University name: American University of Armenia
Student name: Davit Martirosyan
Supervisor name: Gagik Chakhoyan

CAPSTONE

“Design and Performance Evaluation of Content-based & Collaborative Filtering Recommender Systems for Suggesting Coding Lessons to SoloLearn's Mobile Application Users”

Abstract

This research aimed at designing and evaluating content-based and collaborative filtering recommender systems for suggesting coding lessons to SoloLearn's mobile application users. Both user-based and item-based collaborative filtering were applied on the data. In addition, the famous matrix factorization algorithm called SVD was applied. Only offline evaluation was carried out which was based on a specially designed evaluation framework including the following metrics: RMSE, MAE, hit-rate (HR), average reciprocal hit rate (ARHR), cumulative hit rate (CHR), diversity, user coverage and novelty. Albeit the initial expectations, the best algorithm was the user-based collaborative filtering. All the codes were written in python and can be found in GitHub (https://github.com/Davit98/RecomSystemSolo). Surprise, a python library created for building and analyzing recommender systems, was extensively used throughout the project.

Contents

Introduction

In general, one of the key factors affecting a business growth and success is how personalized they are able to make their products for their users. More is the personalization, more is the user engagement and satisfaction. There are numerous examples of famous businesses which have succeeded in using personalization to boost their business. Perhaps, the simplest example one can think of is “Facebook”. “Facebook” has done so well at creating personalized accounts for its users, that “Facebook” is no longer solely a means of messaging, but also a convenient information source where each user sees only the content that matters to him/her. This is one of the reasons that “Facebook” is the biggest social network in the world and that it has one of the best user retention rates.

“YouTube”, developed by the company “Google”, is yet another notable example of a successful personalized product. “YouTube” uses the data generated by its users to create personalized video recommendations. This practice allows users to always find something new that interests them leading to high satisfaction and an ultimate desire to continue using the product.

Undoubtedly, if used correctly, personalization can be one of the driving forces of a business. This explains why many companies such as “SoloLearn”, an Armenian start-up offering free coding classes in different programming languages through web and mobile applications, want to build a **recommender system** – an engine that will be able to wisely utilize user data and produce personalized recommendations for the users.

This research paper addresses the problem of developing a recommender system for “SoloLearn” and investigates a number of algorithms that can possibly be used to generate proper recommendations for their users.

1. Recommender Systems

1.1 Introduction

Recommender Systems are software tools and techniques providing suggestions for *items* to be of use to a *user*. "Item" is the general term used to denote the product being recommended and "user" is the general term used for the entity to which recommendations are provided.

The basic idea of recommender systems is to utilize the data that the business provides to infer customer interests. The data collected may include *explicit* as well as *implicit feedback* from the users. For example, "Netflix", a major provider of streaming delivery of movies and television shows, provides users the ability to rate its items on a 5-point scale and then uses this data to generate recommendations. This is an example of a use of an explicit feedback. "Amazon", on the other hand, when making recommendations, takes also into account the act of buying and browsing an item and considers those activities as an endorsement for that item. This is an example of a use of an implicit feedback.

Whether the data collected includes explicit or implicit feedback (or both) from the users, the underlying principle of building a recommender system is the creation of user-item ratings matrix and then using it to generate recommendations.

Lastly, it is worth mentioning, that solving the problem of making recommendations has two main approaches. The first approach is about predicting a rating for each of the user-item combination missing in the training data. This is often referred to as **the matrix completion problem**, since the algorithm solving the problem fills in the missing values in the initial user-item ratings matrix. The second approach, which is often referred to as **top-n recommender**, tries to find the top-n relevant items for a particular user or top-n users to target for a particular item, without doing rating predictions. Note that the first approach is more general, since having predicted ratings for all missing user-item pairs, one can easily rank the predictions and return the top-n results. Nevertheless, which of the two approaches will yield better results, depends entirely on the particular situation one is dealing with.

1.2 Types of Recommender Systems

Generally speaking, recommender system models can be classified into 5 broad categories: **content-based**, **collaborative filtering**, **knowledge-based**, **demographic**, and **hybrid**.

In content-based recommender systems, recommendations to a user are made based solely on the attributes of items. As an example, if we are trying to recommend movies to the user Gagik who gave high ratings to the movies "Creed" and "Southpaw", then our recommender system is going to recommend those movies that contain similar genre keywords to "Creed" and "Southpaw". One good recommendation in this case is, for example, the movie "Boyka: Undisputed".

Collaborative filtering models leverage the behavior of other users to make recommendations. At a very high level, recommender systems based on collaborative filtering try to find users similar to the target user and recommend items they liked. To continue the movie example, let's assume that two users Davit and Arman, who are similar to Gagik, both gave very high rating to the movie "The Imitation Game". Our recommender system is, therefore, going to recommend "The Imitation Game" to Gagik, since both Davit and Arman, who are similar users to Gagik, liked the movie.

In knowledge-based recommender systems, item recommendations are made based on the requirements explicitly specified by the user. These type of recommender systems are particularly useful in domains where the items are not purchased very often. Examples include items such as apartments, cars, financial services, etc. In fact, knowledge-based recommender systems are often considered to be closely related to content-based recommender systems and it is sometimes questioned whether a clear demarcation exists between the two classes of methods or not.

In demographic recommender systems, recommendations are generated based on the demographic profile of the user. The key assumption behind these systems is that different recommendations should be generated for different demographic niches. In fact, there are many Web sites which adopt simple but yet effective personalization solutions based on demographics. Examples include dispatching users to particular Web site based on their language or customizing suggestions according to the age of the user.

And finally, in hybrid recommender systems, as the name suggests, recommendations to a user are made based on the suggestions of 2 or more recommender systems, i.e. some sort of hybridization is done. The idea is very similar to that of ensemble methods, where multiple machine learning algorithms are combined to create a more robust model. Hybrid recommender systems try to leverage the power of combining different recommender systems by eliminating the individual weaknesses of each member recommender system.

To sum up, choosing the best model very much depends on the problem one is dealing with. Some recommender systems, such as knowledge-based systems, are more effective in cold-start settings where only small amount of data is available, while other recommender systems, such as collaborative filtering models, are more effective when an abundant amount of data is available. However, regardless of the situation, it is always a good idea to create a hybrid approach by combining different methods.

