommender systems are so-called cold start issues. These arise when new users or items are introduced into the system. Since the user hasn't interacted with any items yet (or the item has not been interacted with), it becomes difficult to draw meaningful assumptions about preference [9].

To alleviate this problem, the system might at first recommend only the most popular items to new users, or ask them explicitly to share their preference towards some items. This is done in order to provide the system with at least some information about the user's tastes. Content-based recommender systems, which provide a natural remedy for cold start issues (from the perspective of items), are discussed later in this chapter (section 1.2.1).

Sparsity

Dealing with data sparsity is another common challenge. It is the result of large numbers of users and items as compared to the volume of available feedback. At this scale, it becomes unfeasible for every user to state how they feel about each item. In fact, quite the opposite tends to be true. The vast majority of users interact with a small subset of items (in the tens or even single digits). The MovieLens dataset (which we use for our framework's demonstration in section 4.1) has an average of 145 ratings per user. Since it contains more than 27,000 movies, this means that only about 0.5% of the rating matrix is filled. This dataset contains only users that rated at least 20 movies, so in reality the matrix would be even sparser.

This problem is most apparent in e-commerce websites, for which the only way to identify users is IP tracking or the use of cookies (if we're lucky, the user might register an account and be a returning customer).

Now combine this knowledge with the fact that most users (as far as they can be identified) visit a given website only once. This leaves recommender systems without a sufficient user profile.

Training

Training of recommender systems becomes an issue, particularly in online use. To take into account new information received while the system is operational, we have to retrain the model. Training the model from scratch usually takes too long and thus cannot be performed often.

One common approach is that the system performs a lot of small iterative retraining steps as it receives new feedback. This provides a fast way to adjust to new information. Over long periods of time, after many such small iterations have been performed, the predictive power of an RS starts to suffer. Therefore at some point, the entire system has to be retrained from scratch, with the data gathered since the last training. When the process is finished the older iterative version gets replaced with the newly trained system.

1.2 Types of recommender systems

In this section, we look at some currently used algorithms for providing recommendations. There are 3 main approaches to creating recommender systems: content-based, collaborative filtering, and hybrid.

1.2.1 Content-based

The working hypothesis of content-based recommendation is: if a user likes an item, then they will probably like other items similar to it.

To be able to recommend based on content, all we need is a way to describe some concrete properties of items (movie's genre, phone's screen size, ...) and a way to tell, which items are similar. We include a brief overview of cosine similarity and *tf-idf* as two methods for finding similar items.

This approach is mostly used for items since it is difficult to gain explicit characteristics of users (like age, socioeconomic status, ...) and these characteristics are likely to change with time.

Cosine similarity

Each item is represented as a vector of n features. The value at position i describes to what extent the item exemplifies the i-th feature. We consider two items similar if the cosine of their vectors is "close" to 1.

$$sim = cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

where A, B are feature vectors of the 2 items, θ is the angle between the two vectors

TF-IDF

Term Frequency-Inverse Document Frequency is a way to find frequently used words in a document while ignoring words that are commonly found across many documents.

tf-idf can be used for recommendation by, for instance, finding similar news articles or by comparing verbal descriptions of items.

$$tf(t,d) = \frac{f(t,d)}{|d|}$$

$$idf(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

$$tf\text{-}idf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

where $f(t,d)$ is the number of occurrences of term t in document d, D is the set of all documents

1.2.2 Collaborative filtering

The category of collaborative filtering includes all methods that try to look not only at the items and their similarity but also at broader sets of interactions between users and items. They look for patterns in user behavior and try to emulate them and recommend items based on them. This way we can find other users with similar tastes, or other items that are complementary to a given item.

Matrix factorization

As has been discussed before (section 1.1), we can represent the recommendation problem as a user-to-item preference matrix some of whose values are missing. We then try to fill in these missing values and show the items with the highest predicted ratings to users.

MF does this [10] by imagining the matrix of known preferences M as the combination of two other matrices $U \cdot I \approx M$. Each column of I represents some latent features the item exhibits and each row of U represents a user and their preferences for each feature. Once we have obtained U and I that ”fit” the known values of M, we can get an estimate of the unknown preference simply by multiplying the vectors for given user and item.

The main parameter that has to be optimized for matrix factorization training is the number of latent features (number of columns of U / rows of I). Training is usually done by gradient descent or some better-optimized version of it.

K-NN

Another popular method for finding similarities is the so-called k-nearest neighbors algorithm. We represent users or items as n-dimensional vectors of ratings (that the user has given each item or that the item has received from each user). Each of these vectors marks a point in n-dimensional space. For a given point (a user in user-based *k*-NN), we then look for other points that are nearby with the assumption that these points represent users with comparable tastes.

Which points should be considered ”close” to each other is not clearly set. Among the most used distance metrics are the aforementioned cosine similarity (see section 1.2.1), Pearson correlation, the Euclidean distance, and the Manhattan distance.

As the number of objects increases, so does the number of features. The nearest neighbor algorithm doesn’t scale well into highly dimensional vector spaces. This is because finding similar points becomes too computationally expensive. Some precomputation in the form of clustering can reduce the vector space of possible similar points and in turn shorten search times [11].

1.2.3 Hybrid

Hybrid recommender systems work by using multiple different approaches (such as collaborative filtering, content-based, knowledge-based, . . .) within one recommender system. By combining different kinds of recommender systems we might be able to alleviate some of their individual shortcomings [12].

For example, using content-based algorithms can provide a good starting point for new items and users. Once the objects have been interacted with and enough relationships have been revealed, we can use more powerful collaborative filtering methods. This way, it is possible to avoid the cold start problems for new items, while not being limited only to the knowledge we have about each item.

Hybridization can also be used to enhance the performance of one type of RS. Convolutional neural networks have been used as a way to provide additional item features [13]. More specifically, by using user-provided pictures about restaurants (of the building, location, food, and drinks) we are able to extract visual features representing a given restaurant. These features can then be added into item vectors for matrix factorization.

The potential usefulness of this approach is even more clear when we look at the world of online apparel shopping. Physical characteristics alone (size, materials, . . .) aren't enough to distinguish between relevant and irrelevant items. Since no person buys a piece of clothing without seeing it first, exploiting visual features becomes necessary for providing optimal recommendations [14].

1.3 Feedback

Before we can start to compare, which recommender system is the best, we need to know, what kind of information about their performance we are working with. This information (from now on referred to as feedback) is limited by the domain but also depends on how the user interacts with the system.

1.3.1 Explicit

The most valuable type of feedback is the so-called explicit feedback. It can be obtained, when a user consciously states their feelings about an item (giving it thumbs up/down or rating it on a scale of 1 to 10, . . .).

We can be reasonably sure that this method provides an accurate measure of user's sentiment at the time. But we should keep in mind that users' tastes change with time.

1.3.2 Implicit

In the majority of cases, we can only obtain implicit feedback. This term describes a broad range of data that can be used to estimate how the user feels about an item. In most cases, this data has to be aggregated and converted to a numerical value, which is easier for the RS to interpret.

Implicit feedback could be observations like:

- user clicked on an item
- user spend long time looking at the item's page (dwell time)
- user put a book in the ”want to read” category
- user went back to the item's page later
- user bought the item

During this process, we have to keep in mind that users have different motivations. Even something that seems like a clear sign of user's preference might not be useful. For instance, a user can purchase an item as a gift for someone else. We may then assume that the user was personally interested in the item, even though that might not be the case.

1.4 Methods of evaluation

In this section we present the 3 main approaches to RS evaluation.

1.4.1 User studies

The most direct way to gauge user's response towards recommendations is simply to ask. This is usually done by performing user studies, where volunteers are asked to interact with the system and try to emulate their real-world behavior. At the end of this process, participants are asked to fill in a questionnaire.

The advantage of this approach is that we're able to ask users about specific features of an RS. Since user studies are performed in a controlled environment, we can also measure physical responses that the user might not even be aware of. For instance, tracking where they look and for how long.

With this approach, on the other hand, we can never be sure about the intentions of users (what is their motivation for participating in the study) and how this affects their behavior. The main drawback is the time and effort necessary to execute these studies. It can also be difficult to convince a representative sample of users to participate.

1.4.2 Offline evaluation

This form of evaluation is performed on historical data. The data consists of information about how users interacted with the system, especially in terms of showing a preference towards given items (how they rated items, . . .). Some part of this data is then used to train the recommendation algorithm. The rest of the data is saved for later evaluation.

The main advantage of this method of testing is that it can be performed cheaply and is easy to repeat.

1.4.3 Online evaluation

For the purposes of this work, we will mainly be focusing on online evaluation. This method involves testing recommender systems against real users in a realistic environment.

This is usually achieved by A/B testing, where users get split into groups. Traffic from users of one group gets diverted to a given RS for some period of time (i.e. a session, a certain number of weeks/months, . . .). Some other set of users gets diverted to another system. Data is then collected over a long period of time. At the end of a given period (or during), the performance is compared among the various systems.

If our RS is working properly, we would expect it to influence user behavior. Online evaluation allows us to ask important questions not possible with the offline method like:

- did the user respond to the recommendation at all?
- did the user click on the recommendation?
- did the recommendation lead to a purchase?

We can also allow users to explicitly state how they feel about the recommendation (rate it, hide it, . . .).

One added bonus is that the online setting allows us to test other aspects connected to recommendation that affect the behavior of users but aren't the responsibility of the RS directly. This includes deciding when is the appropriate time to show a recommendation, how it should be presented and so on.

