



NTNU – Trondheim
Norwegian University of
Science and Technology

Design of a Hybrid Recommender System: A Study of the Cold-Start User Problem

Sigurd Støen Lund
Øystein Tandberg

Master of Science in Informatics

Submission date: May 2015

Supervisor: Helge Langseth, IDI

Co-supervisor: Professor Agnar Aamodt, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Recommender systems are used to help users discover the items they might be interested in, especially when the number of alternatives is big. In modern streaming websites for music, movies, and TV-shows, E-commerce, social networks, and more, recommender systems are widely used. These recommender systems are often looking at the ratings on items for the current and other users, and predicting a rating on the items the user have not seen. Others match the content of an item itself against a user profile. A mix of the two is often used to make the predictions more accurate, and this can also help to the problem when a new user sign up where we have no knowledge about him. This issue, is a well-known problem for recommender systems often described as the cold-start problem, and much research has been done to find the best way to overcome this.

In this thesis, we look at previous approaches to recommender systems and the cold-start problem in particular. We have developed our application, Eatelligent, which is recommending dinner recipes based on our study of previous research. Eatelligent has been designed to examine how we can approach the cold-start problem efficiently in a real world application, and what kind of feedback we can collect from the users.

Sammendrag

Anbefalingssystemer brukes til å hjelpe brukere å oppdage ting de kan være interessert i, og spesielt når det finnes mange alternativer å velge fra. I moderne strømmingstjenester for musikk, tv-serier og film, eller nettbutikker og sosiale medier, er anbefalingssystemer mye brukt. Disse anbefalingssystemene ser ofte på historikk i form av vurderinger for en bruker, og predikterer en vurdering på objektene brukeren ikke har vurdert selv. Andre systemer sammenlikner innholdet av et objekt mot brukerens profil. Ofte brukes en kombinasjon av disse to metodene for å gjøre prediksjonene enda mer nøyaktige. Dette kan også hjelpe til å løse utfordringen med nye brukere som man ikke har noen kunnskap om. Dette problemet omtales ofte som kaldstart-problemet, og mye arbeid er lagt ned tidligere for å overkomme dette.

I denne masteroppgaven ser vi på tidligere tilnærminger for anbefalingssystemer og kaldstart-problemet spesielt. Vi har laget vår egen applikasjon, Eatelligent, som er en middagsoppskrift-anbefaler basert på vårt studie av tidligere arbeid. Eatelligent er designet for å utforske hvordan vi kan tilnærme oss kaldstart-problemet effektivt i en reel applikasjon, og hvilken type tilbakemeldinger vi kan samle fra brukerne for å overkomme dette.

Preface

This Master's thesis is written by Sigurd Støen Lund and Øystein Tandberg from August 2014 to June 2015 at the Norwegian University of Science and Technology (NTNU). The thesis is the final delivery of the master program in Informatics, with specialization in intelligent systems.

We came up with the idea for the thesis ourselves, wanting to build something related to recommender systems in a real world application. Our supervisors liked the idea and helped us making this into a thesis.


We would like to thank our supervisors, Helge Langseth and Agnar Aamodt, for constructive feedback and discussions, and for always being in pleasant mood throughout the year. We would also like to thank the fellow students in room 363, Korp, for a good working atmosphere, along with the other students taking lunch breaks on the second floor.

We would also like to thank Matprat AS for letting us use their recipes in our application.

Trondheim, Friday 29th May, 2015



Sigurd Støen Lund



Øystein Tandberg

Contents

Abstract	i
Sammendrag	iii
Preface	v
1 Introduction and Overview	3
1.1 Background and Motivation	3
1.2 Goals and Research Objectives	4
1.3 Thesis Structure	5
1.4 Project Scope	5
2 Background Theory and Motivation	7
2.1 Recommender Systems	7
2.1.1 Content-Based Filtering	8
2.1.2 Collaborative Filtering	9
2.1.3 Hybrid Recommender Systems	11
2.1.4 Context-Aware Recommender Systems	12
2.2 Feedback	16
2.2.1 Explicit Feedback	16
2.2.2 Implicit Feedback	17
2.3 The Cold-Start Problem	19
2.3.1 Cold-Start User Problem	19

2.3.2	Cold-Start Item Problem	23
2.4	Case-Based Reasoning	23
2.4.1	The CBR Principles	24
2.4.2	CBR as a Recommender System	26
2.5	Examples of Earlier Projects	27
2.5.1	Amazon.com	27
2.5.2	DieToRecs	28
2.5.3	CHEF	29
2.5.4	JULIA	29
2.5.5	Intelligent Food Planning: Personalized Recipe Recommendation	30
2.5.6	Food Recommendation Using Ontology and Heuristics	30
2.6	Relating the Theory Towards Eatelligent	31
2.6.1	Recommender System	31
2.6.2	Cold-Start	32
2.6.3	Feedback	33
2.6.4	Taking Advantage of the Contextual Information	34
3	Architecture/Model	37
3.1	Requirements for Design	37
3.2	System Overview	39
3.3	Implementation	39
3.3.1	Client	40
3.3.2	Backend	43
3.4	Recommender System	51
3.4.1	Algorithms	52
3.4.2	Feedback in the Client	58
3.4.3	Cold-Start Item Problem	59

- 3.4.4 Composition of Recommendations 59
- 3.5 How it Works 60
 - 3.5.1 A New User Receiving a List of Recommendations . . . 60
- 3.6 Making Eatelligent Public 65
- 4 Experiments and Results 67**
 - 4.1 Experimental Plan 67
 - 4.2 Experimental Setup 68
 - 4.3 Experimental Results 69
 - 4.3.1 Yes/No ratings 69
 - 4.3.2 Star Ratings 70
 - 4.3.3 Comparing Yes/No and Star Ratings 73
 - 4.3.4 Cold-Start Questions 73
- 5 Evaluation and Conclusion 77**
 - 5.1 Evaluation and Discussion 77
 - 5.1.1 RO1 - Gain an understanding of what it means to recommend food recipes 77
 - 5.1.2 RO2 - Study and evaluate different strategies to recommend items for a new user using both collaborative and content-based approaches 78
 - 5.1.3 RO3 - Study existing solutions to the cold-start problem 79
 - 5.1.4 RO4 - Study what kind of data we can collect from the application used by a user over a period of time that is relevant to build more knowledge about the user . . 79
 - 5.1.5 RO5 - Study the challenges that arise when building an application with a recommender system from scratch 80
 - 5.2 Contributions 80
 - 5.3 Future Work 82

A SQL-scheme	85
B Endpoints	91
B.1 GET	91
B.2 PUT	92
B.3 POST	92
B.4 DELETE	93
C Recipe Features	95
C.1 Ingredient Tags	95
C.2 Recipe Tags	96
C.3 Full Recipe Description	97
C.3.1 Paprika Chicken with Asparagus	97
C.3.2 Babi Asam Manis	98
C.3.3 Asian Omelet	99
Bibliography	100

List of Figures

2.1	We want to learn the function between the user and the items he like the most	8
2.2	Multidimensional model for the User \times Item \times Time recommendation space [5]	14
2.3	The CBR cycle	24
2.4	The mapping from problem to solution. The dotted arrow shows a new case and its solution adapted from the previously solved cases.	26
3.1	System overview	40
3.2	Class hierarchy for the mobile client	41
3.3	Entity relation diagram	44
3.4	The sign up process	47
3.5	The user answer 1/5 questions	48
3.6	The users answers “yes” or “no” to a recommendation	49
3.7	Detailed view of the recipe	50
3.8	Rating the recipe	51
3.9	Illustration of the recipe tag tree	57
3.10	Illustration of the ingredient tag tree	57
4.1	“Yes/no” ratings per algorithm	69
4.2	Normalized “Yes/no” ratings per algorithm	70

4.3	Star ratings	71
4.4	Percentage of star ratings for each algorithm	72
4.5	Comparison between yes/no and star ratings. The data for each recipe corresponds vertically between the two charts. In the bottom chart, the impulses represent the number of ratings, and the bars show the average star rating.	74

List of Tables

2.1	The goal is to predict the missing votes for the active user, e.g., User B	10
2.2	Common types of explicit and implicit feedback	17
2.3	Evaluation of different strategies from A. M. Rashid et al.	22
3.1	The different response codes used by the server	46
3.2	The weighting from the cold-start questions	53
3.3	The impact of star ratings	55
3.4	An example on a composed recommendation list	60
3.5	The current state of the weights after the cold-start questions.	61
3.6	The current state of the weights after the “yes/no” ratings	62
3.7	The updated weights after a rating of “Paprika Chicken with asparagus”	64
4.1	The number of ratings for each algorithm	71
4.2	Answers to the cold-start questions	75
4.3	Recommendations made by each algorithm based on cold-start questions	76

List of Algorithms

1	Nearest User Rated Recipes	54
2	Content-based filtering	56

Listings

- 2.1 The KNN algorithm 25
- 3.1 Sign up request 47
- 3.2 Sign up response 47
- 3.3 Cold-Start Request 48
- 3.4 Cold-Start Response 48
- 3.5 Recommendations 49
- 3.6 Recipe 50
- 3.7 Rating request 51
- 3.8 Rating response 51
- 3.9 First list of recommendations to a user 63
- 3.10 New recommendation after interactions 64

Chapter 1

Introduction and Overview

Recommender systems help people to make better decisions in a world filled with choices. Since the recommender systems in most cases are targeted towards individuals, it is a problem to recommend items to a new user you barely know anything about, or a new item you do not know which type of user will like. This is known as the cold-start problem. In this thesis, we will study different approaches to the cold-start problem in a problem domain where we have no knowledge about new users. We will also study how feedback can be collected, to evaluate the recommender engine, and use the feedback for better recommendations in the future.

By exploring this in a real world application we develop on our own, we get the opportunity to be more flexible about how we structure our data, what type of knowledge we try to extract from the user, and what feedback we collect while the user uses the system. A well-suited domain for this research is the recommendation of food recipes, which will be implemented as an application from now called Eatelligent.

1.1 Background and Motivation

Composing recommendations of items to a user is usually done by one of two strategies, or both of them combined. Either by looking at the content of the items itself and matching it against a user profile, or predicting a rating based on other users ratings on the items. Both of the strategies have challenges

relating to the cold-start problem, and a combination of the two is often used to overcome some of these. In this thesis, we want to try combinations of different methods, and see how these perform on new users.

To build knowledge about a new user, we want to study different methods to collect data about how the user is using the application and ask the user to add additional information. We also want to design, structure, and describe the data ourselves, to be able to build knowledge about the items and users in an efficient way. The only way we are free to do all this without any restrictions is if we develop our own application. Many datasets exist to be used for testing recommender systems, but those datasets contain historical data, typical user ratings on items like books or movies. By building a real world application, we can collect both implicit and explicit ratings, look at the content of the items ourselves and therefore be more flexible in the design of the recommender system.

A good test domain would be something where you start with no knowledge about the user and have to extract some knowledge and turn it into predictions. Thinking of a domain where people make choices on a regular basis and there are a lot of options out there that takes some effort for a person to explore, we thought of what people are deciding to make for dinner. There exist a lot of websites that contains food recipes, but they are not necessarily helping the user to make an easy choice of what to go for without having to browse through a lot of recipes. By implementing an application with dinner recipes along with a recommender system, we could help people save time on a daily basis. It is also hard to know what kind of food an unknown person likes, so it gives us a good domain for studying the cold-start problem. In a world where almost everybody have smartphones and more and more of the Internet activity is happening on those devices, we decided to go for a mobile application, to test the software as a real world application.

1.2 Goals and Research Objectives

The goal for this thesis is to study methods for a recommender system in a real world problem situation, explore the challenges with recommending items to new users, and implement and evaluate different strategies to solve this problem. Different recommender strategies will be tested to see which one is most accurate. The different methods will be assessed by star ratings

from the users, and also by looking at which recipes the user looks at in detail. Some of the recommended recipes will be drawn from a uniform random distribution to be used as a baseline to compare the other methods.

Below are the research objectives for this thesis:

- **RO1** Gain an understanding of what it means to recommend food recipes.
- **RO2** Study and evaluate different strategies to recommend items for a new user using both collaborative and content-based approaches.
- **RO3** Study existing solutions to the cold-start problem.
- **RO4** Study what kind of data we can collect from the application used by a user over a period of time that is relevant to build more knowledge about the user.
- **RO5** Study the challenges that arise when building an application with a recommender system from scratch.

1.3 Thesis Structure

Chapter 2 describes the background material to cover the most relevant theory related to recommender systems and the cold-start problem, and material that is either considered or used in our application. In Chapter 3 we describe the design and implementation of the whole application, which is everything from how the server and client are structured to how we approach the recommendation and cold-start problem. Chapter 4 presents the results of our work, and in Chapter 5 we evaluate how this solution worked along with suggestions for future work.