1.3 Evaluation of Recommender Systems

Evaluating recommender systems is not an easy task. There are several common operational as well as technical goals (e.g. relevance, novelty) of recommender systems, therefore, to understand how well these goals have been achieved a special framework of evaluation metrics is necessary. An example of such evaluation framework may include the following:

- evaluating rating predictions
  - **Mean absolute error** (MAE)
  - **Root mean square error** (RMSE)
- evaluating top-n recommendations
  - **Hit rate (HR)** – the number of *hits* divided by the number of test set users, where hit is defined as follows: if one of the recommendations in a user's top-n recommendation is something he/she has already rated, that is considered a hit.
  - **Average reciprocal hit rate (ARHR)** – the same as hit rate with the difference that instead of summing up the number of hits, we sum up the reciprocal rank of each hit. The idea behind this metric is to penalize the hits which appear lower in the top-n list.
  - **Cumulative hit rate (cHR)** – the same as hit rate with the difference that only those hits are counted which have an actual rating higher than some threshold. The idea behind this metric is that a credit should not be given for recommending items to users that we think they will not enjoy.
- other useful measures
  - **Coverage** – the percentage of possible recommendations that the system is able to provide. This is an important metric since a recommender system may be highly accurate but unable to ever recommend a certain proportion of the items or to ever recommend to a certain proportion of the users.
  - **Diversity** – a measure of how broad a variety of items the system is recommending to the users. Low diversity is not good and an example would be a recommender system that just recommends the next books in a series that you have started reading but does not recommend books from different authors. High diversity is not good either, since a system recommending completely random items has very high diversity. Therefore, one should be very cautious when evaluating the diversity score and should always look at diversity alongside metrics that measure the quality of the recommendations as well.
  - **Novelty** – a measure of how likely a recommender system is to give recommendations to the users that they are not aware of, or that they have not seen before. Although it may seem controversial, high novelty is not a good thing. This is because if only recommend items the users have never heard of, they may think that the recommender system does not really know them and may therefore become disappointed with the system and the product as a whole.

In fact, the metrics described above are used for evaluating a recommender system in an *offline* setting. They all are certainly very important for correctly evaluating recommender systems, however, the true effectiveness of a given recommender system can only be measured using *online* methods such as A\B testing. This is because online methods allow to measure the direct impact of the recommender system on the end user. After all, having high conversion rate is what really matters.

2. Background

2.1 Project Description

As already mentioned, “SoloLearn” is an Armenian start-up that offers free coding classes through web and mobile apps. The company is mainly specialized in developing mobile applications. It offers 13 different mobile apps, one of which, namely “SoloLearn: Learn to Code Free”, is their main application.

“SoloLearn: Learn to Code Free” mobile app contains a number of coding classes such as ‘What is HTML?’, ‘Introduction to Node.js’ and ‘LinkiedLists’, to name a few. Some of the classes have been created by the company’s content-team, while the others by the users themselves. For the sake of simplicity, let’s call the classes developed by the content-team *lessons (L)* and the classes by their users, *user-lessons (U).*

The aim of this project was to develop a recommender system for the “SoloLearn: Learn to Code Free” mobile application, recommending lessons and user-lessons to the users. After conducting a thorough research, a sample data was taken from the MSSQL database provided by “SoloLearn”, containing specific information about the app’s users, lessons, and user-lessons. The data was then processed and used to run several experiments.

Python was the main programming language used for writing the codes. Many libraries were used throughout the project, the most important of which was *Surprise* – a python library specifically designed for building and analyzing recommender systems.

2.2 Data

Due to computational power limitations, a sample of only 10,000 users[1] was taken for the research purposes. For each user, the following info was queried from the database:

- The number of **views** for each lesson and user-lesson
- The number of **comments** written under each lesson and user-lesson
- The number of **votes** to the comments written under each lesson and user-lesson
- **Bookmarked** lessons and user-lessons

Afterwards, the data was used to create a user-item ratings matrix – a matrix where each entry that corresponds to some user u and an item i, is a single number showing how much u likes i. Note that the collected data was not an explicit but an implicit feedback from the users. Thus, there was a need to develop a special method of rating calculation that would correctly indicate how much value each item has for each user.

After trying a couple of methods, the following was chosen as the method for rating calculation:

$$r_{ui} = \begin{pmatrix} 0.2 * PRoS(vw_{ui}) + 0.4 * PRoS(c_{ui}) + 0.1 * PRoS(v_{ui}) \\ + 0.3 * PRoS(b_{ui}) \end{pmatrix} / 10$$

In the formula shown above,

- r_{ui} is the rating of item i for user u
- vw_{ui} is the number showing how many times user u has viewed item i
- c_{ui} is the number showing how many comments user u has written under item i
- v_{ui} is the number showing how many votes in total user u has given to the comments under item i
- b_{ui} is a number showing if user u has bookmarked item i or not: 0 if bookmarked, 1 otherwise
- $PRoS(x)$ is a function returning the percentage of the values less than or equal to the provided score x in its corresponding array of values

In order to better understand what the function $PRoS(x)$ does, let's look at a specific example. Based on our data, if $x = v_{ui} = 2$, then $PRoS(x) = 80.575$ meaning that 80.575% of the values in "Votes", an array containing all non-zero v_{ui}-s for all possible user-item pairs, are less than or equal to 2.

The chosen method of rating calculation maps the feedback of each user to a single number between 0-10, i.e. $r_{ui} \epsilon [0,10]$. 0 means the user has not viewed the item being considered and any other number between 0 and 10 shows how much the user likes the item, where bigger number corresponds to greater importance. So, in this system, $r_{ui} = 10$ is the highest possible rating and means the user u likes the item i immensely.

[1] In fact, not exactly but approximately 10,000 users were sampled.

Besides constructing a user-item ratings matrix, a matrix of item (lessons and user-lessons) features was created as well. For each item, the following info was queried from the database:

- The course name(s) to which the item belongs[2]
- The total number of views by the users
- The total number of comments written under, including comment replies
- The language tag

In order to convert language tags to a numeric data, a separate table was constructed by hand, where each row corresponded to a language tag such as Python or C++ and each column to a common descriptive attribute of programming languages. The table is presented in Appendix I.

[2] As an additional information, lessons always belong to a course and to only one course. User-lessons, however, can simultaneously belong to several courses or to no course at all.

3. Results

3.1 Content-Based Filtering

Before diving into the results obtained through content-based filtering for our data, let's first look at the problem we wanted to solve and the algorithm used to tackle it:

Problem: Given a user u and an item i, predict r_{ui}.