The main issue associated with online evaluation comes from the fact that they can be expensive. Both in terms of time and effort of implementation. Online evaluation has to be long-running so that a sufficiently large sample of feedback can be collected. Because these systems do not work with the same subset of users, small sample sizes may not accurately estimate the performance of the system.

Poorly performing RS could lead to the user developing distrust in the system, which in turn could lead to them using a given service less. One clear example of this would be an RS that recommends only those items that are outside of the user's price range.

Why online evaluation in particular

Offline experiments are the preferred choice for RS evaluation in academia. This is because of their ease and speed of execution, repeatability, negligible cost (compared to other methods). It may also be difficult to build an online service that enough people would use or obtain access to an existing one. Historical datasets, on the other hand, are publicly available for different domains. Under all of these criteria, offline beats online evaluation. The issue with offline evaluation is that it might not provide an accurate estimation of real-world performance [15].

The importance of online evaluation cannot be overstated, as it can help us measure one of the key aspects of recommendation performance; real human response. While user studies provide an insight into the reasoning behind users' actions, the mere fact that the users know they're being watched influences their behavior.

Thus, online evaluation becomes the best tool to approximate the real-world performance of recommender systems. Due to some of the associated drawbacks, care should still be taken in choosing online evaluation over its alternatives.

The resulting approach usually comes down to trying out a wide range of recommender systems using offline evaluation. This phase will weed out the worst performing systems. Once we have reduced the pool of candidates to a handful of comparable systems, they can be tested in the online setting, where the best one can be decided.

1.5 Common metrics

Now that we are familiar with how we can obtain the data for evaluation, let us look at specific measures of RS quality [6].

1.5.1 Rating based

Rating based methods require us to know user's preference towards an item expressed as a number. If we don't know user's actual preference, we have to be able to estimate it from implicit feedback. We also need RS to provide an estimated preference for each recommended item, instead of just a list of potentially interesting items.

This method is popular in settings, where we can gauge user's preference easily (i.e. from explicit ratings) and for performing offline evaluation.

Mean Absolute Error and Root Mean Square Error are among the most widely used metrics in this category.

$$MAE = \frac{\sum_{(u,i)\in F} |r_{u,i} - \hat{r}_{u,i}|}{|F|}$$

$$RMSE = \sqrt{\frac{\sum_{(u,i)\in F} (r_{u,i} - \hat{r}_{u,i})^2}{|F|}}$$

where $F = \{(u, i) \mid \text{user } u\text{'s preference for item } i \text{ is known}\}$, $r_{u,i}$ is user's actual preference for item, $\hat{r}_{u,i}$ is user's preference for item as expected by the RS

1.5.2 Ranking based

Modern services using recommender systems usually display recommendations as a list of items in a vertically or horizontally oriented banner at the edge of the screen. For these purposes, it is desirable to display the most interesting items in a place where the user is most likely to see them.

For this type of evaluation, we require an RS that returns items ordered according to expected preference.

When it comes to evaluation, some difficulty is also posed by trying to decide, what the "correct" ordering of items is. This is because the only practical way of finding out about user's preference is by gathering and interpreting their interactions with the items (gathering of feedback). Problems arise when the user does not interact with any item on the recommended list and simply ignores the recommendations altogether. Thus, getting to know the actual preference towards these items might not be possible.

For items that are not interacted with at all, we are not able to draw meaningful conclusions. They might have been ignored either because the user doesn't find them interesting or simply didn't even look at them. The following metrics are calculated only for the first p items, where we can be reasonably sure that the user saw them all.

Commonly used metrics in this category are Mean Reciprocal Rank (measures performance across all recommendation lists), Discounted Cumulative Gain, and Normalized Discounted Cumulative Gain (measure performance within one recommendation list).

$$MRR = \frac{1}{|R|} \sum_{i=1}^{|R|} \frac{1}{rank_i}$$

where R is the set of all lists that were recommended, each list ordered based on the expected preference of its items, $rank_i$ is the position of the first relevant item in the i-th list

$$DCG_p = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)}$$

where rel_i is the relevance of the item at position i

$$IDCG_p = \sum_{i=1}^{|REL|} \frac{rel_i}{\log_2(i+1)}$$

$$NDCG_p = \frac{DCG_p}{IDCG_p}$$

where REL are the items ordered in decreasing relevance up to position p

1.5.3 Binary classification

This set of metrics categorizes items either as interesting or uninteresting for each user. Unlike with some previously mentioned metrics, the extent of the interest is not considered. Simply said, binary classification makes no distinction between items that just barely interest the user and highly interesting items.

In the case of recommender systems, we receive predicted ratings for recommended items and showcase only the items we consider most relevant to the user (items with the highest predicted rating). Since the user cannot see all the items and showcase their interest in them, these metrics are usually evaluated only for the top k items that were presented to the user. These variations of classification metrics are commonly marked with '@k' (i.e. as *Accuracy@k*, *Recall@k*, ...).

Accuracy, Recall, Precision, and F-measure are the most popular metrics.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$F\text{-}measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

where TP: true positives are relevant items that were predicted as being relevant, FP: false positives are items that were wrongly predicted as being relevant, TN: true negatives are irrelevant items that were predicted as being irrelevant, FN: false negatives are items that were wrongly predicted as being irrelevant

1.5.4 Other

- **CTR**: The so-called click-through rate is a measure of user engagement with recommendations. It counts the proportion of displayed recommendations that were clicked on by the user (suggesting the user is interested in the recommended item)
- **Latency** says how fast an RS is able to provide a recommendation for a user. It is an important measure since we want to be able to provide recommendations in real time.
- **Diversity** of recommendations describes how different recommended items are from each other. If we have a way to measure the difference between two items, the diversity of a set can be computed as the average difference of all pairs within this set.
- **Provider's Utility** can differ wildly based on the intentions and goals of the recommendation provider. It can be defined as added profit, time spent using the service, conversion rate, ...

2. Framework architecture

In this section, we aim to take our previously gathered knowledge of recommender systems to create a framework capable of providing the necessary tools for online evaluation.

In this chapter class and property names are highlighted in **bold**.

2.1 Objectives

First and for most, our framework has to be able to deal with a wide range of domains, where recommender systems can be used. We achieve this by placing minimal requirements on how objects and items are should (section 2.3.1).

Adding to the idea of versatility, many of the classes are free to be extended to better suit domain needs. Our framework must also be prepared to handle different kinds of domain-specific feedback (section 2.3.4).

We also include an abstract idea of what a recommender system is and what it should be capable of. We try to take advantage of the online setting which allows us to pass the feedback back to the recommender so it can improve its future recommendations (sections 2.3.2, 2.4).

If we wish to evaluate recommender systems, we must retain some knowledge about their activity, about the recommendations they provide, and how users respond to these recommended items. This data collection process has to be simple in order to not slow down the recommendation process (section 2.5).

All of this data should be automatically stored to be available for future evaluation. We showcase how this can be done even for types unknown to us at the time of this framework's creation. Our method requires minimal effort from the implementer to achieve this (section 2.6).

We have to do all of the above in order to achieve our set out goal. The goal being the evaluation of the different recommender systems. We include several of the most popular evaluation metrics for recommender system performance. We make sure to include CTR measurement as a basic online-only evaluation metric (section 2.9).

Finally, to allow users and system administrator access to the recommendation service and data, we create a centralized point from which they can communicate with the RSs. We also take into account that recommender systems can evolve, be improved over time or their use can be discontinued (section 2.8).