1.4 Project Scope

The scope for this thesis is to study the nature of the cold-start problem within the described problem domain and investigate the results of our approach to solve this problem, by building an application for recommendation

of dinner recipes. This also includes studying previous research, to make sure we build this application while standing on the shoulders of the giants.

Since this application is built from scratch, evaluate the recommender system by the data we collect throughout our test period. The application is released on the open market so that the user can be any user with access to a smartphone. Eatelligent is available for free in App Store for iPhone users and Google Play for Android users.

Chapter 2

Background Theory and Motivation

In this chapter, we will present the subjects that will form the background for this thesis. The chapter covers research regarding recommender systems and its features from early stages up until recent studies. We start off by describing recommender systems in general, and the different approaches that are out there. Then we write about feedback followed by the cold-start problem, which both are relevant challenges in recommender systems, and that we will study deeper in this thesis. Case-based reasoning (CBR) is a way of problem-solving and can also be used as a strategy in a recommender system. Section 2.4 is therefore dedicated to this methodology. In Section 2.5 we describe some relevant projects to recommender systems and food recommendation in particular. The chapter ends with Section 2.6 relating the theory described earlier towards Eatelligent.

2.1 Recommender Systems

Recommender systems suggest items that should interest the user. The recommendations are usually based on knowledge about the user, either it is a user profile with preferences or history. Recommendations of items like movies, books, and music widely use this strategy. It is usually targeted to individuals and used when the number of alternatives is big.



Figure 2.1: We want to learn the function between the user and the items he like the most

In this section, we will discuss the different approaches to recommender systems that are common today.

The key task for every different recommender system can vary. One can be to sort a list of items in a personalized fashion. Another can be to pick a personalized top 10 list from a large item set. Recommender systems also help users widen their horizon by recommending new items, based on similar users or similar items the user liked before. Figure 2.1 illustrates the function between the user and items that a recommender engine want to learn. For a naive recommender system, it is an easy task to recommend the most popular items to all users. In this naive approach, the result is not personalized.

2.1.1 Content-Based Filtering

Content-based filtering methods analyzes the items itself and match them against a user profile [39]. The representation of an item is important in content-based recommender systems since the recommendation technique might be based on the representation. In a structured way, attributes like tags or categories can easily be stored, so it is easy to compute from them. Unrestricted text is more complicated, and by counting words and represent their importance, an unrestricted text can be transformed into structured data. This process is often done by representing a weight for each term TF-IDF (term frequency times inverse document frequency). Before this is done it is common to use techniques like stemming, which means to represent different versions of a word as one term. For instance, “comput” can represent the terms “computation” and “computers”.

Since the items are going to be matched to a user, the recommender system needs to have a way of computing a score for a user on the items. Pazzani et al. [39] describe two types of user profiles: (1) The user’s preferences are stored, e.g. in a food recommender a user might have “vegan” attached to his profile, to describe that he wants vegan dishes. Preferences like this could

be collected by having the user fill out a profile with checkboxes. (2) The history of the user interacting with the system. This information might be history like which items he has looked at before, ratings, or queries. This feedback can either be collected implicitly or explicitly.

Many classification learning algorithms can be used to learn the function that computes an estimate of the probability that a user likes an item, like Naive Bayes along with many others. We will not go into detail with these algorithms here.

2.1.2 Collaborative Filtering

Collaborative filtering is a recommender approach based on a collection of the user’s data like ratings, behaviors, and preferences, as well as analyzing these data and recommend items to a similar user. The task can be described as “To predict the utility of items to a particular user (the active user) based on a database of user votes from a sample or population of other users (the user database)” [11]. Collaborative filtering can be in different forms; memory-based or model-based methods.

In **memory-based algorithms**, we predict the votes of the active user based on some partial information regarding the active user and a set of weights calculated from the user database. The user database contains a set of votes $v_{u,i}$, meaning the vote for user u on item i . If I_u is the set of items on which user u has voted, then the mean vote for user u is as follows:

$$\bar{v}_u = \frac{1}{|I_u|} \sum_{i \in I_u} v_{u,i} \quad (2.1)$$

In memory-based algorithms, we predict the votes of the active user based on partial information regarding the active user and a set of weights from the user database. The assumed predicted vote for the active user u for item i , $p_{u,i}$, is a weighted sum of the votes of the users:

$$P_{u,i} = \bar{v}_u + k \sum_{u' \in U} w(u, u') (v_{u,u'} - \bar{v}_{u'}) \quad (2.2)$$

Where u is the current user, U is the set of users in the database with nonzero weights. The weights, $w(u, u')$, can describe the correlation or similarity

	Item A	Item B	Item C	Item D
User A		1		5
User B	1	?	3	?
User C	2		4	5

Table 2.1: The goal is to predict the missing votes for the active user, e.g., User B

between each user in the database and the active user. k is for normalization. In this thesis, we will cover the correlation weighting model. The correlation between two users u and u' is defined as follows [43]:

$$w(u, u') = \frac{\sum_i (v_{u,i} - \bar{v}_u)(v_{u',i} - \bar{v}_{u'})}{\sqrt{\sum_i (v_{u,i} - \bar{v}_u)^2 \sum_i (v_{u',i} - \bar{v}_{u'})^2}} \quad (2.3)$$

Where i is each item for which both user u and u' have recorded votes.

Model-based algorithms is using a probabilistic approach where the collaborative filtering task can be viewed as calculating the expected value of a vote, given what we know about the user. For the active user, we want to predict the votes for unobserved items. The probability expression for an item i and user u with the rated items for user u is I_u is as follows (This assumes that the votes are integers in range from 0 to m):

$$p_{u,i} = E(v_{u,i}) = \sum_{j=0}^m Pr(v_{u,i} = j | v_{uk}, k \in I_u) \quad (2.4)$$

The probability, Pr , in Equation 2.4 is the probability that the given user will rate a value of the input item, given the previously rated items. Bayesian networks or clustering models is two of the probabilistic models that can be used for model-based collaborative filtering.

Collaborative filtering helps with predicting ratings for an active user based on previous votes in a database of votes. The input data is very often a sparse matrix of votes for different items. The example in Table 2.1 is a small example showing ratings for some users, and where the data is sparse.

2.1.3 Hybrid Recommender Systems

A hybrid recommender system is used to help avoid certain limitations of content-based- or collaborative filtering methods. Collaborative methods have a relatively big weakness around the cold-start problem since the user needs to have rated some items to receive recommendations. A real-life problem domain is much more complicated than a straight movie recommender system [4]. A real business case will have significant amounts of parameters, where a hybrid will help learning a more complex model of the user, hence giving more accurate recommendations. Hybrid solutions are a mix of the methods described earlier. Since different methods can be combined, multiple hybrid solutions can be created. Burke [13] covers seven different approaches to combining recommenders/components:

Weighted hybrid recommender systems go through a training phase where the weights in the recommender system are learned from a training set during a training period. The output of this training phase is the weights of each component. The candidates from each component are weighted in respect of these numeric values each time a recommendation is done. The result is then either a union or intersection of these lists. This approach does not differentiate between users; there is an assumption that each component will have a consistent performance for all users.

Mixed hybrid recommenders present a composed list of items side-by-side from each of the components. There is no actual combining of the different recommender algorithms. The challenge in mixed hybrid is the ordering and confidence in each of the components composed together. A technique often includes merging the lists on a normalization the predicted rating.

A **switching** hybrid recommender have a confidence in each component and selects the recommendation that gives the best results at the given time. This method, similar to the mixed approach, does not combine the algorithms into a new one. A substantial part of a switching hybrid is the selection criteria. Both statistical confidence and external criteria can be used to select which recommender to select in each user case.

Cascading hybrids tries to create a strict hierarchy of recommender components. The different components are ranked, and the secondary ones fill in for the primary recommender when there is a broken tie.

Meta-level hybrid recommender systems are first going through a training

phase. This training phase is to train one recommender to have a concept of a learned model. The actual recommender then uses this model to produce recommendations for the end user. Not all pairs of different recommendation components are suitable for this hybrid.

Feature combination is a hybrid where one of the components uses its output as input to the actual recommender. For instance, the output from a collaborative filtering algorithm can be used as input to a content-based algorithm. This method achieves that the first component is a contributor giving a bias to the algorithm giving recommended list, every time.

Feature augmentation is a strategy where one component is used to produce a new augmented feature set to the next component. This approach also needs a training phase that learns the augmented profile. Feature augmentation can derive new content features at this stage that can be valuable when generating candidate items.

2.1.4 Context-Aware Recommender Systems

Most of the recommender systems do not consider the context of the user when finding the items to recommend [5]. However, it can often be useful to consider the context that the user is in when giving recommendations. For some domains, various information like the day of a week and where the user is can have a significant impact on what the user would want.

What is a Context?

Before explaining context-aware recommending systems (CARS) any further, it is important to define what a context is. As G. Adomavicius and A. Tuzhilin [5] write, the term context is used in so many different disciplines, and the definition often vary across the different ones (computer science, linguistics, philosophy, psychology, and organizational sciences). They focus on the definitions in the fields that are most relevant to recommender systems, and we will just summarize them up here.

- In data mining, the context is sometimes defined as those events that characterize the life stages of a customer and can determine a change in his preferences, status, and value for a company [8]. This information

can be events like a new job, the birth of a child, marriage, divorce, or retirement.

- E-commerce Personalization systems sometimes look at the intent of a purchase made by a customer as contextual information [36]. For example if someone buy something as a gift, he does not necessary want recommendations based on that when he visit again buying something for himself.
- In ubiquitous and mobile context-aware systems, the context was initially defined as the location of the user, the identity of people near the user, the objects around, and the changes in these elements [48]. Date, season, temperature, physical and conceptual statuses of interest for a user, emotional statuses have been added later [12, 46, 19].
- In marketing and management research they have studied the different behavior and decision making a customer can adopt and prefer different products based on the context [9, 33]. C. K. Prahalad [41] defines context as “the precise physical location of a customer at any given time, the exact minute he needs the service, and the kind of technological mobile device over which that experience will be received”. He distinguishes among three dimensions: temporal (when to deliver customer experiences), spatial (where to deliver), and technological (how to deliver).

Modeling

A typical recommender system consists of users and items and is trying to learn a function to estimate the rating for a user on a given item.

$$\text{Rating} = \text{User} \times \text{Item} \quad (2.5)$$

When we want to take the context of the user into consideration as well, we end up with the following function instead:

$$\text{Rating} = \text{User} \times \text{Item} \times \text{Context} \quad (2.6)$$

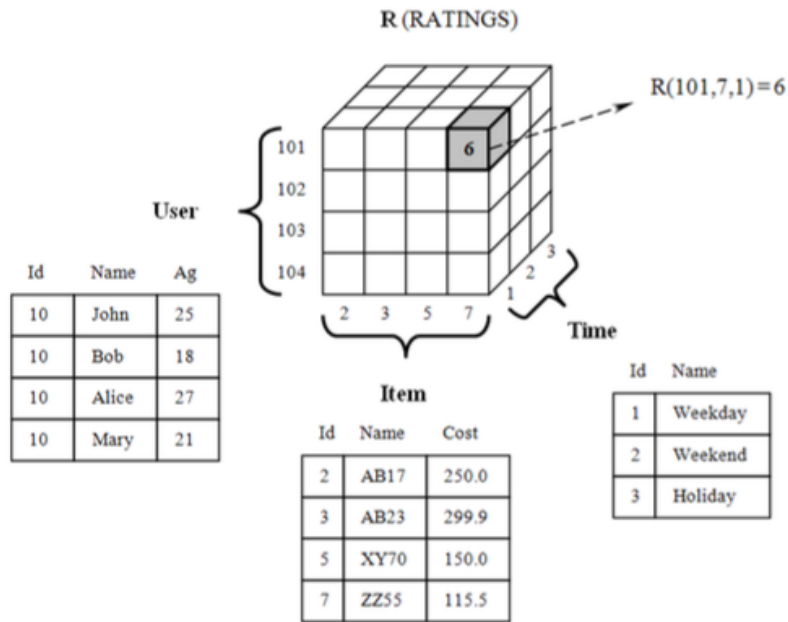


Figure 2.2: Multidimensional model for the User \times Item \times Time recommendation space [5]

If we use the example by G. Adomavicius and A. Tuzhilin [5] where they consider time as a context, we can model it in three dimensions as shown in Figure 2.2.

G. Adomavicius and A. Tuzhilin [5] describes two different approaches to use contextual information in a recommendation process; (1) recommendation via context-driven querying and search, and (2) recommendation via contextual preference elicitation and estimation. Mobile and tourist systems [2, 15, 49] use the first approach, by using the contextual information when querying for some information, like querying for restaurants in the restaurants that are nearby.