The main steps of the content-based algorithm used can be summarized as follows:

1. Calculate *similarity scores* between the item i and all other items rated by the user u
2. Form a pair (sim_score, rating) for each rated item and sort the pairs based on the similarity scores in descending order
3. Take the first k results and calculate the weighted average, i.e.

$$\widehat{r_{ui}} = \frac{sim\_score_1 * rating_1 + \cdots + sim\_score_k * rating_k}{sim\_score_1 + \cdots + sim\_score_k}$$

This algorithm is very simple in its essence and very easy to understand. Perhaps, the only question one may ask regarding the steps of it is this: How are similarity scores calculated? The answer is that there are several effective similarity measures each having its own philosophy. Nevertheless, as the experience has shown over the years, in contrast to other measures, cosine similarity[3] is working well in almost all kinds of problem settings. This is the reason it was also the similarity metric used for our data.

Now, after recognizing the problem and the algorithm used to solve it, let's look at Table 3.1 which summarizes the results for $k = 5$ and $k = 20$ cases:

k\Metrics	RMSE	MAE	HR	cHR	ARHR	User-coverage[4]	Diversity	Novelty[5]
5	0.3916	0.1903	0.0345	0.0342	0.0161	0.3430	0.1213	640.8535
20	0.3786	0.1934	0.0217	0.0217	0.0098	0.2185	0.1547	771.8039

[3] Cosine similarity: $CosSim(\vec{x}, \vec{y}) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$

[4] User-coverage shows the percentage of users who have at least one "good" recommendation, where "good" recommendation means an item in the top-n recommendation that has predicted value greater than or equal 1.73 – the third quartile (Q_3) of all the available ratings in the sampled data.

[5] The most natural and correct way of measuring novelty is through online experimentation in which users are explicitly asked whether they were aware of the item(s) previously. However, online experimentation is not always feasible, as it was in this case. As a result, novelty was calculated based on the popularity of the recommended items.

Table 3.1: Evaluation results for the content-based algorithm for $k = 5$ and $k = 20$ cases

As we can see from Table 3.1, for $k = 5$, RMSE is higher compared to when $k = 20$. However, the same is not true about MAE: for $k = 5$ MAE is lower than for $k = 20$. HR, cHR and ARHR scores are all not very impressive for both $k = 5$ and $k = 20$ cases, but it is worth mentioning that in case of $k = 5$ the results are much better. User-coverage is higher for $k = 5$ as well: for $k = 5$, 34.3% of all the users get at least one recommendation with predicted value greater than or equal 1.73, whereas for $k = 20$ the same figure is just 21.85%. What refers to diversity, the model with $k = 20$ has higher diversity than the one with $k = 5$. And finally, both models, in particular the one with $k = 5$, have a novelty score close to the average novelty score for our data which is 659.

Based on the comparisons, there is feeling that the content-based model with $k = 5$ is doing better than the model with $k = 20$. To see if our feeling is right, let's take some user and look at the top-10 recommendation produced by the two models for that specific user. The test user we will use is the one with $id = 12753303$, whose history of viewed lessons and user-lessons with their corresponding ratings is presented in Appendix II. Now, let's look at the table presented below which shows the top-10 recommendations for our test user for both $k = 5$ and $k = 20$ cases:

	k=5			k=20		
Rank	**Item**	**Type**	**Course name**	**Item**	**Type**	**Course name**
1	Exception Handling	L	Java Tutorial	Properties	L	C# Tutorial
2	Module 2 Quiz	L	C Tutorial	Array Manipulations	L	Ruby Tutorial
3	Protected Members	L	C++ Tutorial	Strings	L	Ruby Tutorial
4	Recursive Functions	L	C Tutorial	Arrays	L	Ruby Tutorial
5	Class Attributes	L	Java Tutorial	Module 5 Quiz	L	C Tutorial
6	Template Specialization	L	C++ Tutorial	The Zen of Python	L	Python 3 Tutorial
7	Two-Dimensional Arrays	L	C Tutorial	Module 1 Quiz	L	C# Tutorial
8	Functions & Arrays	L	C Tutorial	Interfaces	L	Java Tutorial
9	Threads	L	Java Tutorial	More on else Statements	L	Python 3 Tutorial
10	Module 7 Quiz	L	C++ Tutorial	Preprocessor Operators	L	C Tutorial

Table 3.2: Top-10 recommendation results for the test user produced by the content-based algorithm for $k = 5$ and $k = 20$ cases

As we can see from Table 3.2, the recommendations made by both models are quite odd, given that our test user has a long history of viewing lessons from "CSS Fundamentals" and "JavaScript Tutorial" courses. The recommendations generated by both of the models seem to be so inappropriate that is in fact almost impossible to assess which model does a better job.

To conclude, our impression is that content-based recommender systems are not performing well for our data. Neither the evaluation metrics nor the top-n recommendations seem to be good enough. Nevertheless, one should always keep in mind, that only after A\B testing it is possible to truly evaluate how good or bad the recommender systems are performing.

3.2 Collaborative Filtering

3.2.1 User-Based Collaborative Filtering

In general, user-based collaborative filtering can be used both as a rating predictor (referred to as User-based KNN) and as a top-n recommender. Although, only a slight change is necessary to convert user-based collaborative filtering from a top-n recommender to a rating predictor or vice versa, as we will see, the difference between the results can be dramatic.

Let's first look at the algorithm of the top-n recommender. The main steps can be summarized as follows:

1. Construct a *user-to-user similarity* matrix based on the user-item ratings matrix. Given that the latter is a matrix of size $n\_users \times n\_items$, where each row corresponds to a user and each column corresponds to an item, the similarity between users u and v is calculated as $CosSim(row_u, row_v)$.
2. For each user u, take the k-most similar users.
3. Iterate through the items rated by each similar user v giving a score to each item i as follows:
$$score_i = \frac{r_{vi}}{10} * sim\_score(u, v)$$
4. Combine the scores for each item i from different similar users.
5. Sort the items based on the scores in descending order and return the first n non-viewed items by user u.

Now, after understanding how the algorithm works, let's look at the evaluation results[6] which are presented in Table 3.3 shown below:

k\Metrics	HR	cHR	ARHR	Diversity	Novelty
20	0.2183	0.2175	0.1455	0.0590	115.1457

Table 3.3: Evaluation results for the user-based top-n recommender algorithm for $k = 20$

As we can see from the table, the algorithm has pretty high score for hit rate (HR) which is promising. cHR is only slightly less than HR and this is because the threshold used to filter some hits out was quite low. The score for ARHR is good as well.