2.2 Architecture overview

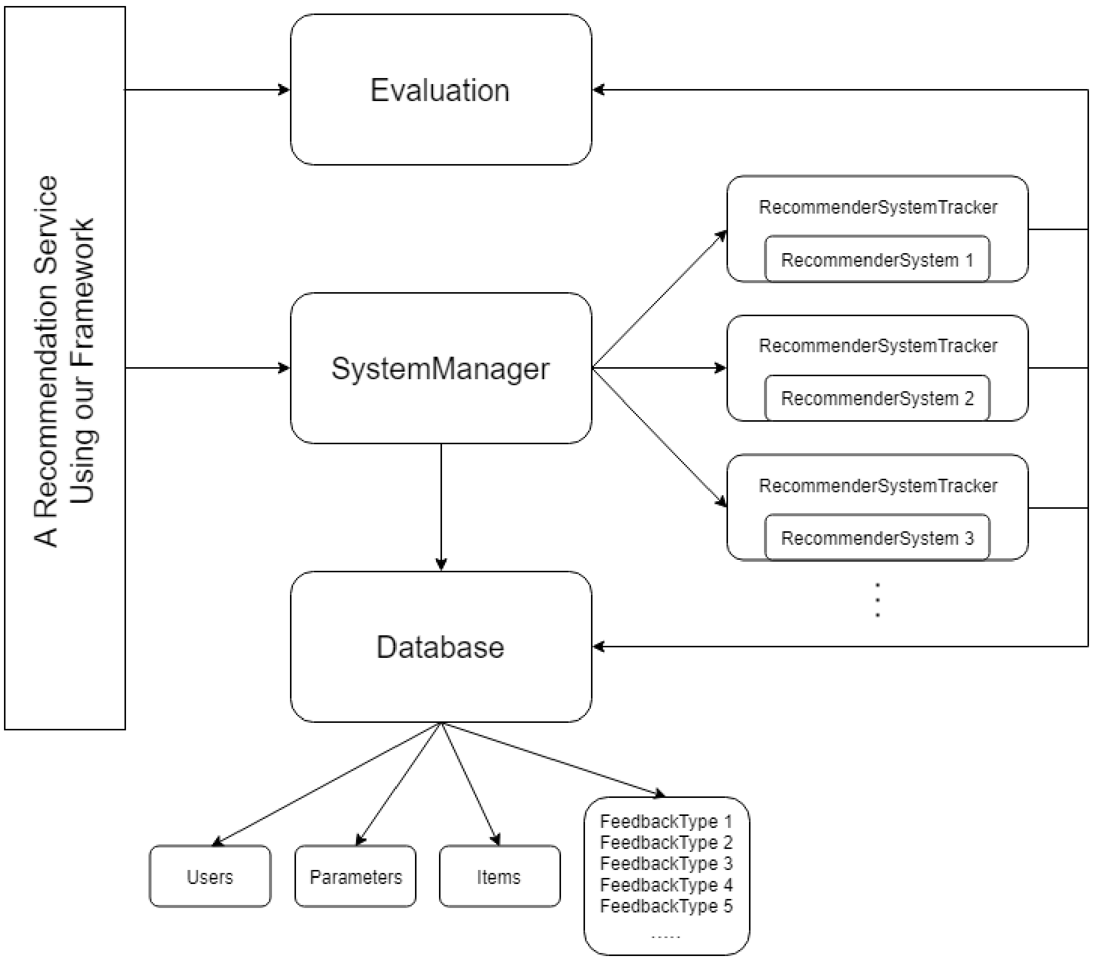


Figure 2.1: Framework architecture overview

This diagram shows what main objects and relations can be found within the framework. The system manager serves as a central point for communication with the "outside world". The manager has access to all of the recommender systems and the recommendation database. This database contains data about users, items, system parameters, and feedback. The last major section of the framework is the evaluation class. The system trackers, as their name suggests, are used to track recommender systems. They must reference the database and evaluation class for the purpose of recommender system evaluation.

Before we get to these important classes, let us start with the basics.

2.3 Data objects

This section describes the public data objects used by the framework.

2.3.1 User and Item

Our **User** and **Item** classes have a simple form. All they contain is a unique identification number.

We do not know before-hand which types of data about users and items the implementer has access to. These additional properties are dependent on the domain. It is required that the implementation derives its own user and item objects from these base classes.

The **Id** property (and by extension the class itself) is marked abstract as a reminder to the implementer that this property has to be specified.

2.3.2 Recommendation

Recommendation represents items that the system thinks will be interesting to a given user and were recommended to them.

Apart from the list of recommended items itself, this class also contains the time at which this recommendation was provided, how long did it take for the RS to provide it, and the identification of the user it belongs to.

We do not keep the information about which system provided this recommendation in the object itself. This is because recommendations are handled internally by the tracker RS (see 2.5). The tracker knows which system it belongs to. Meaning that we only need to connect recommendation to system identification when we wish to store recommendations in the database.

2.3.3 RecommendedItem

RecommendedItem is a class that contains the item itself and the expected preference provided by the RS.

An interesting problem is posed by deciding if the **RecommendedItem** should store a reference to the **Item** object or just its **Id**. For the purposes of recommendation it would be beneficial to have the reference to the item itself. This is because the implementer will most likely wish to showcase the recommended item to the user. To present it in an interesting way, we have to know something about the item. Thus, if we provide only the ID number, the next step would be recalling of the whole item object anyway.

While this logic is sound when it comes to recommendation, it only causes unnecessary overhead when it comes to evaluation. During evaluation we aren't interested in the actual properties of items, knowing their IDs is satisfactory. If we were to store the whole **Item** in the **RecommendedItem**, then requesting all system recommendations would also cause all items to be downloaded from the database, many of which multiple times (as they were recommended multiple times). As an example: we conducted a simple test, where we evaluated RS performance on about 220 recommendation lists containing about 2100 items. Removing the need to download the whole items individually made the calculation faster by a factor of 5.

The implementer may choose to add some information about each recommended item (maybe the RS provides some reasoning behind the recommendation or confidence). If they do, it is advised that they implement their own version of **IDatabase** (see 2.6.2) that is capable of saving and loading of this information.

2.3.4 Feedback

When it comes to **Feedback**, we are interested in 3 common properties for each type:

- Which user provided the feedback
- For which item they provided this feedback
- At what point in time was this feedback provided

  This will allow us to include temporal dynamics. For instance, we can filter out older feedback conflicting with more recent feedback (i.e. user changes a rating).

Each type of feedback is expected to derive from this class.

In addition, the classes for **ClickOnRecommendation** and **ExplicitFeedback** are provided. We can include these types in our framework because their form is uniform across domains. Even then, we do not seal these classes in case the implementer wishes to extend them with some additional information (i.e. the context of the feedback).

There are 2 approaches we consider for the form of implicit feedback. One would be to have only one implicit feedback class that would have some "type" property, through which we would decide what kind of feedback it is. But then we would also need a place to store some value of the feedback (i.e. the length of dwell time).

This would have to be an object. Such an approach would allow us to automatically store the feedback by serializing the object to a string. But it would then have to be parsed back by the implementer.

But this wouldn't make for good code practice. We instead choose to have Feedback split into multiple classes with one common ancestor. It is even possible for us to automatically store this kind of feedback in the database (see 2.7).

2.3.5 Helper objects

NameVersionPair

Used for identification of RSs and, by extension, of their trackers.

UserItemPair

Used as keys during evaluation. This struct serves mostly as an index of user's preference for an item. We choose to define it as a struct because we need to reference it only locally. Since we create many of these pairs during the evaluation, making **UserItemPair** a class instead would only result in unnecessary overhead.

UserItemPreference

Just like **UserItemPair**, **UserItemPreference** is used in evaluation. It stores the expected preference from a recommendation for a given **UserItemPair**.

2.4 Recommender interfaces

We look at the recommender system from 2 main points of view. As something that is capable of providing recommendations and as something that we want to manipulate and whose functionality we want to be able to tweak.

2.4.1 IRecommenderSystem

This interface provides all that is associated with the actual recommendation and thus would be expected of an RS.

This includes the ability to tell, whether the system can with confidence recommend a specific item or recommend something to a given user (allowing us to measure coverage).

Most importantly we want the system to be able to provide recommendations, these can come in 2 forms. We can either ask to explicitly sort a set of items according to the given user's expected preference. Or we can ask the system to recommend a list of items for a given user, be it from all items or some smaller prefiltered subset.

One other advantageous feature of an RS in the online setting is the ability to handle feedback and adapt its recommendations. This is something we cannot measure in offline evaluation and due to the small scale nature of them, probably not even in user studies. Therefore being able to handle user feedback is of the expected abilities.

We also include the **NameVersionPair** of the recommender system in this interface, as this allows us to identify a given recommender (and by proxy its tracker).

2.4.2 IManagedRecommenderSystem

Through this interface we can manipulate an RS.

IManagedRecommenderSystem expects a basic ability to load and save recommender system and its parameters, and to be able to change some of these parameters. This allows for continuous development and tweaking of various recommender systems and to be able to deactivate and replace systems altogether.

The clone option provides a way to easily adjust and test different versions of one RS simultaneously. This is another important feature used in the process of hyperparameter optimization.

2.5 RecommenderSystemTracker

Recording the activity of a recommender systems and being able to recall it for evaluation is the task of the **RecommenderSystemTracker** class (or RST for short). It serves as a wrapper for the recommender system.

When a user requests recommendation it goes through the RST first. The tracker calls the corresponding function of the RS contained within (to get the actual recommendation). This class measures the response time, saves the recommendation to the database and passes it back to the user.

When the tracker, on the other hand, receives some feedback from the user, it saves it into the database and passes it to the RS for handling.

The recommendation data is saved along with the name and the version of the RS, which allows us to easily retrieve it when we wish to evaluate the performance of this tracked system.

We do not save the feedback data along with the identification of the system. We don't do this because we want to use all of the available feedback for given user-item pair. Some of this feedback might have been obtained even before the current RS was in operation, or a different RS might have served the user in the past. Thus, we cannot tie the feedback to any given RS.

Evaluation methods contained within the tracker work by calling their equivalent counterparts in the **Evaluation** class (see 2.9). If it is required for the evaluation, we pull the recommendations of this RS from the database.

If the system consists of large amounts of data, or we wish to perform evaluation with multiple metrics at once, it might be advantageous to first retrieve the relevant recommendation data and feedback manually, and call the evaluation methods of the **Evaluation** class directly. This is because, for some RST methods, each time they are called, the recommendation data has to be pulled and converted to objects. Needless to say, this may become computationally expensive if repeated frequently.

2.6 Database

Creation of the database system proved to be one of the more challenging parts of the framework design.

Since we wish to make data querying as simple as possible, we use inbuilt ORM techniques. While ORM is easy enough for known types, it proves to be more of a challenge for classes we do not know at the time of this framework's creation.

Before we try to tackle these issues, let us have a look at how a database in the framework looks and how it is supposed to operate.

2.6.1 Why do we need a centralized database?

One straight forward approach to storing data would be to keep all recommendations and feedback in the RST they pass through. We wouldn't need to filter data based on which system it belongs to. This is because each system tracker would keep its own data and evaluation happens in the context of a single recommender system anyway.

This would pose a problem if we changed which RS serves a given user. We would then need to keep the provided recommendations in one RST, but copy the feedback we have obtained so far to the new RST. Since we want to use as much feedback for evaluation as possible we would also need to provide all new feedback for this user to the old RST.