The latter approach, recommendation via contextual preference elicitation and estimation, represents a more recent trend for context-aware recommender systems [3, 35, 37, 51]. This approach tries to model and learn the user's preferences by obtaining information while the user is using the application, and by watching how he interact with previously given recommendations. G. Adomavicius and A. Tuzhilin [5] describes three ways that

this recommendation process can be implemented:

- Contextual pre-filtering are done by selecting data filtered on the context, and then it can do a traditional 2D recommendation process on the retrieved dataset.
- Contextual post-filtering is the opposite where the recommendations are first predicted without considering the context, and filter on the context afterward.
- Contextual modeling is a technique where the contextual information is used directly within the recommendation function. Instead of using a traditional 2D approach either before or after as the first two approaches, this will use a multidimensional model. Different algorithms have been developed, but we will not go into further detail about that here.

As we have talked about in previous sections, combining different recommendation techniques, it is possible to do the same within CARS, by combining multiple of the strategies described above.

Obtaining the Context

G. Adomavicius and A. Tuzhilin [5] describes three ways to obtain contextual information:

- By explicitly asking the user questions that help to define his current context.
- Implicitly gather it from the data or environment, e.g. using the GPS on a mobile phone to determine the location of the user.
- Inferring the context using statistical or data mining methods. For example, we might be able to determine who is watching the TV in a family of dad, mother, and son, based on what is being watched right now.

Other than knowing how to gather the information, we also need to decide which information to collect. G. Adomavicius et al. [3] propose that a wide range of attributes can be used, which are selected by domain experts that they believe can be useful information. After the data is collected, various statistical tests can find the most important attributes.

Issues

G. Adomavicius and A. Tuzhilin [5] also describes some issues with the current state of the research in context-aware recommender systems. More research need to be done on how to combine the different approaches and the trade-offs for each of them. Adding the context to recommender systems also increases the complexity, and so far most of the work has been conceptual. To be able to use this in real world applications with tons of data, more research about problems like efficient data structures and storage methods need to be done.

It is also a problem in the cases where the context is obtained by asking the user. These questions will require more effort from the user as well, and most recommender systems want to retrieve as much relevant information as possible without putting more stress on the user [14].

2.2 Feedback

Recommender systems are depending on feedback from the users, to build up knowledge and be able to have more accurate recommendations in the future. Feedback can be given in two forms, explicit and implicit feedback. In this section, we will cover the differences between those methods of gathering feedback from the users.

2.2.1 Explicit Feedback

Explicit feedback is an active action by the user on the system. This kind of feedback heavily depends on the willingness of the user to give feedback. Explicit ratings are often preferred over implicit feedback, because of the accuracy this provides. D. Cosley et al. [16] found that explicit ratings can

Explicit feedback	Implicit feedback
Rating (E.g. stars)	Time tracking
Reviews	GPS positions when a choice is made
Like or dislike buttons	Keyboard/mouse inputs
Surveys	Clicks
	Eye tracking (lab)
	Microphone input (lab)
	Facial expression (lab)

Table 2.2: Common types of explicit and implicit feedback

be of different qualities. Users may rate movies less carefully than they rate bigger choices, e.g. a review after a holiday trip.

In a study by X. Amatriain et al. [7], they found that extreme explicit ratings often are more consistent than mild opinions. The consequence is that we cannot get a better resolution of the user preference model just by increasing the number of stars it is possible to rate. The vast majority are using the extreme values. Another finding in their research was that similar items grouped together gave more consistent ratings and that fast ratings do not yield more inconsistencies. This strategy opens up for gathering a good amount of data about the user in a short period to acquire a basic knowledge quickly.

Explicit ratings required an action from the user. This interruption of the user flow can be critical in a negative fashion for the user to come back on a later occasion. It is hard to ensure that users leave their explicit feedback [40]. The laziness of the average user results in a relatively low ratio of ratings per user.

2.2.2 Implicit Feedback

Implicit feedback takes advantage of the users usage pattern. This technique is in many ways the opposite of explicit feedback where the user does not take an explicit action to give the system more information about themselves. Implicit feedback has the benefit of that it is easier to collect than explicit feedback because of the user willingness. In Table 2.2, examples of different implicit feedback methods are listed.

Y. Hu et al. [26] found that implicit feedback have four prime characteristics. The first one is that implicit feedback gives no negative feedback. It is hard to say something about items the user does not have seen. The user may not know about the item or dislike it, and it reveals a fundamental asymmetry between explicit and implicit feedback.

The second characteristic is that it contains a vast sum of noise. Why is the user browsing right now and why is he browsing the item at all can only be guessed with this type of feedback. For example viewing a shirt in a store does not explain that the user likes this particular shirt. However, if he continues to look at it multiple times, it is very likely that we can assume an interest in the product.

The third discusses the difference between numerical values of explicit and implicit feedback. Numerical values of implicit feedback can, for example, describe how often a customer watches a TV series. An explicit feedback tells more about how well the user relates to the item, e.g. like or dislike buttons.

The last characteristic is the evaluation of implicit feedback. The challenges come with how to evaluate items purchased, viewed or rated more than once. Y. Hu et al. [26] comes with an example where it can be hard to compare two TV shows broadcasted at the same time, where the user cannot watch both shows.

Some implicit measures do not bring any knowledge about the user at all, and some are only useful in combination with others [27]. Implicit feedback can as mentioned above be noisy, but with enough feedback, an appropriate model can be made to help the other parts of different systems be better. The more data about the user, the better, because the user patterns often reveal something (either in one or the other direction) about the end user. Implicit feedback can also be used to construct an explicit type of feedback. For example, if one user is only browsing novels in a book selling service, then after enough data, the system can assume the user does not have any other preferences at all. This action is the same as picking a default search category (explicit feedback).

2.3 The Cold-Start Problem

Well known for all types of recommender systems, is the *cold-start problem*. Some authors define the problem as when a new item is added to the dataset, and it is no ratings on that item. Another problem when a new user signs up that the system do not have any knowledge about is often called either *cold-start user problem* or *new-user problem*. In this thesis we will describe these issues as *cold-start item problem* and *cold-start user problem* [47]. When both of these problems occur at the same time, e.g. a totally fresh system, S. Park et al. [38] defines it as the *cold-start system* problem. In this section, we describe how previous research have approached these problems.

2.3.1 Cold-Start User Problem

Since the recommendations are supposed to be targeted for individuals, it is impossible to know how to compose good recommendations for a new user the system know nothing about. By giving bad recommendations to begin with, the user might be scared away. He could get disappointed by the mistakes of the recommender system, so he will quit using the software before enough knowledge about the user is built up. A crucial feature of most recommender systems is therefore to extract enough knowledge about the new user to be accurate enough. Many different strategies to solve this problem has been tried out, and we will describe some of the most common ways to do this.

Demographics

The user can either enter some information about the user, or retrieve it from social medias, which can help grouping the user into similar groups. Depending on what kind of system, some groups of people might share the same interests, so grouping people by attributes like age, sex, occupation, nationality, city, education, income, marital status, might help. Asking the user to enter all this information is however a little time consuming for the user, and might scare him away. By looking at the domain of the recommender system you want only to collect the data that will be most valuable to give recommendations. More and more applications also allow people to sign up using their social network profile, and this could help retrieving this data without having the user entering it manually.

Y. Wang et al. [50] implemented a demographic recommender system on tourist attractions on TripAdvisor. Using a crawler to collect ratings and demographic data from TripAdvisor, they had information including “Age”, “Gender”, “Travel style”, “Travel for”, “Great vacation” and “Travel with”. To classify the ratings, they used traditional machine learning approaches, Naive Bayes, Support Vector Machine (SVM) and Bayesian Network. SVM performed slightly better than Naive Bays, and both outperformed Bayesian Network. All three methods performed better than the baseline, and can be useful for recommendations. However, they were inaccurate on the rating values 1-4, and only accurate in identifying the users that have given 5 (attractions were rated 1 - 5). This outcome was probably because the dataset was very unbalanced, where most people only rated 4 or 5. The authors suggest for future work to implement a hybrid recommender that can incorporate more information about the attractions and reviews.

While demographic data can help a lot in some recommender systems, it is also less general than other techniques since it depends on domain knowledge and only apply to certain domains as A. M. Rashid et al. [42] mentions in their paper using another approach to the cold-start problem.

Asking the Users to Rate

As A. M. Rashid et al. [42] says, the most direct way of acquiring information about new users, is to ask them explicitly to rate items. However, since it is bad to put more stress on the user, it is important to not ask the new users more questions than necessary. Therefore, it becomes very crucial that these questions will distinguish the users, and give as much information as possible. As mentioned in their paper, it does not help to ask questions which all users agree on, e.g. a food recommender should not ask their users “Do you like vanilla ice cream?”, since most people will say yes to this question. It is also important to ask questions that the user is likely to have an opinion about. The goal must be to extract general knowledge, and too specific questions will only make it hard for some users to know what they should answer, and it gives little knowledge in return. A. M. Rashid et al. explore approaches for finding the best items to present to new users for rating by using the MovieLens movie recommender. They identify four dimensions that a system can be judged on; (1) User effort: How hard was it to sign up? (2) User Satisfaction: how well did the user like the sign up process? (3)

Recommendation accuracy: how well can the system make recommendations to the user? (4) System utility: how well will the system be able to serve all users, given what it learns from this one? A. M. Rashid et al. choose to focus on effort and accuracy, since those two are easy to measure.

They are discussing and testing five different strategies for presenting items to rate to the user, and we will summarize them up here.

- **Random** strategies select random items to present to a user. Another approach is to select one random item from a popular list, and the rest at random of all items. The advantage is that all the items can collect information, but it is also likely that the user will not have an opinion on that item.
- **Popularity** is a strategy where the items are ranked in popularity, and they are presented to the user in descending order. This makes sure that the user is likely to have an opinion about it, but it is also a big risk of presenting items that almost all users like, and hence will not be able to learn anything useful.
- **Pure entropy** presents the items that the users usually either dislike or like very much. An item with most ratings being either 1 or 5 will have high entropy. This method can also suffer from the problem that the user might not have an opinion about the items.
- A **balanced** strategy combine the popularity and entropy methods. This helps on the problems with using one of the methods, since the user is more likely to have an opinion about the item, and the items will also be the ones that you can learn something about the user from.
- An **item-item** personalized approach are at first presenting the items using one of the above methods. Other items from the users that have previously rated an item that the user rates, are retrieved, and presented to the user.

In Table 2.3 is the results from their experiment with the different strategies. They did both an offline experiment with the dataset from MovieLens¹, and an online experiment with new users. Item-item suffers from that the items

¹MovieLens dataset: <http://grouplens.org/datasets/movielens/>

Strategy	User Effort	Accuracy
Random	★	★★
Popularity	★★★★★	★★★
(log) Pop*Ent	★★★	★★★★★
Item-Item	★★★★★	★★

Table 2.3: Evaluation of different strategies from A. M. Rashid et al.

presented to rate for the users will often be clustered together, and it does not gain knowledge about all the items. The user effort is evaluated by how many users dropped out during sign up, and accuracy is the mean absolute error (MAE). MAE is here the sum of the absolute differences between each prediction and corresponding rating divided by the number of ratings.

Which strategy to use in a recommender system depends a lot on the domain. A system that works within e-commerce, should probably suggest the most popular items to begin with when there is no data about the user, and then it could move on to use the item-item strategy. In other systems, it might work better to build up better knowledge about the user using the balanced strategy.

Tags

Collaborative filtering typically uses a matrix $\text{User} \times \text{Item}$, which will be hard to fill out for a new user. If the items are grouped into tags, a matrix $\text{User} \times \text{Tag}$, will be denser, since for every tag you know the users opinion about, you will be able to translate that information towards multiple items. This approach was developed by H. Kim et al. [28], and they tested it on a dataset of bookmarks collected from del.icio.us, a site containing bookmarks tagged by the users. They and achieved better results than regular collaborative filtering on the items when measured with recall.

Their approach to the problem was first to build a model of the candidate tags for a user, using collaborative filtering. Using that model of the tags, they generate the top-N recommendations using Naive Bayes.

H. Kim et al. [28] mention an issue with their method that is noisy tags makes the performance worse. This issue could be a problem for some real

world applications where the users create the tags.

User Select Some Users He Trusts

Another approach that has been used is making the user select one or multiple users that he trusts, and then give recommendations based on what these users like [34]. This technique gives the recommender engine an initial input about the new user that can be valuable in recommending items in the beginning.

2.3.2 Cold-Start Item Problem

When no ratings exist on an item, it is not possible to get a predicted rating from a traditional collaborative filtering approach [47]. A. I. Schein et al. [47] presents a solution to this problem by using a probabilistic approach that combines content and collaborative information by using expectation maximization (EM) [18] learning to fit the model to the data. They test their approach on a movie dataset, and they look at the cast of actors in a movie, and how similar it is to what the user have rated before.

In a content-based filtering system, the cold-start item problem can be easier to work around compared to the collaborative filtering approach. With content-based filtering, the content of the items can be matched against a user profile, even if the item have not been seen by any other user before.