What refers to diversity and novelty, as we can see, the former has pretty low score, telling that the generated recommendations are usually very similar to each other. The score for novelty is low too, meaning that the items being recommended are mostly quite popular.

[6] Note that since the algorithm is not doing rating predictions, no RMSE, MAE, and User-Coverage scores can be calculated.

Now let's look at the top-10 recommendation for our test user, which is presented in Table 3.4 shown below:

	k=20		
Rank	**Item**	**Type**	**Course name**
1	What is HTML?	L	HTML Fundamentals
2	Creating Your First HTML Page	L	HTML Fundamentals
3	Basic HTML Document Structure	L	HTML Fundamentals
4	A Hello World Program	L	Java Tutorial
5	AngularJS: Expressions	U	Development
6	Text Formatting	L	HTML Fundamentals
7	What is Python?	L	Python 3 Tutorial
8	Adding JavaScript to a Web Page	L	JavaScript Tutorial
9	Introduction to Java	L	Java Tutorial
10	Blog Project: About Me	L	HTML Fundamentals

Table 3.4: Top-10 recommendation results for the test user produced by the user-based top-n recommender algorithm for $k = 20$

Given the history of our test user, who has viewed a number of lessons from "CSS Fundamentals" and "JavaScript Tutorial" courses but surprisingly none from "HTML Fundamentals", the generated recommendations seem to be absolutely appropriate. So, the good impression we got when looking at the evaluation metrics is enhanced by the quality of top-n recommendation results.

Now, let's move to the rating predictor algorithm, i.e. User-based KNN. The main steps of the algorithm can be summarized as follows:

1. Construct a *user-to-user similarity* matrix based on the user-item ratings matrix
2. Take the k-most similar users to the user u who have rated the item i
3. Calculate the weighted average, i.e.

$$\widehat{r_{ui}} = \frac{\sum_v sim\_score(u,v) \times r_{vi}}{\sum_v sim\_score(u,v)}$$

The evaluation results of the User-based KNN for our data are presented in Table 3.5 shown below:

k\Metrics	RMSE	MAE	HR	cHR	ARHR	User-coverage	Diversity	Novelty
20	0.4221	0.1947	0.0002	0.0002	0.0000	1.0000	0.2432	1238.2224

Table 3.5: Evaluation results for the User-based KNN for $k = 20$

As we can see, the results are quite disappointing. All HR, cHR, and ARHR scores are extremely low with ARHR being equal to 0. One thing that is interesting is the fact that User-coverage is 1. This means that every user always gets at least one recommendation that has quite high predicted rating.

What refers to diversity and novelty, the former is a little bit lower but the latter is pretty high. High novelty score means that the items being recommended are mostly not very popular.

Now let's look at the top-10 recommendation for our test user to get better idea of the User-based KNN performance and suitability for our data. The results are presented in Table 3.6 shown below:

	k=20		
Rank	**Item**	**Type**	**Course name**
1	Users Route (POST)	U	Node.js
2	Get User By ID	U	Node.js
3	Body Parser	U	Node.js
4	Vector Arithmetic	U	The R Language
5	State & Props: Assignment Solution[7]	U	
6	MongoDB Model	U	Node.js
7	First Request Handler	U	Node.js
8	Update Users	U	Node.js
9	Making API Requests	U	Node.js

[7] This is an example of user-lesson that does not belong to any course.

10	Users Route (GET)	U	Node.js

Table 3.6: Top-10 recommendation results for the test user produced by the User-based KNN for $k = 20$

When looking at the top-10 recommendation results, the first thing that is really eye-catching is that all recommended items are user-lessons. However, it is not very surprising, given the high novelty score.

Another thing that is also immediately noticeable is the fact that 8 of the recommendations out of 10 are from the course called “Node.js”. Although, based on our test user history, it is fair to assume that the user will probably be interested in node.js, however, the facts that simultaneously 8 recommendations are from “Node.js” and that they are not popular, make the top-10 recommendation rather inadequate.

To sum up, as we have seen, user-based collaborative filtering can yield dramatically different results depending on whether it was as used a rating predictor or as a top-n recommender.

3.2.2 Item-Based Collaborative Filtering

Like user-based collaborative filtering, item-based collaborative filtering can be used both as a rating predictor (referred to as Item-based KNN) and as a top-n recommender. At first, let's look at the top-n recommender. The main steps of the algorithm can be summarized as follows:

1. Construct an *item-to-item similarity* matrix based on the user-item ratings matrix. Given that the latter is a matrix of size $n\_users \times n\_items$, where each row corresponds to a user and each column corresponds to an item, the similarity between items i and j is calculated as $CosSim(column_i, column_j)$.
2. For each user u, take the k-highest rated items $\triangleq K$.
3. Iterate through all items giving a score to each item i as follows:
$$score_i = \sum_{k \in K} sim\_score(i,k) \times \frac{r_{uk}}{10}$$
4. Sort the items based on the scores in descending order and return the first n non-viewed items by user u.

It is not difficult to notice that the algorithm described above is just the flipping of algorithm in the user-based collaborative filtering. In simpler words, instead of looking for similar users and recommending stuff they liked, in the item-based collaborative filtering, we are taking the things the target user liked and recommending stuff that is similar to those things.

Now, let's look at the evalution results presented in Table 3.7 which is shown below:

k\Metrics	HR	cHR	ARHR	Diversity	Novelty
10	0.0004	0.0004	0.0001	0.6121	1230.1907

Table 3.7: Evaluation results for the item-based top-n recommender algorithm for $k = 10$

As we can see from the table, the algorithm has pretty low scores for HR, cHR, and ARHR, which is, of course, not desirable. Diversity score, seems okay, maybe is just slightly high. What refers to novelty, its score is very close to the max possible novelty score for our data which is something undesirable. So, unfortunately, the algorithm is faulty in almost all aspects.