Another advantage of having a database is that the data can be accessed easily by other interested parties. For example, the marketing department may wish to have access to the feedback data to see which items become more popular during the holiday season.

2.6.2 IDatabase

IDatabase represents the functionality expected of a recommender database. Its methods can be divided into 4 main areas. These allow for the saving and retrieving of:

- users and items
- feedback
- recommendation lists
- system parameters
  These parameters are added when the system is added to the manager

2.7 Default database implementation

Default implementation expects that the recommendation service is a part of a larger system that uses an SQL database for data storage.

For the object-relational mapping of recommendations, we have created separate mapped classes. For other data types (**User**, **Item**, **Feedback**, ...), we use exactly those classes for the mapping.

When it comes to the **Recommendation** class, we keep the list of items directly in it as a property. Due to this immediate one-to-many nature of recommendations, we split this data into 2 tables to avoid redundancy in the database and join them again when we need to retrieve the full recommendations from the database.

2.7.1 DbRecommendation and DbRecommendedItem

These classes represent a database mapping of the **Recommendation** and the **RecommendedItem** classes. **DbRecommendation** is different from its non-database version in the fact that it has a unique ID number and doesn't directly contain the recommended items. **DbRecommendedItem** is for the table containing recommended items. It contains an ID number that serves as a foreign key that links it to the recommendation it belongs to.

2.7.2 DbSystemParameters

This class stores parameters of a recommender system at the time it was added to the system manager.

Storing of parameters is important since we expect multiple RSs of the same type with different parameters to operate concurrently.

We can also use the database to recall the parameters of inactive systems. Knowing the performance of a system that hasn't been used for a long time is useless if we do not know under which parameters it operated.

2.7.3 ORM

In order to make data querying as simple as possible for the implementer, we wish to use one of the object-to-database mapping frameworks inbuilt into C#. For this purpose we use System.Data.Linq.Mapping which allows us to write SQL-like queries right in the C# source code.

Other methods are, of course, possible. But those usually require the programmer to write/build a query/command string explicitly using the SQL syntax. The query is then sent to the database. With this method, we're prone to making syntax errors. Linq queries, on the other hand, are compiled and syntactically checked together with C# code compilation.

For a more comprehensive overview of the Data.Linq.Mapping framework see the official documentation [16].

How Linq.Mapping works

Mapping is done simply by annotating classes and their properties.

The object gets mapped to a table of the same name (unless it is explicitly stated otherwise by the annotation). Annotated properties represent some attribute of the table.

To connect to the database one simply needs to create a class that contains properties of type **Table<T>**, where **T** is a type representing a mapped object (table row). This class also needs to inherit from the **DataContext** class. We can then simply query or add C# objects of type **T** to these **Table<T>** properties. The conversion to database types is handled by the framework.

For our framework to be able to save and retrieve implementer-created classes, all we need is for the new types to be properly annotated. There too need to be tables corresponding to these annotated classes in the database.

Ideally, we would be able to create the tables just from the annotations too. Unfortunately, this is not possible within the Linq.Mapping framework itself. It might be possible for us to create a custom solution on these new types by reading the annotations ourselves. But the creation of such a tool is beyond the scope of this work.

Linq mapping advantages

Linq mapping has several implementation perks.

The main advantage is being able to refer to database tables and their rows as C# objects. The actual communication when retrieving or inserting data is handled by the framework.

If we have a database property that is auto-generated by the database (i.e. an ID number), then this value is not known before insertion. This mapping framework ensures that after such object is inserted (and property generated) the corresponding property is also set in our C# object. We use this when we want to insert a recommendation list into the database. We first insert a **DbRecommendation** object, whose ID is auto-generated. After the insertion, we can also insert all the recommended items from this recommendation as they refer to the generated ID.

Linq mapping issues and solutions

For the above-described process to work, we need to know the object type **T** at compile time in order to create a property of type **Table<T>**. This works well for the types we define ourselves in the framework (**DbRecommendation**, **DbRecommendedItem**, **ExplicitFeedback**, . . .). For user and item objects, we do not know their exact form. But by making our database generic (for the user and item type), we can ensure that we know their types and we can create these tables for them.

Problems come when we want to be able to save and load domain-specific feedback. All we know about it is that it derives our Feedback class. Linq mapping doesn't allow us to use a parent table to query a child (i.e. table of type **Feedback** cannot be used to retrieve or save **DwellTime**).

We also cannot make our database generic when it comes to feedback. Having a variable number of generic class parameters is not possible. This means that we would have to create a generic database with many different type parameters for each feedback type. Since there could be dozens of types of feedback that are being collected in a given domain, this solution would be impractical.

We could also make it a requirement that the implementer of the framework has to take care of the saving process when it comes to feedback. But this would go against our philosophy of trying to handle most of the grunt work for the implementer.

FeedbackType

To resolve this we will introduce a new generic class called **FeedbackType<T>**. Each instance of this class will store one type of domain-specific feedback. This means that at compile time we have an instance of **FeedbackType<T>**, which means we know the type **0**T and can create a property of type **Table<T>**. Thus, we were able to solve the issue of how to store types unknown to us.

GenericContext

We use the generic context for retrieval of data from the database. Given a connections string, this class lets us query a table of a given type. It is used mainly for feedback retrieval since we do not know all the feedback classes before-hand and cannot create a specific context like in the case of **DbRecommendationContext**.

2.8 SystemManager

SystemManager serves as a way to keep track of all the active recommender systems within the service.

When a system is added, it gets wrapped in a tracker and saved, so it can be accessed later for manipulation or evaluation. During this process, the system parameters are stored in the database along with the system identification.

Its other main task is in knowing which systems is currently serving which user. Before a user requests a recommendation (i.e. at the start of a session), it is expected that the user is registered to the **SystemManager**. As a result, the manager returns a (tracked) system that will be providing the user with recommendations. This system is decided by a method we call assigner. When the session ends, the user should be de-registered from the system.

In some settings, the user's assigned system may change for another system within one session. In these cases, it is advised that each time a recommendation is about to be requested the manager is asked for the current providing system. This way we avoid situations, where an old system that has been removed is still providing recommendations.

Deactivating system with active users could lead to the above-mentioned problem as well. It is desirable to assign these users to a different system before deactivation. The proper way of doing this is to first move all of the users currently served by the system (be it manually or with the provided method) to a new system. The second part is ensuring that no new users will be assigned to the old system after deactivation. The latter is automatically handled by the default assigner because it selects a random system only from the currently active systems. For a custom assigner with a more sophisticated assignment policy, this has to be ensured by the implementer.

Assigning of recommender systems to users is by default random, with equal probability distribution across all systems in the manager. This behavior can be customized by providing own assigner implementation at the time of manager creation or at some point later during its use.

The manager performs the above-mentioned tasks in a thread-safe manner.

2.9 Evaluation

Evaluation contains static methods for estimating recommender performance.

Some of these methods require the use of additional functionality added by the implementer of this framework. This includes methods that are meant to answer questions such as "given this feedback, does the user find this item relevant?". Since this question is highly dependent on the type of feedback collected, it cannot be answered by us and is left to somebody with domain-specific knowledge (i.e. the implementer of this framework)

Formal definitions of some of these metrics can be found in section 1.5. For easier orientation, we split evaluation methods into several categories.

We also note that unit tests of these evaluation methods are provided in the RecommenderFramework solution.

2.9.1 Rating based

The two metrics implemented in this category are Mean Absolute Error and Root Mean Square Error. They take a list of recommendations and known preferences for the user-item pairs in the recommendations.

These metrics are computed over all recommended items for which we know the actual user preference (or for which the preference can be inferred from feedback). Ergo we treat recommended items with unknown preference as if they weren't recommended at all.

2.9.2 Ranking based

The metrics implemented in this category are Mean Reciprocal Rank, Average Discounted Cumulative Gain, and Average Normalized Discounted Cumulative Gain. All of them work with an ordered list of recommended items. We calculate these metrics for each one of the recommended lists and average the results.

For MRR we also need to be able to tell which item is interesting for the user. If no item on the list is interesting to the user, we award this list a score of 0.

For Average DCG and Average NDCG, we require a set of actual preferences and a rank at which they should be calculated. The result is the average over DCGs (NDCGs) of all recommendations.

If no feedback is gathered about a certain item recommended to a user, we give this item the relevance score of 0. For the calculation of Normalized DCG, we need to know the score of an ideally ordered list of items (so-called IDCG). If no item on this list has feedback from which we can draw preference, this IDCG score is equal to zero and the normalized score DCG / IDCG equal to infinity. Therefore we have to leave out those lists of recommended items whose IDCG is zero.

2.9.3 Latency

When it comes to latency, we measure the mean and the median response time of a set of recommendations.

2.9.4 Binary classification

The metrics implemented in this category are Recall, Accuracy, Precision, and Mean Average Precision.

Apart from recommendations, recall and accuracy require a way to tell which items the user finds relevant. Both are calculated for each user and then averaged.

Precision and mean average precision instead just require a way to say whether a given item is relevant to the user. Precision is calculated simply as a proportion of relevant recommended items over the total number of recommended items. Mean average precision is the precision for each user averaged.

2.9.5 Other

The metrics implemented in this category are Click-Through Rate, User Coverage, Item Coverage, and Diversity.

CTR is calculated from recommendations and relevant feedback simply by comparing the sizes of two sets. One set are the user-item pairs, where the user clicked on the recommended item. The other set contains user-item pairs, where the item was recommended to the user.

User (item) coverage is calculated as the proportion of users (items) for which the system can recommend an item (which the system can recommend to a user).