The *ask to rate* strategy described earlier can also help the user discover new items if the items are selected randomly from all the items.

2.4 Case-Based Reasoning

CBR is a way to solve new problems based on previously known solutions to similar problems. It can be done by adapting old solutions to new problems, and be used for problems like explaining new situations, and avoid mistakes that have been done before [45]. It has a lot in common with the way humans solve problems already and is therefore also connected to cognitive science. In this section, we first describe the CBR approach in general, and then show how it has been used in a recommender system.

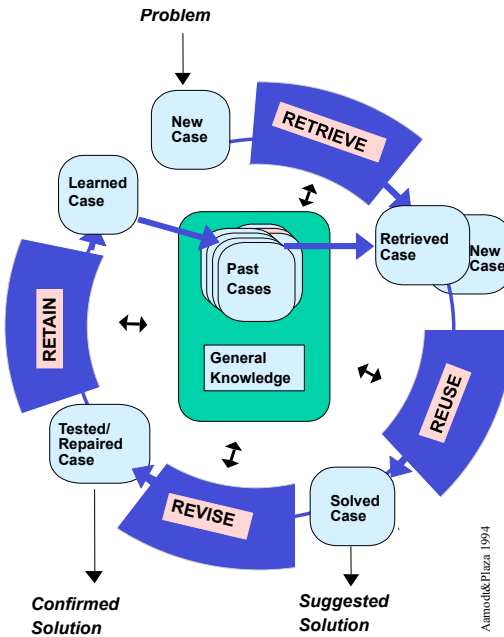


Figure 2.3: The CBR cycle

2.4.1 The CBR Principles

For understanding CBR, it is essential to know what a case is. A case represents specific knowledge. J. Kolodner [29] defines it as: “A case is a contextualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner.”

The process in a CBR application can be described as 4 steps [1, 31] as shown in Figure 2.3 The *retrieve* step is looking up similar cases in the case database. Then one or multiple old cases are *reused* to make a solution to the new target. *Revise* will test the solution and change it if necessary. When the new solution is successful, it will be *retained* in the case database.

$$d(p, q) = d(q, p) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2} \quad (2.7)$$

Case-based reasoning uses a similarity measure to retrieve similar cases. Figure 2.4 shows a new case and its mapping to the previously learned cases.

A computed mathematical distance measure the distance between two cases. The different parameters of each case are normalized/converted to a numeric value, then used in this similarity measure. Different similarity measures are used, but the Euclidean Distance, shown in Equation 2.7, has been very popular and given good results [6]. The distances between cases in the knowledge-base is by itself not very useful, but can be used to retrieve the nearest neighbors. The K-nearest neighbors (KNN) [17], is an algorithm that takes the k as a parameter saying how many neighbor nodes to return. KNN can use this similarity measures to compare the node we now are comparing to all the others. It is shown that nearest neighbor algorithms can struggle with high dimensions where the query case is not part of a cluster [10]. The KNN is a lazy learning algorithm.

Listing 2.1: The KNN algorithm

For each training example $\langle x, f(x) \rangle$:

 Add the example to the list of training_examples.

Given a query instance x to be classified,

 Let $x_1, x_2 \dots x_k$ denote the k instances from training_examples
 that are nearest to x .

 Return the class that represents the maximum of the k instances.

In a standard KNN, the nodes takes the class of the neighbors (depending on what class there is most of). In a case-based approach, the KNN is used, as mentioned above, to retrieve cases similar to itself. This retrieval is done because of the next step, reuse, where solutions from previous cases are mapped to the current case. Often this is not a one-to-one match, so modifications to previous solutions may be necessary. For example, one can use different parts of each neighbor to come up with a new tailored solution. A suggested solution for the new case we are currently working on is the output of the reuse step.

After a solution proposed from the previous step, based on old cases and tailored to the target case, a testing/simulation of the new solution takes place. Testing is an essential step in knowing the impact on the knowledge-base. Simulation of how well the solution performs in the environment reveals weaknesses in the solution for this particular case, and can be repaired in this step. The goal of this step is to come up with a repaired and confirmed solution what performs well.

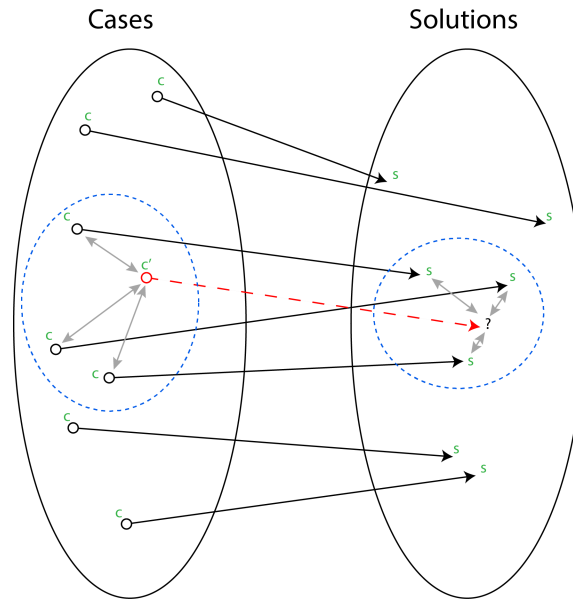


Figure 2.4: The mapping from problem to solution. The dotted arrow shows a new case and its solution adapted from the previously solved cases.

The last step of the CBR-cycle is the retain step. This final step is to preserve old successful cases with good solutions. This knowledge goes back to the knowledge-base and will be one of the cases considered when a new case is assessed with the KNN algorithm.

2.4.2 CBR as a Recommender System

A generic CBR system can be seen as very similar to a recommendation process, hence also used as a recommender system. F. Lorenzi and F. Ricci [32] covers four recommender systems taking advantage of CBR principles and six recommendation techniques (covered below):

- Interest Confidence Value: The assumption of this technique is that the user's interest in new items is similar to the user's interest in past items.
- Single Item Recommendation: Simply a recommendation of one single item.

- Seeking for Inspiration: Prompts the user with explicit feedback to update the recommendations, hence the inspiration for the system.
- Travel completion: This technique offers items that “completes” the current selected/purchased items, e.g. accessories.
- Order-based retrieval: Like in the name, this technique emphasizes the ordering of the items retrieved from the case base.
- Comparison-based Retrieval: Users preferences into query parameters to receive recommended items, e.g. cold-start preferences.

2.5 Examples of Earlier Projects

In this section, we will cover some examples of projects both in research and the industry, which we consider to be relevant for our recommender system that will operate in the food recipe domain. These examples shows the diversity of recommendation methods operating in different domains, and some projects related to food. We picked these examples since they have inspired us in the design and development of Eatelligent.

2.5.1 Amazon.com

One of the lead industry companies in recommender systems is Amazon.com. Amazon.com is one of the largest online retailer in the world ², started out selling books. Amazon offers recommendations to their users to help them select their purchases based on many of the techniques mentioned above. In a real-world problem domain, we often enter a new issues not emphasized in research with fictive users and user responses, with respect to the original problem domain. Amazon is mainly using collaborative filtering and cluster models [30]. In Amazons algorithm recommending new items it is important to find items that are similar to the items that are already purchased by the user, hence item-to-item collaborative filtering and clustering items in categorizing clusters. Real-time recommendations with collaborative filtering is a

²The 5 Largest Online Retailers in the World:
<http://www.insidermonkey.com/blog/the-5-largest-online-retailers-in-the-world-331292/>

challenge due to its complexity and the vast amount of items and users. One of the solutions to this problem has been for Amazon to introduce cluster models. Cluster models are in many ways a similar way to tag items in the same category to have some knowledge about the relation between items. A major upside of this approach due to the expectations of the user to have the result within half a second is that it can be run offline in advance. Rather than combining each of the users to other similar users, an item-to-item collaborative filtering matches the users previous scored items to similar items, then merges those related items into a recommendation list. The intriguing aspect of this system is the retrieval of similar items (based on the tags for each item), in respect of previous choices made by the user. This method scopes down the search space both for items and users, and is very interesting due to scalability and real-time recommendations.

2.5.2 DieToRecs

DieToRecs is a case-based travel planning recommender system [44]. This system is used to make a travel plan for holiday trips to a selected destination. That includes problems like which hotels to stay at and which beaches to visit. DieToRecs implemented three different recommendation techniques: Single item recommendation, travel completion and seeking for inspiration. The first door the user may use is the single item recommendation. It first asks the user for a wide set of general questions and gradually tightens the scope until a single recommendation remains or else a tightening function is used. This technique is typically used to make the major choices for a trip (at least for a machine that can prune possibilities), say finding a destination. The second technique is travel completion, where the system offers items or services that complete the trip. The system retrieves similar cases (travel plans previously built by other travelers). Collaborative features for each case is used to generate logical constraints to the travel completion step. Seeking for inspiration is the third step in DieToRecs. This step involves prompting the user with complete recommendations, hence giving inspiration to the system. Each iteration in a loop the user are given six choices of complete travel recommendations. These six cases is retrieved from a randomly selected case and retrieved by using the K-nearest neighbors to the initial case. The user selects one of the six to move gradually towards the travel recommendation the user wants.

This type of recommender systems could be used to solve problems like planning of a weeks menu of dinner recipes. DieToRecs prompts the user for feedback during the recommendation process, and this can be utilized in the food recommendation domain to enhance recommendations by do some last minute changes to the final recommendations.

2.5.3 CHEF

CHEF [23] is a case-based planner, which domain is recipe creation. It takes in an input consisting of goals that recipes can achieve and outputs a recipe that can reach those goals. Recipe creation is outside the scope of this thesis, but it is interesting to see the knowledge CHEF have about the role of the ingredients. It knows that chicken is meat, and if some goals match an old recipe in everything except it is chicken instead of beef in the new goal, CHEF can understand that chicken can replace the beef.

When CHEF is finished with making a new plan, it will use feedback to know if it worked out or not. If not, it will try to learn from the situation, and make an explanation of why, which will be indexed in the general knowledge, and then can use for repairing the faulty plan.

While CHEF uses the knowledge to make recipes that satisfy the goals and to modify recipes, we will not necessarily need to modify the recipes to make a good recommendations. With enough recipes in the database, it should be possible to find good enough matches, but it can be done to learn more about the ingredients for individual users. If a user does not like salmon, the system should be able to learn that it is salmon the user do not like, and not just the recipes he has tried with salmon in it.

2.5.4 JULIA

JULIA [24] is a case-based designer that works with meal planning. The problems are described in terms of constraints that needs to be achieved, and the solution describe a model that fills as many constraints as possible.

As JULIA is a designer, it is more suitable for complex situations where a recipe needs to be reconstructed to fit constraints. JULIA also has much

knowledge about the ingredients, and it tries to not repeat the main ingredients when creating a plan of several dishes. JULIA's focus is here on creating a starter, main dish, and dessert, but we are focusing on recommending a main dish. However, the same approach as JULIA can be used to avoid recommending multiple recipes with the same main ingredient in a row. Many people will prefer to have some variety in their diet, e.g. not eat salmon every day.

2.5.5 Intelligent Food Planning: Personalized Recipe Recommendation

Personalized Recipe Recommendation [21] is a recipe recommender system where the recipe is split up into its ingredients and the actual recipe. The weighting for the recommendation is based upon not just the recipe, but also the ingredients used to make the dish.

This system is using collaborative filtering to make the calculations for each user and discusses issues around the cold-start problem that collaborative filtering suffers from. The system shows some statistics of results from the project with 183 users and 136 recipes, and 337 classified food items (ingredients).

This system is connected to this thesis in the way it defines the items (recipes and ingredients), where it sees the recipe as a set of ingredients. Across the related recipes, these ingredients are recycled. On a normalized level where the ingredients are comparable across recipes, the learning can then be done. Section 3.4 describes how this idea is implemented in this project along with the tag abstraction.

2.5.6 Food Recommendation Using Ontology and Heuristics

This paper describes a system using term frequency-inverse document frequency (TF-IDF) to do the food recommendation for users. Collaborative filtering is as mentioned in [21] suffering from the cold-start problem, and that is solved by looking on previous meals by each user.

Nutrition is an important factor in this system, where the system gather all nutrition information for each ingredient from United States Department of Agriculture (USDA). Nutrition is used to make a more healthy recommendation.

This system uses a query language to do the requests (the further plan is to define the grammar to standardize the query language.) This language is a content- and semantics-based approach where the attributes like ingredients and description has a significant role. The results presented in the paper describes a rather good accuracy for TF-IDF.

2.6 Relating the Theory Towards Eatelligent

In this section, we will discuss the theory and background described earlier, and look into what will be implemented in our recommender system. Some of the strategies are general and can be used in any recommender system, and some are more specified to the domain of recommending food recipes.