Let's now look at the top-10 recommendation for our test user to see if they are as inadequate as we might expect. The results are presented in Table 3.8 shown below:

	k=10		
Rank	**Item**	**Type**	**Course name**
1	Implementing DFS using a Stack	U	
2	Adding Matrices	U	

3	Mood Today	U	Coding Challenges
4	Users Route (GET)	U	Node.js
5	National Dish	U	Coding Challenges
6	Implementing a Hash Table	U	
7	How Often to Commit	U	
8	Implementing Binary Search	U	
9	Writable Stream Functions	U	Node.js
10	Implementing a Graph using Adjacency List	U	

Table 3.8: Top-10 recommendation results for the test user produced by the item-based top-n recommender algorithm for $k = 10$

As it was expected, the results are totally inappropriate: all recommended iterms are unpopular and they are far from what our test user would find interesting to learn about.

Let's now move to the Item-based KNN and look at its results. The main steps of the algorithm can be summarized as follows:

1. Construct an *item-to-item similarity* matrix based on the user-item ratings matrix.
2. Take the k-most similar items to the item i rated by the user u
3. Calculate the weighted average, i.e.

$$\widehat{r_{ui}} = \frac{\sum_k sim\_score(i,k) \times r_{uk}}{\sum_k sim\_score(i,k)}$$

The evaluation results of the Item-based KNN for our data are presented in Table 3.9 shown below:

k\Metrics	**RMSE**	**MAE**	**HR**	**cHR**	**ARHR**	**User-coverage**	**Diversity**	**Novelty**
10	0.2971	0.1520	0.0031	0.0031	0.0006	0.4035	0.4638	1004.9081

Table 3.9: Evaluation results for the Item-based KNN for $k = 10$

The accuracy scores RMSE and MAE, look impressive in comparison to all other models discussed so far. In fact, the Item-based KNN has the lowest RMSE and MAE scores among all other algorithms. So, the algorithm seems good in terms of doing rating predictions. However, what refers to the HR, cHR, and ARHR scores, they are pretty low, suggesting that the the Item-based KNN is probably not good at generating proper top-n recommendations.

The scores for both user-coverage and diversity seem okay, but the novelty score is higher from what is expected to be appropriate.

Now let's look at the top-10 recommendation produced by the Item-based KNN for our test user. The results are presented in Table 3.10 shown below:

	k=10		
Rank	**Item**	**Type**	**Course name**
1	Alter, Drop, Rename a Table	L	SQL Fundamentals
2	Doing Math	L	Ruby Tutorial
3	Views	L	SQL Fundamentals
4	UNION	L	SQL Fundamentals
5	Predefined Macros	U	Latest Additions
6	Working with Strings	L	Swift 4 Fundamentals
7	Password Validation	U	Latest Additions
8	Functions as Objects	L	Python 3 Tutorial
9	More on Function Arguments	L	Python 3 Tutorial
10	__main__	L	Python 3 Tutorial

Table 3.10: Top-10 recommendation results for the test user produced by the Item-based KNN for $k = 10$

As anticipated, the items in the top-10 recommendation are not satisfying and do not seem to be something our test user would enjoy looking at. Of course, it is not correct to reject the possibility that lessons from the "SQL Fundamentals" course might be interesting to the user, nevertheless, it is more reasonable to assume that lessons/user-lessons from JavaScript and JS-

related topics are more interesting to the user and therefore it would have been better to have JS and/or JS-related topics appearing in the first few ranks of the top-10 recommendation list.

In addition, as we can see from Table 3.10, the lesson called “Doing Math” from the course “Ruby Tutorial” is in the second position in the overall list. While it seems to be an inappropriate recommendation, it is a perfect example for the explanation of the notion of *serendipity* – the occurrence wherein the items recommended are somewhat unexpected, and therefore there is a modest element of lucky discovery, as opposed to obvious recommendations.

In conclusion, as we have seen neither the item-based top-n recommender nor the Item-based KNN were good at generating suitable top-n recommendations. However, it is worth mentioning, that the Item-based KNN is doing a good job in predicting ratings and is in fact the best rating predictor algorithm among all others discussed so far.

3.2.3 Matrix Factorization: Singular Value Decomposition (SVD)

Before diving into the results, let's first get familiar with matrix factorization and with Singular Value Decomposition (SVD), in particular.

In the context of recommender systems, matrix factorization refers to the family of models which aim to solve the matrix completion problem via approximating the given ratings matrix R of size $m \times n$ through decomposing it into matrices U of size $m \times k$ and V of size $n \times k$, where $k \ll \min\{m, n\}$, as follows:

$$R \approx UV^T$$

Singular Value Decomposition (SVD) is a form of matrix factorization in which the columns of U and V are constrained to be mutually orthogonal. For ease of discussion, for now, let's assume that the given ratings matrix is fully specified. Then, one can approximate the ratings matrix R of size $m \times n$ by using *truncated* SVD of rank $k \ll \min\{m, n\}$. Truncated SVD is computed in the following way:

$$R \approx Q_k \Sigma_k P_k^T$$

where Q_k, Σ_k, and P_k are matrices of size $m \times k$, $k \times k$, and $n \times k$ respectively. The matrices Q_k and P_k respectively contain the k largest eigenvectors of RR^T and $R^T R$, whereas the (diagonal) matrix Σ_k contains the square roots of the k largest eigenvalues of either matrix along its diagonal.

Note that although, SVD decomposes R into three matrices rather than two, it is indeed inherently a matrix factorization, since the diagonal matrix Σ_k can be absorbed in either Q_k or P_k to get only two matrices. By convention, in SVD, matrices U and V are defined as follows:

$$U = Q_k \Sigma_k$$
$$V = P_k$$

Now, let's look at the optimization problem proposed by the SVD algorithm:

$$Minimize\ J = \frac{1}{2}\|R - UV^T\|^2 = \frac{1}{2}\sum_{(i,j)}(r_{ij} - \overrightarrow{u_i} \cdot \overrightarrow{v_j})^2 \qquad \text{(F1)}$$
$$subject\ to:$$
$$Columns\ of\ U\ are\ mutually\ orthogonal$$
$$Columns\ of\ V\ are\ mutually\ orthogonal$$

r_{ij} is the rating of item j given by user i, $\vec{u}_i$ is the ith row of the matrix U and $\vec{v}_j$ is the jth row of the matrix V.

As we can see, the problem is to minimize the square of the Frobenius norm of the residual matrix $(R - UV^T)$ over matrices U and V, subject to columns of U and V being mutually orthogonal.

So far, we have assumed R to be a fully specified matrix, however, in reality, R is a sparse matrix. Consequently, a natural question arises: how do we solve $(F1)$ when the matrix R is incompletely specified?