For diversity calculation, we need a set of recommendations and a way to tell just how dissimilar two items are. Diversity of one recommendation list is the average diversity of each pair of items in the list. The total diversity is calculated as the average of each list's diversity.

3. Usage requirements

This chapter lists the requirements and steps needed to complete for proper implementation of this framework.

3.1 Definition and loading of data objects

Users in the framework implementation must derive from the prepared User class, items must derive from the Item class. These classes implicitly contain only identification number.

Similarly, domain-specific feedback types must also inherit from the prepared abstract class Feedback. All feedback contains user and item identification, and the time at which this feedback was collected.

For the default database implementation, user ID, item ID and the timestamp together form the identification within the feedback table. Meaning if a user gave feedback to some item, all other feedback of the same type between these two must have a different timestamp.

3.2 Database setup

The implementer may choose to utilize our default database implementation or to create their own implementation of the IDatabase interface. The following paragraphs apply only to the default implementation.

For the purposes of data storage, it is required that all of the newly defined classes (User, Item and Feedback types) are correctly annotated using Data.Linq.Mapping tags. For details, see the official documentation [16].

Apart from the connection string, our database implementation takes a list of generic FeedbackType<T> classes. Their type argument is a newly defined feedback class that the system will be using.

It must be also ensured that these classes (for users, items, and feedback) have tables corresponding to the annotations in the SQL database.

3.3 Implementation of recommender systems

Once these data objects have been specified, recommender systems can be added. This simply requires that the used RSs implements the IManagedRecommender-System interface.

3.4 Creation of system manager

Next step is creating an instance of the SystemManager. Its constructor requires only some database implementation and a function that aggregates user-item feedback into numerical preference.

Optionally, a custom system assigner can be provided to the manager.

The Recommender systems that are ready for use can be added to the manager next. These systems will be wrapped in a tracker that will follow their activity.

3.5 Usage

The basic workflow of the framework is as follows: Once the system manager has been set up and the RSs added, we can ask the system for recommendations. This is done by registering the user at the manager.

The manager will then return a (tracked) system that will perform the recommendations. This system is decided by the assigner (implicitly random). On this system, we may call methods for retrieving recommendations. If it is possible for the user to have their system changed during one session, we should always ask the manager for the user's current recommender before getting the recommendation. Inactive users should be de-registered from the manager (for instance when their session expires).

For the purposes of RS development, it is possible to create and add new systems, change the existing ones or remove them entirely during the use of the framework. If the implementer wishes to test another version of the same system, we advise that they clone this system and either change its name or its version before adding it to the manager. Not doing so would cause confusion about which parameters belong to which version and the recommendations of the old system would be attributed to the new system.

Finally, the process of RS evaluation is described in the next section.

3.6 Evaluation

Evaluation can be done by a calling given evaluation function on the system's tracker. This approach automatically pulls the required data from the database and passes it to the corresponding Evaluation function. By required data, we usually mean a set of all the recommendation lists the system provided during its use. If necessary, the framework will also pull all feedback the user provided to the recommended items.

For large datasets, this can be a lengthy process. So if we wish to evaluate along multiple metrics with the same input data. Getting the data explicitly and calling these methods directly on the Evaluation class might be preferable. This can also be useful when we wish to evaluate the system on a subset of recommendations it provided (i.e. for a certain time period, certain demographic of users, . . .)

For the complete list of evaluation methods, see the Evaluation chapter in the framework architecture part of this work (section 2.9).

3.7 Further customization of functionality

New users will be assigned to systems randomly, although we may wish to implement a more sophisticated policy. For instance, we may want the probability distribution to be uneven across systems. Furthermore, we may wish to change this probability or to cut off some system from being assigned entirely.

Another big part of functionality that can be customized is the creation of own IDatabase implementation better suited for a given use case. The public methods in our default implementation are all virtual, so extending of our implementations is also possible.

Other customization methods are used for evaluation. Like methods for deciding whether a user finds a given item relevant based on feedback or retrieval of user's relevant items. Another method is used to tell to what extent 2 items are different. These allow us to calculate metrics such as Precision, Diversity, ...

4. Framework instance

To showcase the framework and its capabilities we've built a web application that will serve as a mock-up of a movie database website. This website will be using our default database implementation and will be storing the data in a local SQL database.

4.1 Data and setting

Data for the training of our RSs will be obtained from the MovieLens dataset [1]. More specifically, we will be using the "MovieLens 20M Dataset" described by the authors as:

"Stable benchmark dataset. 20 million ratings and 465,000 tag applications applied to 27,000 movies by 138,000 users. Includes tag genome data with 12 million relevance scores across 1,100 tags. Released 4/2015; updated 10/2016 to update links.csv and add tag genome data."

For demonstration purposes and in order to save on storage, we will reduce the number of ratings we use to about 4 million. This is done by randomly sampling users, selecting only 1 in 5. These 4 million ratings span $\sim$ 27,000 movies (some of which are actually TV shows) in the original data. If required, it is possible to use the whole dataset. All of the data preprocessing code used is also provided alongside this work.

To make the website more "life-like" we will also need some additional information about these movies. For this, we will be using the TMDb API [17].

4.1.1 User and Item objects

The MovieLens dataset anonymizes user data for the sake of privacy. Therefore all we know about our users is their identification number.

For movies, on the other hand, the dataset provides their IMDb [18] and TMDb [17] IDs. We will be using the TMDb API to get additional information about movies (see 4.1.3). This data will be displayed to the users of our application.

Figure 4.1: TMDb logo

4.1.2 Feedback objects

When it comes to feedback, we will be using ratings obtained from the MovieLens dataset. We will also be measuring 2 online-only types of feedback: how long the user spends looking at each movie's page and whether they clicked on a recommended movie.

We will not be using MovieLens tags given by the users. We mention their availability here nonetheless in case the implementer wishes to make use of them. The source code provided also contains methods used to filter tag feedback and to load it into the database. These methods are not used by default during the preprocessing phase.

4.1.3 Data preprocessing

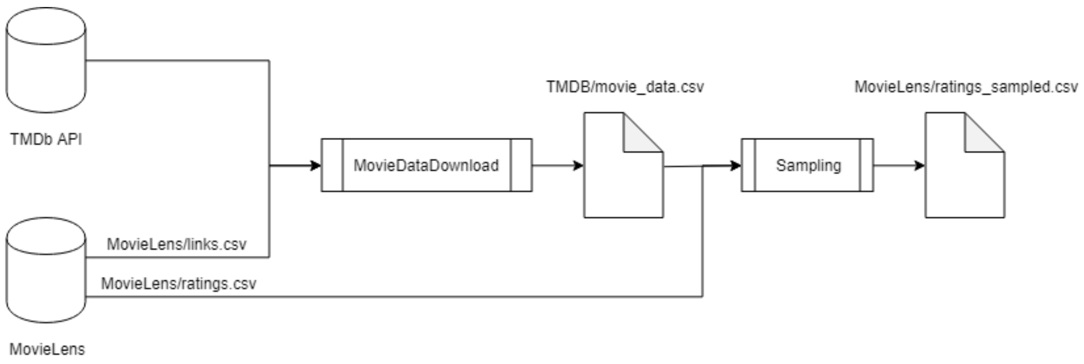


Figure 4.2: Diagram of the data preprocessing pipeline

Downloading of movie data

After we have obtained the MovieLens dataset, the first step in the preprocessing is the downloading of movie data. This is done by using TMDb API. To communicate with this API we will be using the ”TMDbLib” library [19]. Data that is saved for each movie consist of the titles, overviews, genres, runtimes, release dates, rating counts, and rating averages.

We leave out those movies (and TV shows), whose TMDb ID is not known. We also leave out those that don’t have a name or don’t have an overview. This data is then stored in a CSV file.

User sampling

Next step is the sampling of users. As a first step, we load all of the user IDs found in the ratings file. Then we keep each user with the probability of 1/5.

Once we have the list of sampled users, we can filter the ratings. We keep only those ratings whose user ID is within the sampled users and whose movie ID is within the filtered movies (downloaded in the previous step).

Optionally, if we wish to include tag feedback, we can filter it in the same way as the ratings.

Downloading of posters

Movie posters are not included in the data provided with this work as they would take up too much disk space. The code to download the movie posters is provided with this work. Its use is encouraged for demonstration purposes.

4.2 Used recommenders

For the implementation of RSs, we will be using the MyMediaLite .NET library [2]. It provides the implementation for several popular recommendation algorithms.

We create classes that serve as wrappers of MyMediaLite classes. These wrappers implement the IManagedRecommenderSystem interface (see section 2.4.2). Wrapping them in this way is necessary in order to allow us to add these systems to the manager. The algorithms we use are User Average, Item Average, Global Average, Matrix Factorization, Biased Matrix Factorization, and User-Item Baseline.

All of these systems have been trained on our filtered rating data. The application is set up to start only with one version of Biased Matrix Factorization and one version of User-Item Baseline. The number and types of systems on application startup can be configured if desired.

4.3 RecommenderService

RecommenderService functions as a central point for our application.

It defines user, item, and feedback objects in the forms of Viewer, Movie and DwellTime classes respectively. It implements wrappers for the above-mentioned 6 recommender systems from the MyMediaLite library. The service sets up an instance of our recommender framework and uses it to provide users with recommendations.

The system users can use the service to look up movies, their viewing history, and statistics about their rating habits. The administrator can use the service to evaluate, add, and remove recommenders.