2.6.1 Recommender System

The implemented recommender system in this thesis is a hybrid recommender where some of the topics covered in this chapter are used. The mixed approach to the hybrid recommender is used, because it is easy to compare the different methods. The implementation is focused on the study of the cold-start problem in connection with a real-world example. The system starts out with zero users and zero ratings. Each of the items is never seen by any user (cold-start system). This matter excludes the model-based collaborative filtering method because it is nothing to build a learning model from. A memory-based approach is chosen because of the online learning part that is developed during the user data gathering.

For the CBR part of this hybrid recommender it mainly looks at the nearest neighbors in connection to tags for both items and users, covered in Section 3.4.1.

2.6.2 Cold-Start

The cold-start problem can be helped in many different ways, and have focused on trying out a few strategies that we believe can assist in recommending food recipes to a new user.

Demographics

Some demographic facts about the user can be relevant for grouping people together that might have a similar taste. Attributes like age, location, sex, income and occupation could all be helpful information to know what type of food the user likes. However, we also have to take into consideration what kind of information the user will be willing to enter into a mobile application. First of all everything the user has to enter at sign up will add more stress and make it more likely that the user will quit using the application even before he has signed up successfully. Secondly some information might sound too sensitive to most users, e.g. people will most likely not want to enter their income in an application for recipes.

Location, or at least city or nationality, can be valuable information for a global application since people around the world can have different food traditions, and different groceries available. We will let the user enter the city on their profile, but since this application will be implemented for Norwegian users to begin with, we will not use this information for now. Age can also be entered into their profile, but this is also something that we do not think is the most efficient way to recommend recipes to a new user. Our implementation focuses therefore on other methods.

Ask to Rate and Tagging of Items

To extract some knowledge explicitly from the user in a few questions, we have designed a strategy with the knowledge that it is food recipes that we are going to recommend. Instead of just asking the user to rate recipes, we ask the user some yes/no questions that we extract into knowledge about both recipes and ingredient. This knowledge is extracted by grouping the ingredients and recipes into tags, and each answer will make a score for the user on a tag. This strategy will be described more in detail in Chapter 3.

The user answers to these questions can help in two ways: (1) Knowing a score for the user on different tags, makes it possible to aggregate a score on the recipes, and the best score are the recipe that fits the user's model the best. (2) The answers can also be used for finding a similar user, by looking at the distance between two answers, and finding the recipes that the most similar user likes the most.

The tagging model also work after the initial questions have been answered. Every time a recommendation is given to a user, he can select yes or no, and the score on the tags will be updated. A user provides therefore information to the system by just viewing different recipes.

The tags also help the system to know more about the ingredients. Like CHEF, it knows that chicken is meat or that pepper is a spice. This knowledge can help identifying which tags are more important in a dish. It is likely that a user who dislikes some of the greens in a dish are willing just to skip or replace them and still make the dish. However, a user who does not like salmon will probably not want a recipe containing that.

Since the tags are supposed to be entered in the recipes themselves, and the ingredients are also tagged, this approach helps on the cold-start system problem. This tagging help since it will be able to score each recipe in the system regardless of if ratings from other users exist.

Trusted Users

Asking the user to select one or multiple users he trusts is something that could be helpful, since a user often know some people that have the similar food taste like him. This feature could also be implemented by using a social media so that the user can easily select friends from a list. Some might be following somebody that blogs about food, and also know that they trust their taste. However this feature needs many users to work, and since our system will suffer from having very few users to begin with, we will not focus on implementing this strategy.

2.6.3 Feedback

The implementation in this thesis covers a mobile application. As mentioned in Section 2.2, the gathering of explicit ratings from a user can be challenging,

so we used the benefits of smartphones to report implicit ratings to the server. The final mobile application uses tracking on every screen and how long the user is on the page. This is implicit time tracking which can be used to say whether or not the user has made the recipe at a later point in time. Each time the user asks for a recommendation, it is logged to the server. This information is then used to evaluate the recommendations for this particular user.

As mentioned in Table 2.2, some of the implicit feedback methods is not suitable for a real-world problem domain, because of privacy. Eye tracking, microphone input, and facial expression are therefore excluded from this experiment. For a pure recommender system, it is also limited how much this type of feedback will bring to the table. Locations for a user, when a recommendation is made, is related to context, and in some conditions group recommendations. GPS positions are chosen to not be implemented because this is falling outside of the research questions.

Primary explicit feedback methods for the implemented system is as follows: Star ratings for the explicit ratings of the different items. Like and dislike points for the cold-start methodology, e.g. yes or no to chicken. For the implicit feedback, we choose to emphasize time tracking and clicks to learn more about the usage pattern for each user.

2.6.4 Taking Advantage of the Contextual Information

We have considered some contextual information that could have an impact on what the user would want to make for dinner.

- How much time the user has available is can be very relevant. It is not preferable to recommend a recipe that takes one hour to make if the user has a busy day and only 20 minutes available. However, it can put too much stress on the user if we ask him explicitly every time he makes a request for new recommendations about how much time he has available. Therefore, we have concluded that this will be a bad approach. A better strategy would be to extract this information implicitly, e.g. by learning the days the user want something fast, and when he wants something fancier.

- What groceries the user already has available at home, can also be considered as contextual information. Some days the user would not want to have get new groceries, and by giving the application some items, a recipe containing those would be a good recommendation. This problem will, however, be more similar to a recipe composition problem, than a recommendation problem, and will not be the focus of this thesis.
- The fact if the user is going to make food for only himself or others, could also affect what he wants to make. If all the people that were going to be served, also had this application and a user profile, the system could turn this into a group recommendation if the user selected all the people that he would serve. Again this makes the problem more complicated and also require many users to get it to work in the real world, so this will not be the focus here.
- Season, Holidays, weekends, could also be interesting to consider in the composition of recommendations. The season of the year could have an impact of which groceries are available, and people might have different traditions depending on warm and cold weather. On the weekends, people often have more time and put more effort into making good food. Some holidays include food traditions, and you might want the recommendations to represent this, and at the same time, we do not want to recommend Christmas food outside Christmas.

Using contextual information in a recommender system for recommending food items at restaurants is shown to improve the accuracy compared to a traditional collaborative filtering approach [25]. We have reason to believe that this could also improve the recommendation of food recipes. However it also makes the problem a lot more complicated, and in this thesis we will only discuss it for future work in Section 5.3. We also believe that it is more likely that the users who signs up with our application will be mainly ask for recommendations during the weekdays, since most people tend to have their own traditions for holidays and weekends, so the dataset will consist mostly of recipes that can be made on a regular day.

Chapter 3

Architecture/Model

This chapter will cover the design and implementation of Eatelligent, the dinner recipe recommender. As mentioned earlier, we found it necessary to develop our application to promote flexibility to what data we can use for the recommendations. This chapter, therefore, describes the full application in detail, and not only the recommender system, since both of them play a significant part in this thesis.

Eatelligent is released on the open market as a mobile phone application. The system consists of two main components. A client, the mobile application, and the server, which serves the client with data and stores user information, recipes, and feedback. The recommender engine is implemented on the server.

We will start by describing the functionality of the application, implementation details, and in the end of the chapter the details about the recommender engine.

3.1 Requirements for Design

This section lists the requirements for the system as user stories to give an overview of what the system should fulfill, and why we need each requirement.

- As a user, I can sign up and login so that the information about me can be used next time I use the application to give more accurate recommendations in the future.

- As a new user, I will be prompted with some “yes/no” questions to extract some basic knowledge about my food preferences so the system will have some knowledge before recommending recipes.
- As a user, I retrieve recommended recipes so I can easily explore new recipes and find out what to eat for dinner.
- As a user, I can indicate if I want a recipe by answering “yes” or “no”. If “yes”, have a detailed look at the recipe, if “no”, view the next recommendation.
- As a user, I can rate the recipes 1 to 5 stars, to give an indication of how much I liked a recipe, to build up the knowledge of which recipes I like.
- As a user, I can add recipes to favorites so that I can easily retrieve the best recipes.
- As a user, I can add a recipe to a shopping cart so that the ingredients for the recipe I choose to make, will be easy to go and get at the grocery store.
- As a user, I can search for and retrieve recipes knowing part of the title.
- As a user, I can look at the last viewed recipes, to easily look at the recipes I just made if I want to give them a rating or make them again.
- As a user, I can reset my knowledge base, in case I wish to start over and answer the cold-start questions again, and reset the model the system has developed on me.
- The application stores all the recommendations given to a user, so that the user will not get the same recommendations in a time interval, and then the system will also know where the user got the recipe from if he rates the recipe.
- The application stores how long a user looks at a recipe, to give an indication if the user did make this recipe or not, and the longer you look at a recipe, the more interested you might be.
- The application gives out some recommended recipes as random recipes, to compare the rest of the recommendations with a baseline.

- As an administrator, I can group ingredients and recipes into tags so that it can make a relation of what kind of tags the user likes or dislike.
- As an administrator, I want to add, modify, delete and publish new recipes into the application.

3.2 System Overview

Before we go into detail about the different parts of the system, we want to give an overview of the whole user process. We describe the process from a user sign up with the mobile application, and how the client interacts with the server and recommender system.

We store the information on the server when the user signs up, so that all the knowledge to the user can be used every time the user logs in. Since no knowledge about the user is available at this point, the user is supposed to answer some “yes/no” questions. These responses are stored to give an initial value of the preferences for the user. When a user receives recommendations, the server stores all of them with a field *source* which describes which algorithm promoted the recipe. This information makes it possible to always know where the user discovered the recipe when he or she rates it. Stored along with the information is the new rating, and for recommendations containing a predicted rating, we can evaluate the difference between the predicted and the actual rating. Figure 3.1 shows the flow between the clients through the backend to retrieve recommendations.

3.3 Implementation

In the following section, we describe the different parts of the system, discuss the various choices that we have done, and describing the implementation that we ended up using in the final product. The three main parts is the client, backend (server), and the recommender system.

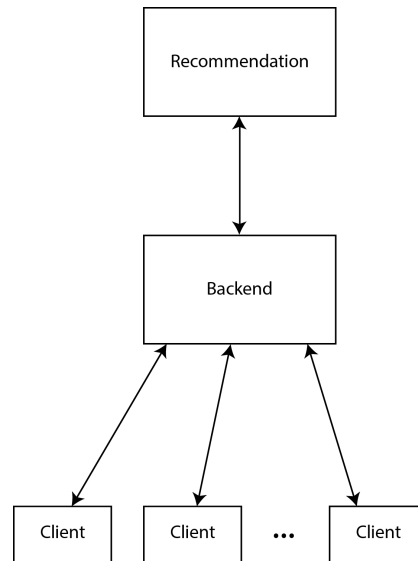


Figure 3.1: System overview

3.3.1 Client

Since more and more of the web activity happens through smartphones, we decided to address the end user with a smartphone application. The client for this project is an HTML5¹ application embedded in a web view and packed into an application running on all devices, available in the respective application stores for Android and iPhone. With limited resources and time to develop a client for both Android and iPhone, we decided to develop one code base that was able to deploy the application on the different mobile phones. This approach is possible by using HTML5 with the Cordova project², and it also makes it very easy to communicate with the servers Application Programming Interface (API).

The client is fully backed from the API (Section 3.3.2), with Hypertext Transfer Protocol³ (HTTP) requests to acquire all the content visible for the end user.

As described in the Figure 3.2, the classes are strictly distinguished between

¹HTML5: <http://www.w3.org/TR/html5/>

²Cordova: <https://cordova.apache.org/>

³HTTP: <https://tools.ietf.org/html/rfc2616>

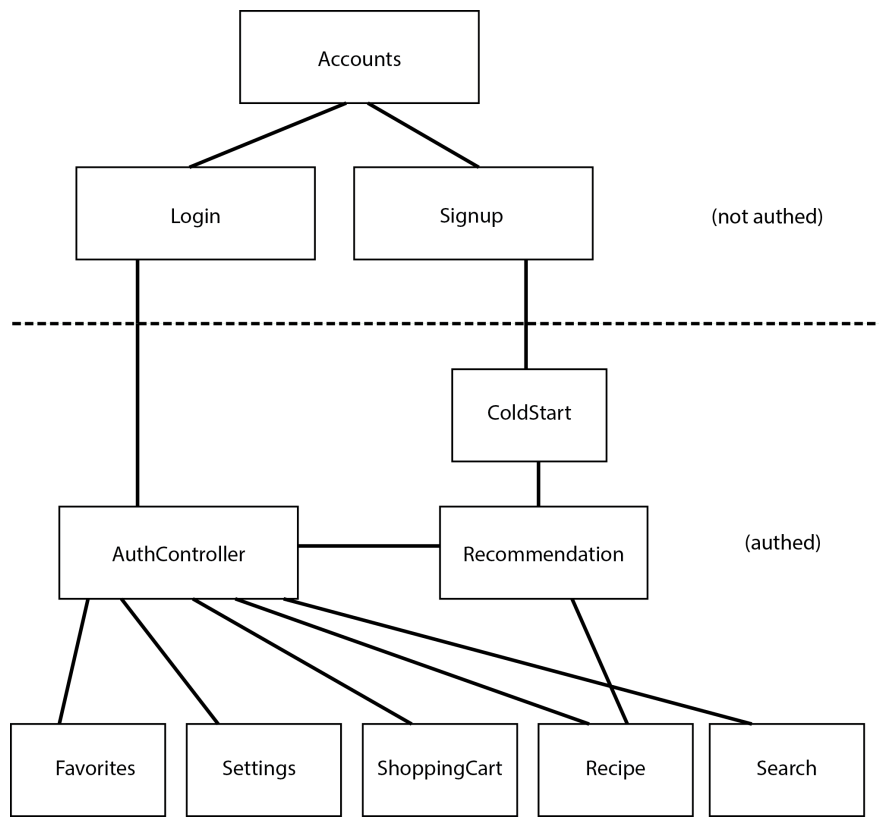


Figure 3.2: Class hierarchy for the mobile client

authorized and not authorized classes with abstract parent classes. Different functionalities are split up into modules. For example, Favorites and Settings are both standalone modules. This object-oriented approach was helping to keep track of more complex tasks.