One of the ways to overcome this problem is imputation. Imputation, however, is not always a good idea, since it significantly increases the amount of data and also, inaccurate imputation might distort the data considerably. For these reasons, it is a more common approach to do the following:

$$Minimize\ J = \frac{1}{2}\sum_{(i,j)\in S}(r_{ij} - \overrightarrow{u_i}\cdot\overrightarrow{v_j})^2 + \underbrace{\frac{\lambda_1}{2}\sum_{i=1}^{m}\|\overrightarrow{u_i}\|^2 + \frac{\lambda_2}{2}\sum_{j=1}^{n}\left\|\overrightarrow{v_j}\right\|^2}_{regularization\ for\ avoiding\ overfitting} \qquad (F2)$$

$$subject\ to:$$
$$Columns\ of\ U\ are\ mutually\ orthogonal$$
$$Columns\ of\ V\ are\ mutually\ orthogonal$$

where S is the set of all specified entries in the ratings matrix R. So to learn the vectors $\{u_i\}_{i=1}^{m}$ and $\{v_j\}_{j=1}^{n}$, we are using only the set of (i,j) pairs for which r_{ij} is known.

Since $F2$ is a constrained optimization problem, learning algorithms such as Stochastic Gradient Descent (SGD) or Alternative Least Squares (ALS) are not applicable. However, a variety of modified methods such as Projected Gradient Descent exist to tackle the problem.

To test the SVD algorithm on our data, the SVD implementation of the Surprise library was used. It is important to note, however, that the SVD of the Surprise is not solving $(F2)$ but the unconstrainted optimization version of $(F2)$, i.e. no conditions are put on the matrices U and V. In addition, it is noteworthy that the Surprise's SVD implementation has the option of adding bias terms to the objective loss function, thus trying to capture more of the signal existing in the data. This is the loss function Surprise is trying to minimize:

$$Minimize\ J = \frac{1}{2}\sum_{(i,j)\in S}(r_{ij} - \mu - b_i - b_j - \overrightarrow{u_i}\cdot\overrightarrow{v_j})^2 + \underbrace{\lambda\left(\sum_{i=1}^{m}\|\overrightarrow{u_i}\|^2 + \sum_{j=1}^{n}\left\|\overrightarrow{v_j}\right\|^2 + b_i^2 + b_j^2\right)}_{regularization\ for\ avoiding\ overfitting} \quad (F3)$$

where μ is the global mean of the ratings and the parameters b_i and b_j are the bias terms indicating the observed deviations of user i and item j, respectively, from the average. In $(F3)$, not only the system of vectors $\{u_i\}_{i=1}^{m}$ and $\{v_j\}_{j=1}^{n}$ are learned but also the terms $\{b_i\}_{i=1}^{m}$ and $\{b_j\}_{j=1}^{n}$.

Now, let's look at the evalution results presented in Table 3.11 which is shown below:

Train RMSE	Test RMSE	Train MAE	Test MAE	HR	cHR	ARHR	User-coverage	Diversity	Novelty
0.3901	0.3950	0.2124	0.2174	0.0002	0.0002	0.0001	0.9989	0.1056	1252.6700

Table 3.11: Evaluation results for the SVD algorithm

As we can see, the results are quite disappointing, albeit the initial expectation which was very high given the immense reputation of matrix factorization methods. Neither accuracy nor hit rate scores are good. Moreover, the diversity score is too low and the novelty score is too high, suggesting that top-n recommendations usually consist of very similar and unpopular items. In fact, user-coverage is the only metric that has a good score.

Now let's look at the top-10 recommendation presented in Table 3.12, to visually see if SVD is performing as poor as the evaluation results suggest.

Rank	Item	Type	Course name
1	Users Route (POST)	U	Node.js
2	MongoDB Model	U	Node.js
3	Body Parser	U	Node.js
4	Timers	U	Node.js
5	Get User By ID	U	Node.js
6	State	U	React.js
7	Cross-Origin Resource Sharing	U	Node.js
8	Redux: Core Concepts	U	React.js
9	Making API Requests	U	Node.js
10	State & Props: Assignment Solution	U	

Table 3.12: Top-10 recommendation results for the test user produced by the SVD algorithm

At first glance, it may seem that for our test user, who as his/her history suggests has a strong affinity towards JavaScript, the recommended items are not bad at all. However, the facts that the recommendations are very similar and also unpopular, make the top-10 recommendation rather inadequate.

In addition, based on the top-10 recommendation produced by the SVD algorithm for a user other than the test user, it was revealed that our SVD algorithm is not really learning the patterns in the data: a user with a history of machine learning and python classes got very similar recommendations to that of our test user.

To conclude, the SVD algorithm did not produce the results one would anticipate. However, there is a probable explanation for it: SVD works well only when its parameters are properly tuned which was unfortunately impossible due to computational power limitations. The only parameter which was changed from its default was the regularization parameter and it was done in order to prevent overfitting.

3.3 Evaluation: Comparison of the Algorithms

In order to compare the algorithms that have been applied on the data, let's look at the Table 3.13 which summarizes the evaluation results for each of them. The table[8] is shown below.

Algorithms\Metrics	**RMSE**	**MAE**	**HR**	**cHR**	**ARHR**	**User-coverage**	**Diversity**	**Novelty**
Content-Based KNN (k=5)	0.3916	0.1903	0.0345	0.0342	0.0161	0.3430	0.1213	640.8535
Content-Based KNN (k=20)	0.3786	0.1934	0.0217	0.0217	0.0098	0.2185	0.1547	771.8039
User-Based Top-N	-	-	0.2183	0.2175	0.1455	-	0.0590	115.1457
User-Based KNN	0.4221	0.1947	0.0002	0.0002	0.0000	1.0000	0.2432	1238.2224
Item-Based Top-N	-	-	0.0004	0.0004	0.0001	-	0.6121	1230.1907
Item-Based KNN	0.2971	0.1520	0.0031	0.0031	0.0006	0.4035	0.4638	1004.9081
SVD	0.3929	0.2139	0.0002	0.0002	0.0001	0.9989	0.1056	1252.6700
Random	0.5980	0.4137	-	-	-	-	-	-

Table 3.13: Summary of the evaluation results for all algorithms

As we can see from the table, among our rating predictor algorithms, Item-Based KNN is the best one in terms of both RMSE and MAE scores. While choosing the winner in this category was obvious, the same is not true about the worst algorithm. This is because if compare the algorithms based on the MAE scores, then the worst algorithm is SVD, whereas comparing based on the RMSE scores the worst is User-Based KNN.