Recommendations are provided to users according to the following policy: Implicitly the user is in anonymous mode, where the recommended items are just a randomly sampled list of popular movies.

If the user decides to "sign in" they will be assigned a recommender and a recommendation will be displayed to them. We keep recommendations persistent using RecommendationManager. This manager stores the user's recommendation until the end of the session. A user session, by default, lasts 30 minutes from the time the recommendation was provided. If we receive noteworthy feedback (a rating) we update the recommender and request a new recommendation.

DwellTimeTracker serves as a way for us to keep track of how long users spend looking at a given movie. For more information about how we track dwell time see section 4.5.2.

4.4 Web application

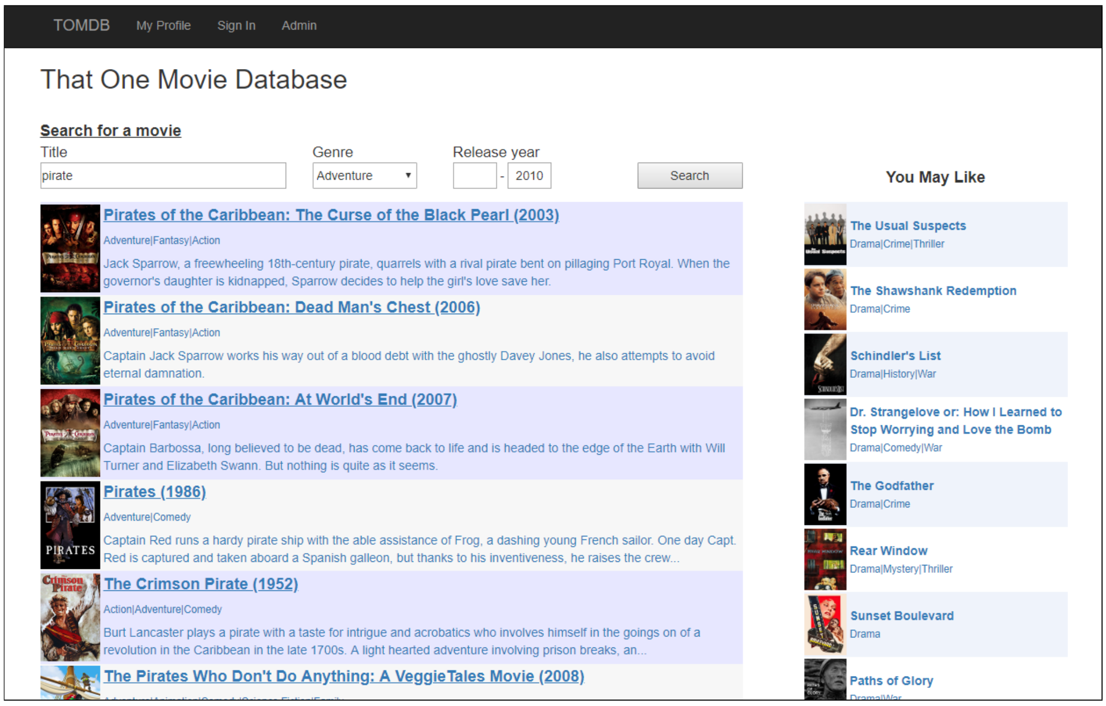
This section serves as a brief overview of our web application, of the individual pages, and of what they allow us to do.

4.4.1 User side

The application landing page allows users to search for movies based on criteria we think they might find relevant. These being the title of the movie, its genre, and the year of release.

By default, the movies displayed in the search list and in the recommendation section on the right are randomly sampled from movies with at least 3000 ratings.

If the user wishes, they may "sign in" by using their unique ID number. A new user can also be added by using the register option. Once the user is signed in, the recommendation banner will start showing a personalized list of recommended items. They also get access to the "My Profile" section, where they can see the movies they have rated in the past and some statistics about their ratings.



That One Movie Database

Search for a movie

Title: pirate | Genre: Adventure | Release year: - 2010

Pirates of the Caribbean: The Curse of the Black Pearl (2003)
Adventure|Fantasy|Action
Jack Sparrow, a freewheeling 18th-century pirate, quarrels with a rival pirate bent on pillaging Port Royal. When the governor's daughter is kidnapped, Sparrow decides to help the girl's love save her.

Pirates of the Caribbean: Dead Man's Chest (2006)
Adventure|Fantasy|Action
Captain Jack Sparrow works his way out of a blood debt with the ghostly Davey Jones, he also attempts to avoid eternal damnation.

Pirates of the Caribbean: At World's End (2007)
Adventure|Fantasy|Action
Captain Barbossa, long believed to be dead, has come back to life and is headed to the edge of the Earth with Will Turner and Elizabeth Swann. But nothing is quite as it seems.

Pirates (1986)
Adventure|Comedy
Captain Red runs a hardy pirate ship with the able assistance of Frog, a dashing young French sailor. One day Capt. Red is captured and taken aboard a Spanish galleon, but thanks to his inventiveness, he raises the crew...

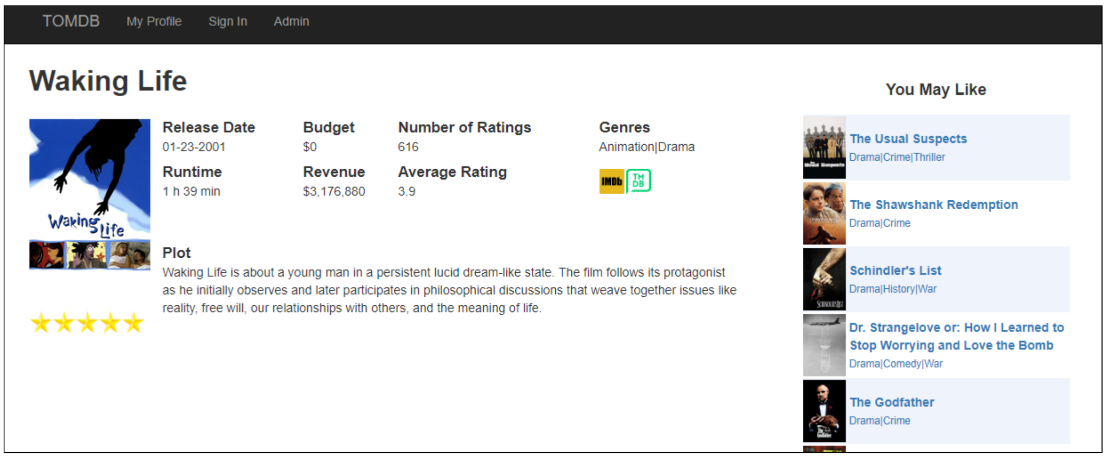
The Crimson Pirate (1952)
Action|Adventure|Comedy
Burt Lancaster plays a pirate with a taste for intrigue and acrobatics who involves himself in the goings on of a revolution in the Caribbean in the late 1700s. A light hearted adventure involving prison breaks, an...

The Pirates Who Don't Do Anything: A VeggieTales Movie (2008)

You May Like

The Usual Suspects — Drama|Crime|Thriller
The Shawshank Redemption — Drama|Crime
Schindler's List — Drama|History|War
Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb — Drama|Comedy|War
The Godfather — Drama|Crime
Rear Window — Drama|Mystery|Thriller
Sunset Boulevard — Drama
Paths of Glory

Figure 4.3: Application landing page



Waking Life

Release Date	Budget	Number of Ratings	Genres
01-23-2001	$0	616	Animation|Drama
Runtime	**Revenue**	**Average Rating**	
1 h 39 min	$3,176,880	3.9	

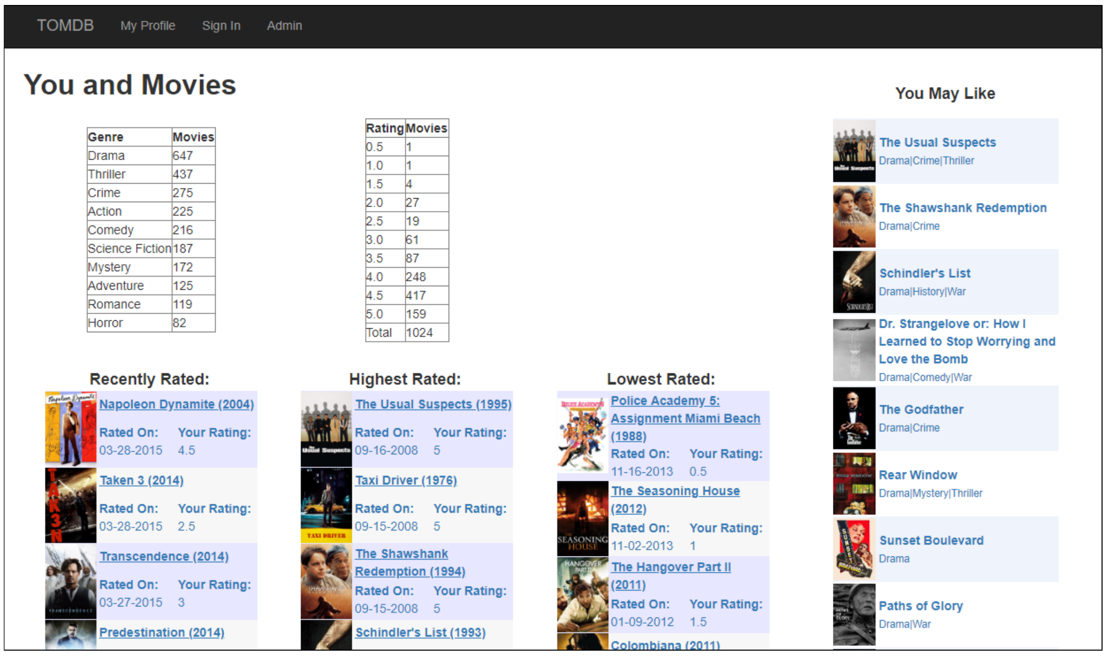
★★★★★

Plot
Waking Life is about a young man in a persistent lucid dream-like state. The film follows its protagonist as he initially observes and later participates in philosophical discussions that weave together issues like reality, free will, our relationships with others, and the meaning of life.