We put some effort into user experience because the average smartphone user is used to use great application of different sorts. This client follows conventions for the modern smartphone application, to make the user feel like it is a good experience to use the application.

Recommendation Cycle

When the end user asks for recommendations, the server gives a list of recipes presented to the client. This list is ordered based on the recommendation for the requesting user. Starting from the top of the list, the user have to answer either “No thanks” or “Yes please”. If he selects “No thanks”, it is recorded to the API and taken into consideration for later recommendations. The next recipe in the list will be presented until the user pick one recipe (answers “Yes please”). When he select “yes”, it is also synchronized with the server and taken into consideration for later recommendations. Then we present a detailed view of the selected recipe for the user, so he gets all the information needed to make the dish.

After the user has come up with his opinion on the recipe, the user will hopefully rate the recipe from 1 to 5 stars. All this represents the cycle of how the user interacts with the recommender system.

Search

To open up the entry to view the other recipes, the client also provides a search for other recipes. When the user finds something and clicks it, it will be collected as a view of the recipe.

Favorites

The users of the client have the possibility to add recipes to a list of favorites. The favorites module is located in the sidebar along with the other parts of the application.

Shopping Cart

The shopping cart is an add-on feature just to make it easy for the user to buy the ingredients for one or multiple recipes.

Last Viewed Recipes

To view the history of recipes for one user, the client contains this module. This history can be useful for several reasons, for instance, some users may say “Yes please” to a recipe from the recommender, and afterward figure out that it was not a good time for this recipe after all (for several reasons). The user will not have to remember the name of the recipes because they are listed here.

Data Collection

To be able to study the results of this project, we had to collect every action done on the client. To collect these data, we used Google Analytics to track every user’s user pattern. There are many different tracking services that can provide this type of tracking, but since this is an HTML5 application, Google Analytics is a relatively easy integration to use instead of custom tailoring for each platform. Google Analytics is free and helped us collect our results presented in Chapter 4.

3.3.2 Backend

The backend (or server) of this system is where all the clients get their recipes, recommendations and post their feedback. In other words, this is where the clients get the content they display for the user, and the place they save their information. The server is implemented using the modern web architecture style Representational State Transfer (REST) [20]. It is done with the programming language Scala⁴ with Play Framework⁵, which gave us a modern language and framework, and at the same time it was possible to use any Java library if necessary. We hosted the server on the Heroku

⁴The Scala Programming Language: <http://www.scala-lang.org/>

⁵Play Framework: <https://www.playframework.com/>

cloud service⁶, which enabled easy deployment without too much configuring on our own.

The server stores all the data, and the recommendation engine is also implemented there, which is essential since we need the data for all the users to be able to make recommendations for one user. All the different models and the structure of the data we store in the database have been designed for being able to recommend recipes, and evaluate the system in the best way.

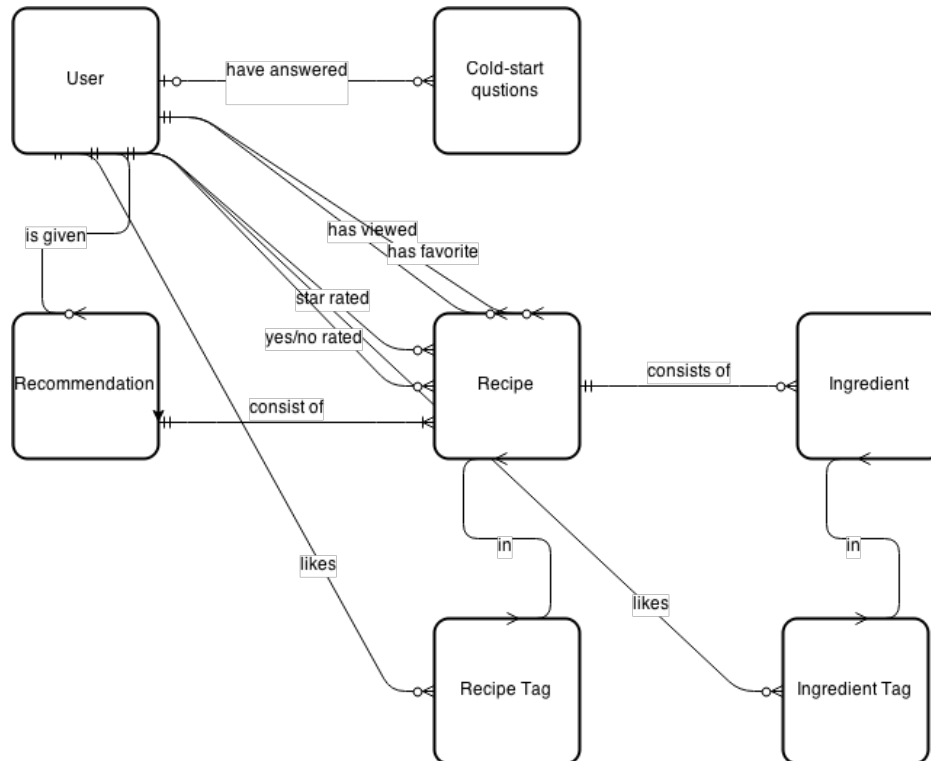


Figure 3.3: Entity relation diagram

Model

Here is some information about every different object, which helps to understand the rest of this chapter. The relations between the entities are shown in Figure 3.3, and Appendix A shows the database scheme in detail.

⁶<https://www.heroku.com/>

- **Recipe:** Containing all the information needed to show a recipe to the user.
- **Ingredient:** The recipes have links to the ingredient so that two recipes with “tomatoes” in it will link to the same ingredient. This is essential to be able to learn which ingredients an user likes or do not like.
- **Recipe Tag:** Tags on recipes to group them with similar recipes. One recipe can have many tags. The tags are made to group similar recipes together and help to know what categories a user typically likes the best. It is not shown to the user, and are only here to give information to the recommender system about the recipe.
- **Ingredient Tag:** Same as Recipe Tag, but just for ingredients. These are not supposed to be shown to the user either, and are only here to make knowledge about ingredients. These tags are manually set by ourselves, and makes it easy to know if an ingredient is from an animal. It also makes it possible to know if the ingredient is a main ingredient or something not that important in a dish.
- **Unit:** The units that are used to describe the amount of an ingredient in a recipe.
- **Star Rating:** A user’s rating of a recipe. The rating is an integer between 1 and 5. Along with the rating the time stamp when it is rated and the source (where the recipe was discovered, e.g. collaborative filtering, search, etc.) with additional data are stored for evaluation.
- **“Yes/No” Rating:** When a user gets recommendations, he can either say “yes” and look up the recipe or “no” to get the next. We store this information, in order to not show a recipe that the user already have said no to multiple times. This information can be aggregated, e.g. we know if an user have said no 3 times.
- **User Viewed Recipe:** To have an indication if the user made the recipe or not, other than the rating, we store how long time the user looked at a recipe.
- **Cold-start Questions:** These are “yes/no” questions that the user can answer to give us more knowledge. The answer and the time are stored.

- Given Recommendation: All the recommended recipes given to a user are stored to be used for evaluation.

Status	Code	Description
OK	200	Everything went OK
Bad request	400	The request was not understood by the server
Unauthorized	401	The user requesting is not authenticated
Forbidden	403	The user is not allowed to retrieve the resource requested
Not found	404	The server could not find the resource that was requested
Conflict	409	Used for duplicates that are not allowed
Internal server error	500	Unexpected server error

Table 3.1: The different response codes used by the server

Requests

The client uses the server by making requests on different URLs which is called endpoints. Appendix B lists the documentation for each endpoint. Different HTTP methods⁷ are used for different types of requests. GET requests are used when the client only asks for data, and nothing will be changed on the server. Requests for storing data, e.g. a rating on a recipe, will be done using POST requests. If something is supposed to be updated it is done using PUT. The response from the server contains a response code⁸ that tells the client if the request succeeded, or if something went wrong. In Table 3.1 the different codes used by our server are listed and explained briefly.

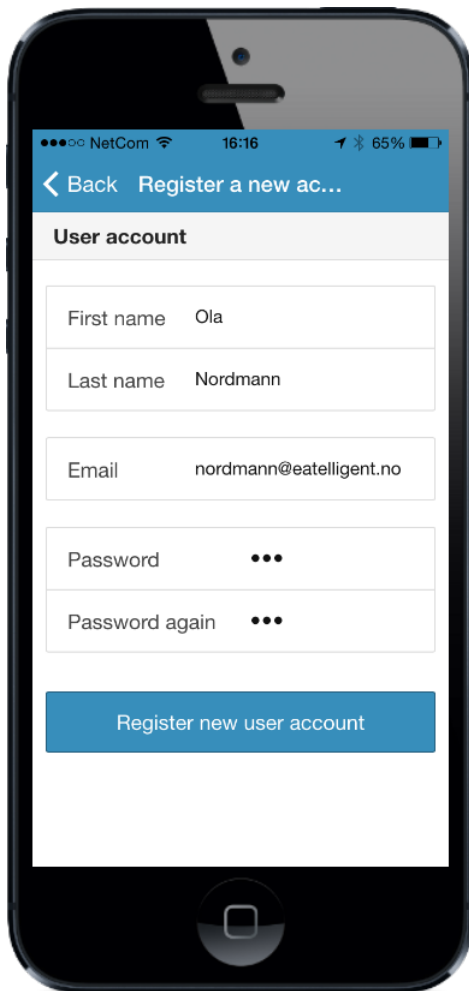
All the data passed to or from the server is in a format called JSON⁹. In

⁷HTTP Method Definitions: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>

⁸HTTP Status Code Definitions: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

⁹The JavaScript Object Notation standard: <http://www.json.org/>

Figure 3.4 - 3.8, we show the screenshots of the app, requests from client, and response from the server.