In general, RMSE is a better metric when outlier evaluation is important, since RMSE is more significantly affected by large error values than MAE. On the other hand, MAE is a better reflection of the average error. So, both RMSE and MAE have their unique importance and this is why it is difficult to choose between the SVD and the User-Based KNN.

[8] Note that besides the evaluation results of our algorithms, the RMSE and MAE scores of an algorithm called *Random* are also included in the table. Random is the algorithm called NormalPredictor in Surprise library, which is doing rating predictions by randomly generating a number from normal distribution with parameters μ and σ estimated from the training data using Maximum Likelihood Estimation. Random has been included to show that all our algorithms are doing much better rating predicition.

If compare the algorithms based on the hit rate scores, then the absolute, indisputable winner is the User-Based Top-N. It has really impressive HR and consequently cHR and ARHR scores which explain why it was so good in recommending items to our test user. While it is totally justified to call the User-Based Top-N the best algorithm in generating top-n recommendations given its results, it is important to note that the User-Based Top-N has some significant drawbacks. The first drawback is that it has very low diversity. We have seen this in the top-10 recommendation produced for our test user, but the problem became much more apparent when top-10 recommendation was generated for another user: a user with a history of mainly python classes got only recommendations from "Python 3 Tutorial". The next shortcoming is that the User-Based Top-N has quite low novelty score, meaning that the algorithm is mostly recommending popular items, paying very little attention to unpopular ones. This is not very good, since in many cases, unpopular items can be of huge interest to users and also lead to serendipitous discoveries. And the final major drawback is that like the User-Based KNN, the User-Based Top-N needs to calculate a user-to-user similarity matrix which can take huge amount of memory.

Now, let's compare the algorithms based on user-coverage, diversity, and novelty scores. From Table 3.13 we can clearly see that the User-based KNN and the SVD outperformed the other algorithms with regard to the user-coverage score. Both the User-based KNN and the SVD have extremely high user-coverage scores, 1 and 0.9989 respectively, while others algorithm did not even achieve the score of 0.5

What refers to the diversity, if assume that 0.5 is the most desirable score, then the winner is Item-based KNN. The two Content-Based KNN algorithms and SVD have quite low diversity scores, but the most problematic is User-Based Top-N's situation, which has extremely low diversity as was mentioned earlier.

And lastly, taking novelty as the basis for comparison, the best algorithm is Content-based KNN with k=5. It has novelty score of 640.8535 which is almost equal to average novelty score for our data (average score = 659), meaning that both popular and unpopular items are being recommended. The worst algorithm in this category is User-Based Top-N.

To sum up, as we have seen, no single algorithm is the best in all aspects. The Item-based KNN is the winner in rating predictions, the User-Based Top-N is the best when it comes to hit rates, while User-Based KNN (or SVD), Item-based KNN and Content-based KNN with k=5 are the best algorithms in terms of user-coverage, diversity, and novelty scores respectively. However, if we were to choose a single algorithm to serve as our recommender system, then it would definitely be the User-Based Top-N. This is because, although it has poor diversity and novelty scores and is not predicting ratings, it has the ability to generate relevant top-n recommendation better than any other algorithm.

Conclusion & Further Developments

As we have seen based on our research experience on the data of “SoloLearn: Learn to Code Free” mobile application, designing and evaluating recommender systems is a very subtle and laborious task. In particular, when the available data is in the form of implicit feedback, huge amount of time and effort needs to be spent on retrieving data as well as doing the necessary data processing before algorithms can be tested.

It came as a surprise, but as we have seen, the best algorithm in terms of generating top-n recommendations was the User-Based Top-N. It had a pretty impressive hit rate scores and produced great recommendations for our test user. Nevertheless, despite having the ability to produce relevant recommendations, it had a shortcoming of recommending only mostly similar and popular items. So, we can say that it is a quite good recommender but not excellent.

It is important to mention that although a lot of work has been done on designing, testing, and evaluating different recommender systems for the “SoloLearn: Learn to Code Free” mobile app with the ultimate goal of developing a proper recommender system, there is still much work left to do and room for improvements.

First of all, there are many other algorithms that can be tested. One such example is SVD++. It is in fact the same as SVD with the difference that it has one additional term in its loss function which tries to model the idea that merely rating an item at all, is some sort of implicit interest in the item, no matter what the rating was. However, like SVD, SVD++ needs to have its parameters properly tuned before one can draw a valid conclusion about its suitability and performance. In addition to SVD++, artificial neural networks such as Restricted Boltzmann Machines can be applied to solve the matrix completion problem.

Secondly, more training data can be used to test the algorithms. Recall that the data of only 10,000 users was used, in spite of the data availability of more than 3 million users. Having more training data can immensely improve the results. The only problem is that for processing such massive amounts of data, a cluster of computers is required that needs to be programmed to perform the necessary work. One can use, Apache Spark, for example, for this purpose. Apache Spark is an open-source distributed general-purpose cluster-computing framework.

And finally, another important thing that can be done, is building hybrid recommender systems. As we have seen in “3.3 Evaluation: Comparison of the Algorithms”, different algorithms may have different strengths and weaknesses. Therefore, the use of hybrid approaches can help combine the power of different algorithms and thus get better results – better than any single algorithm can produce alone.

Appendix I

Language	Application	Client-side	Web	Server-side	Web-Synergy	Moble-development	General-purpose	Data Science	System	Imperative	Object-oriented	Functional
C	1	0	0	0	0	0	1	0	1	1	0	0
C++	1	0	0	0	0	0	0	0	1	1	1	1
C#	1	1	1	1	0	1	1	0	0	1	1	1
CSS	0	1	1	0	1	0	0	0	0	0	0	0
HTML	0	1	1	0	1	0	0	0	0	0	0	0
Java	1	0	1	1	0	1	1	0	0	1	1	1
JavaScript	0	1	1	1	1	1	0	0	0	1	0	1
Kotlin	1	1	1	1	0	1	0	0	0	1	1	1
PHP	0	0	1	1	1	0	0	0	0	1	1	1
Python	1	0	1	1	0	0	1	1	0	1	1	1
Ruby	1	0	1	1	0	0	0	0	0	1	1	0
jQuery	0	1	1	0	1	0	0	0	0	0	0	1
SQL	0	0	0	1	0	0	0	1	0	0	0	0
Swift	1	0	0	0	0	1	1	0	0	1	1	1
R	1	0	0	0	0	0	0	1	0	1	1	1