You May Like

The Usual Suspects — Drama|Crime|Thriller
The Shawshank Redemption — Drama|Crime
Schindler's List — Drama|History|War
Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb — Drama|Comedy|War
The Godfather — Drama|Crime

Figure 4.4: Detailed view of a movie



You and Movies

Genre	Movies
Drama	647
Thriller	437
Crime	275
Action	225
Comedy	216
Science Fiction	187
Mystery	172
Adventure	125
Romance	119
Horror	82

Rating	Movies
0.5	1
1.0	1
1.5	4
2.0	27
2.5	19
3.0	61
3.5	87
4.0	248
4.5	417
5.0	159
Total	1024

Recently Rated:

	Rated On:	Your Rating:
Napoleon Dynamite (2004)	03-28-2015	4.5
Taken 3 (2014)	03-28-2015	2.5
Transcendence (2014)	03-27-2015	3
Predestination (2014)		

Highest Rated:

	Rated On:	Your Rating:
The Usual Suspects (1995)	09-16-2008	5
Taxi Driver (1976)	09-15-2008	5
The Shawshank Redemption (1994)	09-15-2008	5
Schindler's List (1993)		

Lowest Rated:

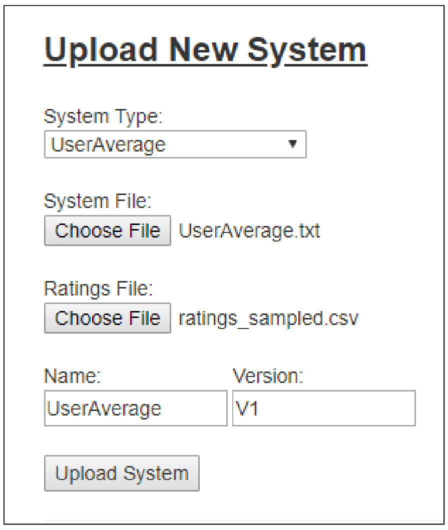
	Rated On:	Your Rating:
Police Academy 5: Assignment Miami Beach (1988)	11-16-2013	0.5
The Seasoning House (2012)	11-02-2013	1
The Hangover Part II (2011)	01-09-2012	1.5
Colombiana (2011)		

You May Like

The Usual Suspects	Drama|Crime|Thriller
The Shawshank Redemption	Drama|Crime
Schindler's List	Drama|History|War
Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb	Drama|Comedy|War
The Godfather	Drama|Crime
Rear Window	Drama|Mystery|Thriller
Sunset Boulevard	Drama
Paths of Glory	Drama|War

Figure 4.5: Page containing user's rating statistics

4.4.2 Administrator side



Upload New System

System Type:	UserAverage
System File:	UserAverage.txt
Ratings File:	ratings_sampled.csv
Name:	UserAverage
Version:	V1

Figure 4.6: System upload section of the admin page

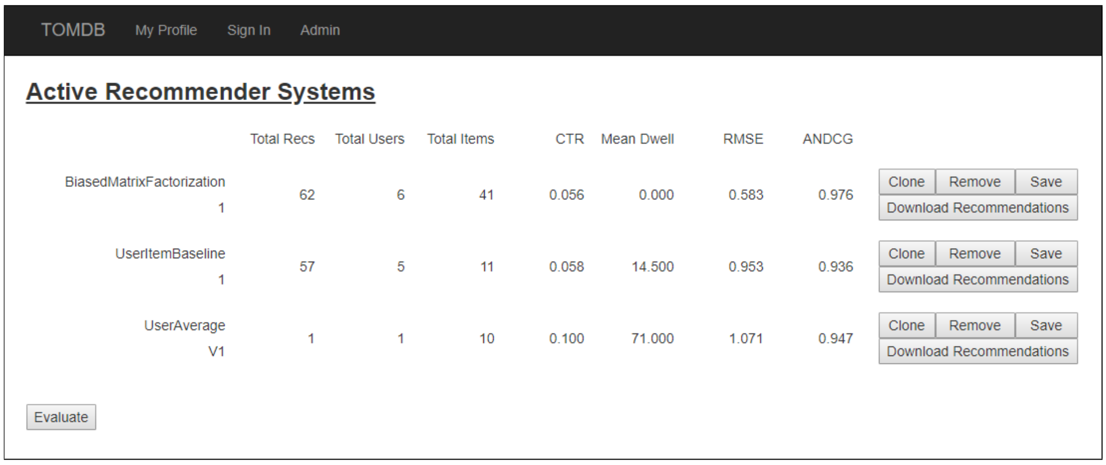
Now let's shift our focus to the "Admin" side of the web application.

This page allows the administrator to see the currently operational systems. It is possible for them to clone a given recommender system and change its parameters. For this process to complete either the systems name or (more likely) the version has to be changed. This has to be done in order for us to be able to distinguish these 2 systems and their recommendations.

It is also possible for the admin to download a given system serialized in the form of a text file. If desired, they may also upload a new recommender system. This system's type has to be one of the 6 implemented in the recommendation service (see 4.2). They require the administrator to give the system a unique identification and to upload 2 files. One containing the serialized recommender system and the other containing the ratings on which the system trained.

The administrator can perform evaluation of the systems to see their performance up to this point. The evaluation shows how many recommendations the system provided, to how many different users it provided those recommendations, and how many different items were recommended. We are also interested in how likely users are to click on system's recommendation and how much time they spend on looking at the recommended movies.

The administrator may wish to perform their own evaluation or use the data for a different purpose. It is possible to download the recommendations of each system and all feedback stored in the database by type. This data is downloaded in the form of a CSV files.



Active Recommender Systems

	Total Recs	Total Users	Total Items	CTR	Mean Dwell	RMSE	ANDCG	
BiasedMatrixFactorization 1	62	6	41	0.056	0.000	0.583	0.976	Clone Remove Save Download Recommendations
UserItemBaseline 1	57	5	11	0.058	14.500	0.953	0.936	Clone Remove Save Download Recommendations
UserAverage V1	1	1	10	0.100	71.000	1.071	0.947	Clone Remove Save Download Recommendations

Figure 4.7: System evaluation section of the admin page

4.5 Feedback tracking

We track 3 different types of user feedback: ratings, clicks on recommended items, and dwell time. When we record some type of feedback, it gets added to our database and passed to the recommender via a RecommenderSystemTracker (see 2.5). When our RS receives information about a new rating, it performs an incremental training step to take into account this new rating for future recommendations.

4.5.1 Ratings and clicks

Rating and click on recommendation measurements are simple. When they happen, they trigger a server-side handler method. This method can then parse the feedback and submit it to the service.

Our aggregation function used for evaluation works simply by looking at whether the user rated a given item. If so, the most recent rating is returned.

4.5.2 Dwell time

For measuring dwell time we create a timer running on the client-side web browser when they open any movie's page. This timer periodically (every 5 seconds) pings the server telling us "this user is still looking at the page of this movie".

We then keep track of such events on the server-side. More specifically, we know when the first ping and when the last ping (up until now) were sent. If it's been more than 15 seconds since the last ping, we assume that the user closed the movie's page. At this point, the dwell time feedback gets sent back to the RS.

4.6 How to run the applications

The application is built for the .NET Framework version 4.6.1.

4.6.1 Project structure

Data

The data section contains 3 main parts. These are the sampled MovieLens data (subfolder: **MovieLens**), trained and serialized recommender systems (subfolder: **Systems**), and TMDb movie data (subfolder: **TMDb**).

Code

Code section contains 3 main project solutions. First one is the recommender framework itself (subfolder: **RecommenderFramework**). The second one contains projects for data preprocessing (subfolder: **DataPreprocessing**). Finally, the third solution contains program used for loading data into the database, the recommendation service using our framework, and the web application (subfolder: **FrameworkInstance**).

The individual projects (programs) include a windows compatible .exe build located in ***ProjectName*/bin/Release/** folder. They allow for configuration through their respective config files named ***ProjectName*.exe.config**. These files mostly allow to configure file locations but differ for each project. We encourage the user of this project to have a look at the configuration options for themselves.

Since the web application is not a standard console application but is built in ASP.NET, its folder structure is slightly different. The build and the configuration file (**Web.config**) of the web application can be found within a subfolder named **FrameworkInstance/WebApplication/bin/Release/Publish/**.

4.6.2 The necessary

There are 2 main steps needed to run the web application.

Loading of the database

Since we are using our default database implementation, we have to load the users, items, and feedback into the database. These classes include Data.Linq.Mapping annotations. Instantiating of the default database takes in a connection string of the database. This connection string must be specified in the DatabaseLoader configuration file.

If the tables for recommendations, recommended items, and system parameters aren't yet present in the database, the default database constructor creates them automatically. Similarly, the application itself sets up the tables for users, items, and feedback.