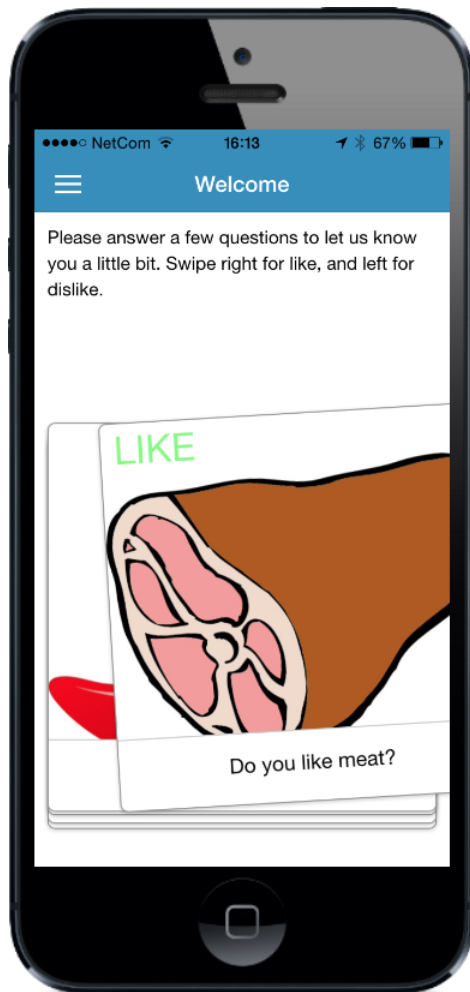
Listing 3.1: Sign up request

```
{  
  "firstName": "Ola",  
  "lastName": "Nordmann",  
  "email": "nordmann@eatelligent.no",  
  "password": "ola"  
}
```

Listing 3.2: Sign up response

```
{  
  "ok": true,  
  "message": {  
    "id": 14,  
    "loginInfo": {  
      "providerID": "credentials",  
      "providerKey": "nordmann@eatelligent.no"  
    },  
    "firstName": "Ola",  
    "lastName": "Nordmann",  
    "email": "nordmann@eatelligent.no",  
    "image": null,  
    "role": "user",  
    "created": "2015-04-13T16:47:04.402",  
    "recipeLanguage": null,  
    "appLanguage": null,  
    "city": null,  
    "country": null,  
    "sex": null,  
    "yearBorn": null,  
    "enrolled": false,  
    "metricSystem": true  
  }  
}
```

Figure 3.4: The sign up process



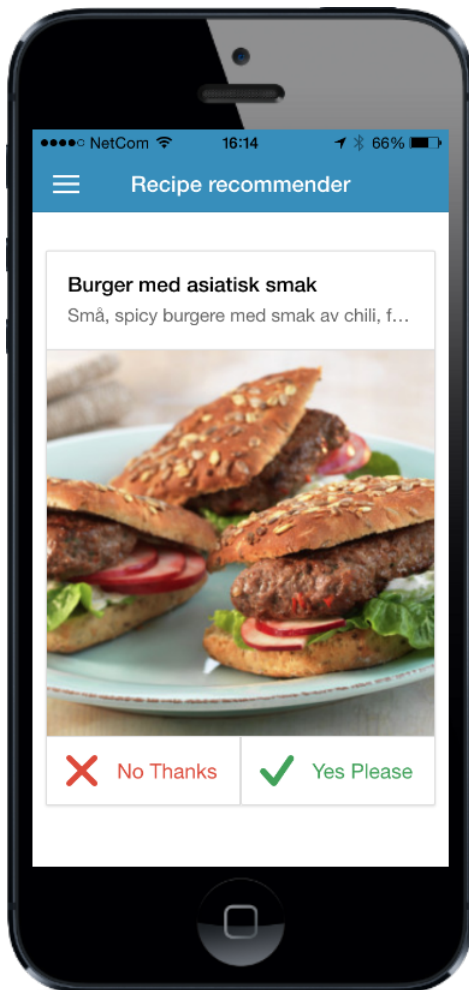
Listing 3.3: Cold-Start Request

```
{  
  "coldStartId": 3,  
  "answer": true  
}
```

Listing 3.4: Cold-Start Response

```
{  
  "ok": true,  
  "answer": {  
    "userId": 14,  
    "coldStartId": 3,  
    "answer": true,  
    "answerTime": "2015-04-15T12:12:36.591"  
  }  
}
```

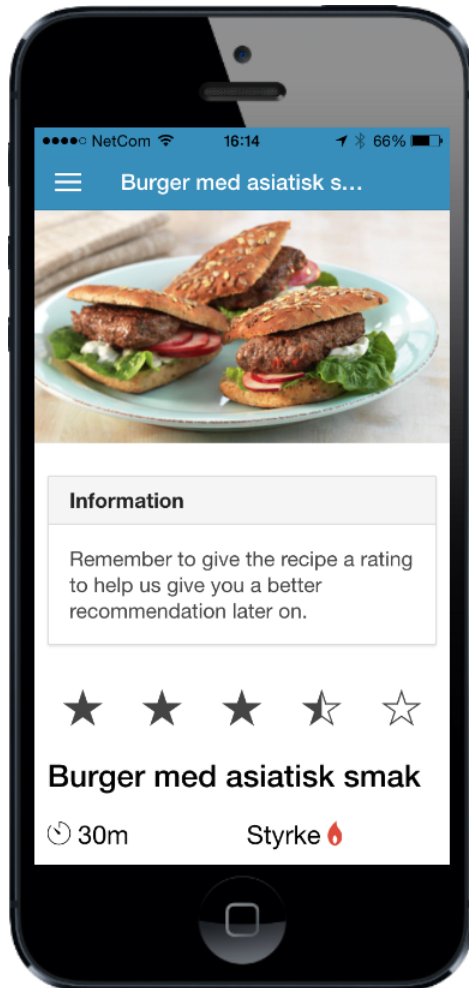
Figure 3.5: The user answer 1/5 questions



Listing 3.5: Recommendations

```
{
  ok: true,
  recommendations: [
    {
      __meta__: {
        userId: 1,
        recipeId: 14,
        source: "CF",
        predictedRating: "4.5779"
      },
      recipe: {
        id: 14,
        name: "Burger med asiatisk smak",
        image: "http://res.cloudinary.com/hnjelkrui/image/upload/v1423563013/jmdigci5h3k7si69e7wz.jpg",
        description: "...",
        spicy: 2,
        time: 30,
        difficulty: "Enkel"
      }
    },
    ...
  ]
}
```

Figure 3.6: The users answers “yes” or “no” to a recommendation



Listing 3.6: Recipe

```

{
  ok: true,
  recipe: {
    id: 14,
    name: "Burger med asiatisk smak",
    image: "http://res.cloudinary.com/hnjelkrui
            /image/upload/v1423563013/jmdigci5h3k7
            si69e7wz.jpg",
    description: "...",
    language: 1,
    calories: "0",
    procedure: "...",
    spicy: 2,
    time: 30,
    difficulty: "Enkel",
    source: "http://www.matprat.no/opskrifter/
            familien/burger-med-asiatisk-smak/",
    created: "2015-02-10T10:09:51.636",
    modified: "2015-03-12T14:31:51.965",
    published: "2015-03-11T14:11:55.016",
    ingredients: [
      {
        id: 70,
        name: "Kjøttdeig",
        unit: "gram",
        amount: "125"
      },
      ...
    ],
    tags: [
      "Familien",
      "Asiatisk"
    ],
    currentUserRating: null,
    averageRating: "4"
  }
}

```

Figure 3.7: Detailed view of the recipe

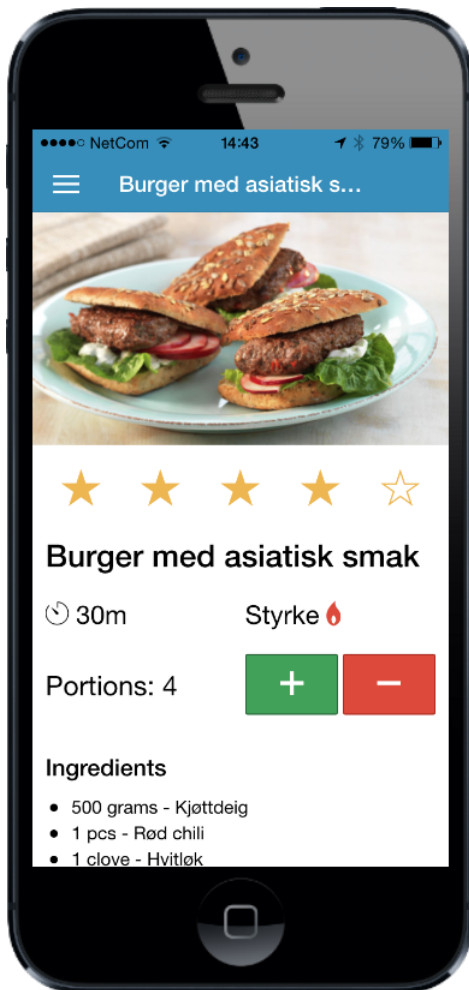


Figure 3.8: Rating the recipe

Listing 3.7: Rating request

```
{  
  "recipeId": 14,  
  "rating": 4.0  
}
```

Listing 3.8: Rating response

```
{  
  "ok": true,  
  "rating": {  
    "userId": 14,  
    "recipeId": 14,  
    "rating": "4",  
    "created": "2015-04-15T14:46:54.497"  
  }  
}
```

3.4 Recommender System

The client reaches the recommender system through one of the API endpoints, and the recommender engine is a part of the server. This was done to

have easy access to the database. This section describes the implementation details of the recommender systems.

3.4.1 Algorithms

We have implemented three different algorithms that work separately to recommend recipes for a user. Here we will describe each of them in detail.

Nearest User Rated Recipe (NURR)

The first thing the user does in the application is to answer a few questions regarding taste and expertise on the kitchen. The five questions asked is the following:

- Do you like meat?
- Do you like spicy food?
- Do you view yourself as a good chef?
- Do you like fish?
- Do you like chicken?

Each of the cold-start questions affects the weights for the current user. These numbers correspond to how much each ingredient tag is weighted by the Euclidean distance for the KNN algorithm (see Section 2.4.1). Table 3.2 displays the matrix of weights for each question. This information will help the KNN calculate distances between users the first few times before it has enough ratings from explicit star ratings. A negative answer to one of the questions will result in updating the corresponding ingredient tags with a negative sign. The weights have a maximum cap at 5 and minimum of -5. This maximum cap is to avoid that weights do not sum up to infinite values and to make it possible to change the taste.

All these questions will have a boolean answer (“yes” or “no”) which we are using for similarity measure between two users. We define \vec{u}_a as the vector of the responses for user a . Equation 3.1 defines the similarity measure used

	Meat	Spicy	Skills	Fish	Chicken
Spice	–	0.5	–	–	–
Spicy	–	1.0	–	–	–
Salty	–	0.2	–	–	–
Composed	–	–	-1.0	–	–
Fish	–	–	–	1.0	–
Seafood	–	–	–	0.8	–
Animal product	0.3	–	–	0.1	0.2
Chicken	–	–	–	–	1.0
Meat	1.0	–	–	–	–
Pork	0.8	–	–	–	–
Beef	0.8	–	–	–	–

Table 3.2: The weighting from the cold-start questions

to compare two users cold-start answers. The difference between two vectors is defined in Equation 3.2 where $u_{a,i}$ is the i th answer from user a and n is the number of questions, in this case, 5.

$$\text{sim}(\vec{u}_a, \vec{u}_b) = \frac{1}{\sqrt{\text{diff}(\vec{u}_a, \vec{u}_b) + 1}} \quad (3.1)$$

$$\text{diff}(\vec{u}_a, \vec{u}_b) = \sum_{i=1}^n |u_{a,i} - u_{b,i}| \quad (3.2)$$

For example, if user A and user B have the following responses:

User A responses: yes, yes, no, no, yes

User B responses: yes, no, yes, no, yes

The similarity between these users is calculated in Equation 3.3.

$$\begin{aligned} \text{sim}(A, B) &= \frac{1}{\sqrt{\text{diff}(A, B) + 1}} \\ \text{sim}(A, B) &= \frac{1}{\sqrt{|1 - 1| + |1 - 0| + |0 - 1| + |0 - 0| + |1 - 1| + 1}} \\ \text{sim}(A, B) &= \frac{1}{\sqrt{3}} \end{aligned} \quad (3.3)$$

<p>Algorithm 1: Nearest User Rated Recipes</p> <p>Data: currentUser - current user, users - all other users, <i>n</i> - number of recipes to return, foreach user <i>in</i> users do user.similarity \leftarrow sim(currentUser, user) end users \leftarrow users filtered on user.similarity > minThreshold recipes \leftarrow empty collection foreach user <i>in</i> users do r \leftarrow retrieveTopRatingsForUser(user, maxRecipesFromOneUser) foreach ratedRecipe <i>in</i> r do ratedRecipe.score \leftarrow user.similarity \times ratedRecipe.rating recipes \leftarrow recipes + ratedRecipe end end return recipes.sortBy(recipe.score)[0 : <i>n</i>]</p>

Based on this similarity measure, we can find the most similar users. These users ratings will be measured with the same similarity, and be the product of their actual rating and the similarity to that similar user. For example, say the similarity is 0.8, and the other user have rated a recipe 4 stars, then the score for this recipe will be $0.8 * 4 = 3.2$. Note that this score is not a rating, but a scoring used to rank the different recipes to return the top *n* recipes. Algorithm 1 describes this procedure in pseudocode. This algorithm is designed by ourselves and is inspired by the first two steps from CBR (see Section 2.4).

Content-Based Filtering (CBF)

One or multiple recipe tags marks every recipe, to group the recipes. These tags are set by ourselves. We do the same thing with the ingredients, with another set of tags that we call ingredient tags. The system is developed with a relation between user and both the recipe and ingredient tags so that a score for the user is stored for the tags. This forms a user profile as mentioned in Section 2.1.1, and Figure 3.9 and 3.10 illustrates this relation. The scores on

the tags are made up from three different sources:

- Cold-start questions: Some of the questions maps to some ingredient tags. For example, if a user says that he does not like fish, the system will store a negative score for the tags seafood and fish. A value between -1 and 1 updates the score, and are set by ourselves with the best guess that we can do between the relations.
- “Yes” or “no” to a recipe: When the user answers “yes” or “no” to a recommended recipe in the application, we take that as a weak indication that the user likes or dislikes the recipe. A value of -0.3 or 0.3 updates the corresponding recipe tags and the tags for the ingredients in the recipe.
- Rating: A rating on a recipe is considered to be a stronger indication if the user likes or dislikes it. The different possible values decide how much the score should be updated. These impacts is listed in Table 3.3.