Appendix II

Item	Type	Course name	Rating
Valentine's Day	User-Lesson	Coding Challenges	1.3822
What is AngularJS?	User-Lesson	AngularJS	1.3822
Node.js Cheat Sheet	User-Lesson	Coding Career	1.3822
Introduction to React.js	User-Lesson	Latest Additions	1.3822
Introducing the Box Model	Lesson	CSS Fundamentals	1.3822
Understanding the Box Model	Lesson	CSS Fundamentals	1.3822
Borders	Lesson	CSS Fundamentals	1.3822
Width and Height	Lesson	CSS Fundamentals	1.3822
background-color	Lesson	CSS Fundamentals	1.73824
background-image	Lesson	CSS Fundamentals	1.3822
background-repeat	Lesson	CSS Fundamentals	1.73824
background-attachment	Lesson	CSS Fundamentals	1.73824
Styling the Lists	Lesson	CSS Fundamentals	1.3822
Styling the Tables	Lesson	CSS Fundamentals	1.73824
Styling the Links	Lesson	CSS Fundamentals	1.3822
Customizing the Mouse Cursor	Lesson	CSS Fundamentals	1.3822
Module 3 Quiz	Lesson	CSS Fundamentals	1.3822
What is CSS?	Lesson	CSS Fundamentals	1.3822
Inline, Embedded, External CSS	Lesson	CSS Fundamentals	1.3822
CSS Rules and Selectors	Lesson	CSS Fundamentals	1.3822
CSS Comments	Lesson	CSS Fundamentals	1.3822
Style Cascade and Inheritance	Lesson	CSS Fundamentals	1.73824
Module 1 Quiz	Lesson	CSS Fundamentals	1.3822
font-family	Lesson	CSS Fundamentals	1.3822

font-size	Lesson	CSS Fundamentals	1.3822
font-style	Lesson	CSS Fundamentals	1.3822
font-weight	Lesson	CSS Fundamentals	1.3822
font-variant	Lesson	CSS Fundamentals	1.3822
color	Lesson	CSS Fundamentals	1.3822
Aligning Text Horizontally	Lesson	CSS Fundamentals	1.3822
Aligning Text Vertically	Lesson	CSS Fundamentals	1.73824
text-decoration	Lesson	CSS Fundamentals	1.73824
Indenting the Text	Lesson	CSS Fundamentals	1.3822
text-shadow	Lesson	CSS Fundamentals	1.73824
text-transform	Lesson	CSS Fundamentals	1.3822
letter-spacing	Lesson	CSS Fundamentals	1.3822
word-spacing	Lesson	CSS Fundamentals	1.3822
white-spacing	Lesson	CSS Fundamentals	1.3822
Module 2 Quiz	Lesson	CSS Fundamentals	1.3822
The display Property	Lesson	CSS Fundamentals	1.3822
The visibility Property	Lesson	CSS Fundamentals	1.3822
Positioning	Lesson	CSS Fundamentals	1.3822
What is DOM?	Lesson	JavaScript Tutorial	1.3822
Selecting Elements	Lesson	JavaScript Tutorial	1.97458
Changing Elements	Lesson	JavaScript Tutorial	1.73824
Introduction to JavaScript	Lesson	JavaScript Tutorial	1.3822
Creating Your First JavaScript	Lesson	JavaScript Tutorial	1.3822
Module 1 Quiz	Lesson	JavaScript Tutorial	1.3822
Math Operators	Lesson	JavaScript Tutorial	1.3822
Assignment Operators	Lesson	JavaScript Tutorial	1.3822
Comparison Operators	Lesson	JavaScript Tutorial	1.3822
Logical or Boolean Operators	Lesson	JavaScript Tutorial	1.3822

String Operators	Lesson	JavaScript Tutorial	1.3822
Module 2 Quiz	Lesson	JavaScript Tutorial	1.73824
The if else if else Statement	Lesson	JavaScript Tutorial	1.3822
The switch Statement	Lesson	JavaScript Tutorial	1.3822
The For Loop	Lesson	JavaScript Tutorial	1.3822
The While Loop	Lesson	JavaScript Tutorial	1.3822
The Do...While Loop	Lesson	JavaScript Tutorial	1.3822
Break and Continue	Lesson	JavaScript Tutorial	1.3822
Module 3 Quiz	Lesson	JavaScript Tutorial	1.3822
User-Defined Functions	Lesson	JavaScript Tutorial	1.3822
Function Parameters	Lesson	JavaScript Tutorial	1.3822
Using Multiple Parameters with Functions	Lesson	JavaScript Tutorial	1.3822
The return Statement	Lesson	JavaScript Tutorial	1.3822
Alert, Prompt, Confirm	Lesson	JavaScript Tutorial	1.3822
Module 4 Quiz	Lesson	JavaScript Tutorial	1.3822
Introducing Objects	Lesson	JavaScript Tutorial	1.8713
Creating Your Own Objects	Lesson	JavaScript Tutorial	1.73824
Object Initialization	Lesson	JavaScript Tutorial	1.73824
Adding Methods	Lesson	JavaScript Tutorial	1.3822
Module 5 Quiz	Lesson	JavaScript Tutorial	1.73824
Arrays	Lesson	JavaScript Tutorial	1.73824
Other Ways to Create Arrays	Lesson	JavaScript Tutorial	1.73824
Array Properties & Methods	Lesson	JavaScript Tutorial	1.3822
Associative Arrays	Lesson	JavaScript Tutorial	1.73824
The Math Object	Lesson	JavaScript Tutorial	1.73824
The Date Object	Lesson	JavaScript Tutorial	1.73824
Module 6 Quiz	Lesson	JavaScript Tutorial	1.3822

Bibliography

- *Aggarwal, C. C. (2016). Recommender Systems: The Textbook. Cham: Springer.*
- *Ricci, F., Rokach, L., Shapira, B., & Kantor, P. B. (2011). Recommender Systems Handbook. Boston, MA: Springer US.*
- *Y. Koren, R. Bell, & C. Volinsky, "Matrix Factorization Techniques for Recommender Systems", IEEE CS Press, 2009*
- *Kane, F., 2019 "Building Recommender Systems with Machine Learning and AI", Udemy*