Starting the application

The web application can be started using IIS. In the Internet Information Services (IIS) Manager open you local machine in the connections tab and right click on "Sites" then "Add Website...". In the set up form, at least the port and the "Physical Path" need to be specified. This path should lead to the published app folder **FrameworkInstance/WebApplication/bin/Release/Publish/**. For a more detailed startup guide see [20].

The application also requires a connection string to the database (the same one as was used during data loading). This string should be provided in the publish folder in the file named **Web.config**.

4.6.3 The optional

The user of this project may wish to run the whole preprocessing pipeline again (maybe because a new version of MovieLens dataset is released). The intended order of the projects in which they should be run is: 1. MovieDataDownload, 2. Sampling, 3. SystemTraining (4. MoviePosterDownload).

The configuration files specify the file locations for these steps. Implicitly the pipeline expects the MovieLens raw data in the data/MovieLens folder. If the user wishes to download the TMDb movie data, they need to obtain an API key and provide it in the appropriate config file.

The most likely optional step the user of this project may wish to perform is the downloading of movie posters. The application that does this is located in the **DataPreprocessing/MoviePosterDownload** project. The configuration file for this project contains locations of the file where we can find movie data (containing the poster paths) and of the folder where the posters should be saved. The web application expects to find these poster images in the subfolder which is named **WebApplication/bin/Release/Publish/Images/Posters/**. The configuration file also specifies the resulting size of the images.

5. Instance adaptation to a different dataset

This chapter describes how it would be possible to adapt the application to a different (but similar enough) dataset. As an example, let's imagine that we wish to create a similar application, but instead of movies, it would show books to our users.

Many of the steps described here are similar to steps that have already been presented in chapter 4. We will also leave out the data preprocessing stage as that has to be handled by the implementer.

5.1 Data preparation

We shall start again with the definition (or in this case re-definition) of what our items are.

We would first rename our Movie class to Book and then decide which properties we wish to keep. The title, overview, genres, and release date seem like useful properties we should keep alongside the number of ratings the book received and the rating average. One important property we need to add is the author of the book. Another might be the book's ISBN number and the publisher.

Feedback types we have used for the previous dataset (ratings, clicks on recommendation, dwell time) seem appropriate for this dataset too, therefore we will not be changing them. Recommender systems used in the last chapter are quite versatile since they don't take any content data as input, so we can reuse them as well.

5.2 Application functionality

Most of the work needed to adapt our application will be required in the service section.

First off, let's start with the parts of the service we don't need to change. Thanks to the abstraction of what an item actually is, we are able to have recommender systems that are not dependent on item types. This means that all manipulation with recommender systems (their adding, removing, saving, cloning, and evaluation) can stay unchanged. Same goes for the downloading of system recommendations.

We also don't need to change the methods used to measure feedback (unless some feedback type was added). This also means that downloading of feedback as a CSV file can stay the same.

Where we do need to perform changes is the area of movie (now to be book) browsing. For instance, before we had the option of looking up movies by the year of their release, but this might not be as relevant for books. Instead, we could replace this option and allow users to look for books based on the author. Similarly, we may wish to refactor the rest of the methods in the "Movie Browsing" source code region.

Same goes for the user's profile page (source code region named "My Profile"). We don't need to change methods for the calculation of user's rating distribution and favorite book genres. What we need to change is the part where we retrieve user's recently, highest, and lowest rated books. Even though we don't look at any movie properties directly, we do keep some of them for display purposes in the RatedMovie class.

5.3 Web presentation

Another big change needed when going from movies to books is in the form we wish to present them. As has been mentioned before, the date of the book's release is probably not that important. On the other hand, we should always display the book's author close to the title.

To achieve this, we need to change the way the web application displays items. Where possible, we try to use markup to specify display templates. More specifically, we are talking about ItemTemplates for the DataList ASP.NET element. These templates can be divided into 3 categories.

The first category is the search. This one is used on the landing page when a user searches for movies (books).

The second one is the recommendation template. This one is used to display the recommendations on the right side of the page. This template has to be copied to all the pages where it is used. These are the landing page (called Main), the MovieDetail page, and MyProfile page.

The third template we need is used 3 times on the user's profile page (named MyProfile). It is used to display user's recently, highest, and lowest rated items. Again, this template has to be copied between all 3 location.

To display these items properly, the DataLists need to have a bound data source from which they can pull item data. In the case of the item's own page (MovieDetail), we aren't able to use a markup template and thus, binding has to be done manually for each of the page's display elements.

Conclusion

In this work, we have been able to showcase how modern recommender systems operate, what requirements are placed on them, what might be their desired properties, and how their developers may try to satisfy these requirements and what problems they usually encounter.

We have also discussed possible ways of gathering data about systems' performance and how this data can be used for their evaluation. We have placed a special emphasis on online evaluation and clarified why we hope for wider adoption of this evaluation method.

We have been able to take this knowledge to design and implement a framework that would allow us to track and manipulate these various recommender systems in online use. We have designed this framework in a way that is easy to use and requires minimal effort on the side of the implementer.

To showcase the possible usage of this framework we have built a movie database website where users could rate movies and in turn receive recommendations. We have shown how our framework allows the implementer to change the application's recommender systems during use, how they can add or remove systems, and made it possible to save these systems, their recommendations and received feedback.

Lastly, we have also shown how our web service and application could be used as a base for a more sophisticated system or be repurposed for another dataset.

Future work

During the development, we've encountered some areas where this framework could be further improved.

One of these areas would be the ability to extract SQL table definitions from Data.Linq.Mapping annotations. This table data could then be used to create/delete tables. This way, the implementer wouldn't need to perform the table creation themself.

Another area of improvement is feedback aggregation. Currently, we query all feedback for each interesting user-item pair. But as we have found out in chapter 4, sometimes we only need one type of feedback (explicit). Thus, we might be querying the database when not necessary. A better aggregator would be one that checks whether an explicit rating exists before retrieving more data. Another possibility is allowing the aggregator to download the data itself.

Bibliography

[1] Movielens dataset. `https://grouplens.org/datasets/movielens/`. [Online; accessed 09-July-2019].

[2] Mymedialite recommender system library. `http://www.mymedialite.net/`. [Online; accessed 09-July-2019].

[3] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to Recommender Systems Handbook*, pages 1–35. Springer US, Boston, MA, 2011.

[4] Dietmar Jannach, Lukas Lerche, Fatih Gedikli, and Geoffray Bonnin. What recommenders recommend - an analysis of accuracy, popularity, and sales diversity effects. In *In Proc. UMAP 2013*, 2013.

[5] Guy Shani and Asela Gunawardana. *Evaluating Recommendation Systems*, pages 257–297. Springer US, Boston, MA, 2011.

[6] Thiago Silveira, Min Zhang, Xiao Lin, Yiqun Liu, and Shaoping Ma. How good your recommender system is? a survey on evaluations in recommendation. *International Journal of Machine Learning and Cybernetics*, 10(5):813–831, May 2019.

[7] Arvind Narayanan and Vitaly Shmatikov. How to break anonymity of the netflix prize dataset. *CoRR*, abs/cs/0610105, 2006.

[8] Dan Cosley, Shyong K. Lam, Istvan Albert, Joseph A. Konstan, and John Riedl. Is seeing believing?: How recommender system interfaces affect users' opinions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 585–592, New York, NY, USA, 2003. ACM.

[9] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. *Collaborative Filtering Recommender Systems*, pages 291–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[10] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, pages 30–37, 2009.

[11] Badrul M Sarwar, George Karypis, Joseph Konstan, and John Riedl. Recommender systems for large-scale e-commerce: Scalable neighborhood formation using clustering. In *Proceedings of the fifth international conference on computer and information technology*, volume 1, pages 291–324, 2002.

[12] Robin Burke. *Hybrid Web Recommender Systems*, pages 377–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[13] Haihua Luo, Xiaoyan Zhang, Bowei Chen, and Guibing Guo. Multi-view visual bayesian personalized ranking from implicit feedback. In *Proceedings of the 26th Conference on User Modeling, Adaptation and Personalization*, UMAP '18, pages 361–362, New York, NY, USA, 2018. ACM.

[14] Ruining He and Julian McAuley. VBPR: visual bayesian personalized ranking from implicit feedback. *CoRR*, abs/1510.01784, 2015.

[15] Joeran Beel, Stefan Langer, Marcel Genzmehr, and Andreas Nürnberger. A comparative analysis of offline and online evaluations and discussion of research paper recommender system evaluation. In *in Proceedings of the Workshop on Reproducibility and Replication in Recommender Systems Evaluation (RepSys) at the ACM Recommender System Conference (RecSys), 2013*, pages 7–14, 2013.

[16] System.data.linq.mapping namespace. `https://docs.microsoft.com/en-us/dotnet/api/system.data.linq.mapping`. [Online; accessed 09-July-2019].

[17] The movie database. `https://www.themoviedb.org/`. [Online; accessed 09-July-2019].

[18] Internet movie database. `https://www.imdb.com/`. [Online; accessed 09-July-2019].

[19] Tmdblib. `https://github.com/LordMike/TMDbLib`. [Online; accessed 09-July-2019].

[20] How to create website in iis on windows. `https://tecadmin.net/create-website-in-iis/`. [Online; accessed 09-July-2019].

List of Figures

Attachments

Included as attachments are the following:

- Data folder:

  This folder contains sampled MovieLens rating data, additional data about these movies obtained from TMDb, and a set of serialized recommender systems.

- Code folder:

  This folder contains source projects and builds of our data preprocessing applications, the recommender framework itself, and a web application used to showcase the framework's functionality.