★	★★	★★★	★★★★	★★★★★
-1	-0.5	0	0.5	1

Table 3.3: The impact of star ratings

The score each user has for the ingredient and recipe tags, makes it possible to compute a score for each recipe for a user, by aggregating the tag scores of the tags in a recipe. Algorithm 2 shows how this is calculated. The recipes are first filtered on some preferences for the user, based on two questions he answered when he started using the app. If he answered no to “Do you like spicy food”, we filter out the recipes that are spicy. The recipes belong to one out of three levels of spiciness, and the people who do not like spicy food, will only get the lowest category. If the user likes spicy food, we do not filter anything because we believe that people who like spicy food not necessarily only want the spicy recipes. The same strategy are done with how long time it takes to make the food, and how hard the recipe is to make. If the user answered no to “Do you view yourself as a good chef?”, we filter out the recipes that are not in the *easy* category (out of *easy*, *medium*, and *hard*), and the ones that take more than 45 minutes to make. After this filtering is done, a score is calculated for each recipe based on the scores the user have

on the recipe and ingredient tags in the recipe, and the top n recipes are returned. We designed this algorithm ourselves, but inspiration is taken from earlier projects. E.g., when CHEF (see Section 2.5.3) knows about the link between chicken and meat.

Algorithm 2: Content-based filtering

Data:

recipes - all recipes,

user - user preferences

urt - user score on all recipe tags

uit - user score on all ingredient tags

 n - number of recipes to return,

Result: Top n recipes

```

if not user.likeSpicyFood then
  | recipes  $\leftarrow$  recipes filtered on not spicy
end
if not user.isAGoodChef then
  | recipes  $\leftarrow$  recipes filtered on time < 45 min. and easy difficulty
end
foreach  $r$  in recipes do
  | score  $\leftarrow$  0
  | foreach recipeTag in r.recipeTags do
  | | score  $\leftarrow$  score + urt[recipeTag]
  | end
  | foreach ingredient in r.ingredients do
  | | foreach ingredientTag in ingredient.ingredientTags do
  | | | score  $\leftarrow$  score + uit[ingredientTag]
  | | end
  | end
  | r.score  $\leftarrow$   $\frac{\text{score}}{|\text{recipeTags}| + |\text{ingredientTags}|}$ 
end
recipes  $\leftarrow$  recipes filtered on score > 0
return recipes.sortBy(r.score)[0 :  $n$ ]

```

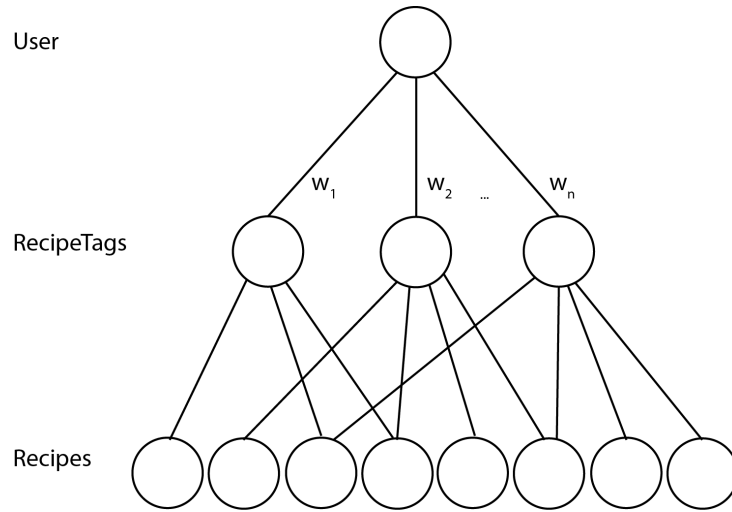


Figure 3.9: Illustration of the recipe tag tree

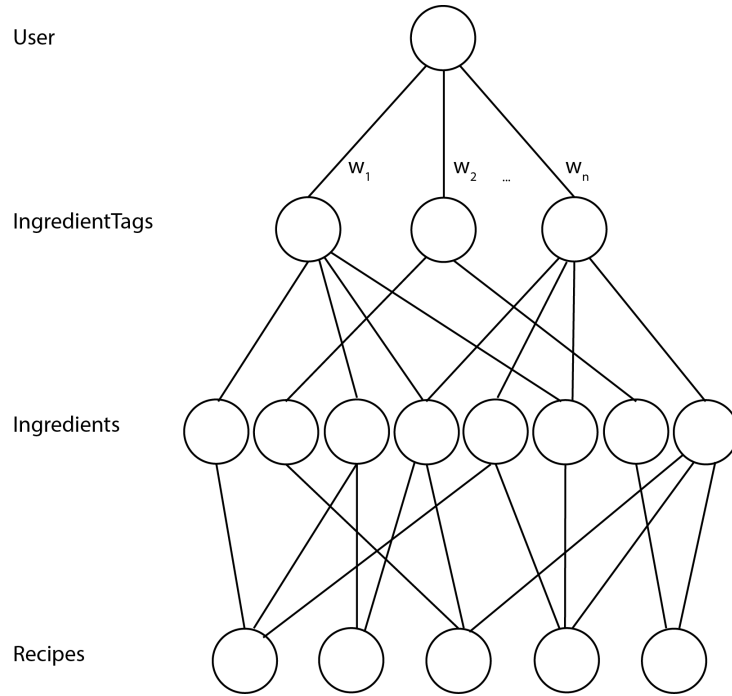


Figure 3.10: Illustration of the ingredient tag tree

Collaborative Approach

As a helper for the collaborative filtering mechanism we used the open source project from GroupLens called LensKit¹⁰. LensKit is an implementation of collaborative filtering algorithms along with a set of tools support of such systems. LensKit is split up in different modules to help developers only to use the parts needed for the particular implementation. This projects uses two of this modules, `lenskit-core` and `lenskit-data-structures`.

When building a recommender with LensKit, the core module is required and is a dependency for the other modules. For this implementation, the core is used to initialize the building of the collaborative filtering recommendation. An item recommender is built from an access object to the database where it will gain access to the other users, ratings and items (recipes). This method uses KNN to retrieve similar items where the similarity is calculated by a cosine vector similarity. This approach is a memory-based method where all the calculations are done when the user asks for a list of recommendations.

Some problem specific configuration is needed, to help LensKit compute the recommendations. The `lenskit-data-structures` module provides data structures to help with this configuration

3.4.2 Feedback in the Client

As described in Section 2.2, feedback is necessary to understand what preferences each of the users have. In the client, we gather both explicit and implicit feedback. Two different explicit feedback mechanisms are used: (1) After a recommendation list is presented, the user answers either “Yes please” or “No thanks”. (2) When a user views a recipe, he can report a star rating from 1 to 5 stars.

We gather the implicit feedback in two ways: (1) When a user looks at a recipe, a timer is started. This timing gives us an indication whether or not the user made the dish. (2) For every request, timestamps are logged. These timestamps helps us to not recommend the same recipes within the same period for a user.

¹⁰The LensKit project: <http://lenskit.org/>

3.4.3 Cold-Start Item Problem

Pushing items that do not have ratings would help to promote new items, and let the users discover the whole set of recipes. By randomly picking some recipes, both new and recipes with no rating all have a small chance of being promoted. This strategy makes the evaluation of the system easier since the baseline are the random recipes, and the cold-start item problem is not the primary focus of Eatelligent at this moment. Therefore, no special promotion strategy is implemented for new items.

3.4.4 Composition of Recommendations

The composition of the recommendation list of any size is based mainly on the collaborative filtering engine and the CBF algorithm. These two algorithms compose 80% of the list (40% from each). When there is not enough knowledge for the collaborative filtering or CBF algorithms to find enough recipes, the NURR algorithm fills in the rest. For CBF, this happens because of the threshold that is defined. With a lower score than this threshold, the recipes are excluded. For CF, this can be when the user have not rated any recipes and the engine has nothing to compare against. The last 20% of the list is a set of random recipes. This strategy is done to be able to compare the other algorithms to random in the evaluation, and to help integrating new recipes without ratings. To give all methods a fair chance we randomize which of the algorithms to get the top of the list. Therefore, the recommendation list is not deterministic. The random contribution helps the unrated recipes to gather some ratings as well.

All the different set of recipes from the recommendation methods except the ones we choose at random are filtered on additional attributes. The recipes that are not rated the last 14 days, not been seen in total the last seven days, nor been said no to in the last three days, are returned. This filtering is done to make sure the user do not get the same recipes recommended every time. Table 3.4 shows an example of a list of recommendations where the collaborative filtering results on top. The next 5 to 7 items is from the CBF algorithm. Since CBF only filled three out of four possible slots, the NURR algorithm had to fill in one recipe for this example.

#	From	Predicted Rating	Similarity to User	Score	RecipeId
1	CF	4.4	–	–	5
2	CF	4.1	–	–	6
3	CF	3.7	–	–	4
4	CF	3.7	–	–	7
5	CBF	–	–	3.88	42
6	CBF	–	–	2.75	2
7	CBF	–	–	1.27	9
8	NURR	–	0.7	–	1
9	Random	–	–	–	10
10	Random	–	–	–	23

Table 3.4: An example on a composed recommendation list

3.5 How it Works

To describe the functionality with the weights and new recommendations system in the system, we will show an example of a user going through the following steps: A recommendation, answering “yes” or “no” on a few recipes, picking a recipe and rating the recipe and receiving new recommendations in the end.

3.5.1 A New User Receiving a List of Recommendations

After the user have signed up, they will be prompted with five questions as mentioned earlier. The ingredient tag weights will be updated from 0 to a more personalized weighting in respect to the cold-start questions answered. For this example, say the user have answered the following:

Meat: Yes, Spicy: No, Skills No, Fish: Yes, Chicken: Yes

After processing these questions, the user will have the following weights stored in the *user_ingredient_tag* and *user_recipe_tag* relations shown in Table 3.5. Note that *composed*, *spicy*, *spice*, and *salty* got the negative value due to the negative answer on the previous questions. An explanation of each ingredient tag and recipe tags is listed in Appendix C. Answering this

Ingredient Tag	Value
Meat	1.0
Animal product	$0.3 + 0.2 + 0.1 = 0.6$
Pork	0.8
Beef	0.8
Chicken	1.0
Seafood	0.8
Fish	1.0
Composed	1.0
Spicy	-1.0
Spice	-0.5
Salty	-0.2

Recipe tag	Value
Healthy	0
Asian	0
Quick	0

Table 3.5: The current state of the weights after the cold-start questions.

questions does not affect the recipe tags. These tags are updated after ratings of the recipes (described in Section 3.4.1), not by answering cold-start questions.

The user is presented a list of recommendations from different recommendation engines, and then the list is logged in the database to gather results. For our example, the output from the recommendation endpoint is shown in Listing 3.9. Note that the CF algorithm returns no items before the user performs a star rating on a recipe. This is because the CF algorithm exclusively does calculations based on this type of rating, and cannot do a comparison with an empty user-rating vector.

After the ordered list of 10 items is displayed, the user will say yes or no to the recipes recommended. For our example, say the user says no to the first two recipes and wants to check out the third one. This feedback is then used to update the users weights. The update depends on the recipe and what ingredients that the recipe contains because of the ingredient tags that is updated. Of course, this is a rather “harmless” choice, unlike a star rating. The results of this examples behavior is listed in Table 3.6

Ingredient Tag	Difference	New value
Meat	1 negative = -0.3	0.9
Animal product	1 negative, 1 positive = 0	0.6
Pork	1 negative = -0.3	0.5
Beef	1 negative = -0.3	0.5
Chicken	1 positive = 0.3	1.3
Seafood	–	0.8
Fish	–	1.0
Composed	1 negative, 1 positive = 0	1.0
Spicy	1 negative, 1 positive = 0	-1.0
Spice	1 negative, 1 positive = 0	-0.5
Salty	1 positive = 0.3	0.1
Extras	1 negative, 1 positive = 0	0.0
Green	2 negative, 1 positive = -0.3	-0.3
Fruit	1 negative = -0.3	-0.3
Milk product	1 positive = 0.3	0.3

Recipe tag	Difference	New value
Healthy	1 positive = 0.3	0.3
Asian	2 negative = -0.6	-0.6
Quick	1 negative, 1 positive = 0	0

Table 3.6: The current state of the weights after the “yes/no” ratings

Listing 3.9: First list of recommendations to a user

```
{
  ok: true,
  recommendations: [
    {
      recipe: { name: "Babi Asam Manis" }
    },
    {
      recipe: { name: "Asian omelet" }
    },
    {
      recipe: { name: "Paprika Chicken with asparagus" }
    },
    ...
  ]
}
```

The selected recipe, “Paprika Chicken with asparagus” is chosen by the user as today’s dinner (for a full description of the recipe, see Section C.3.1). A timer is present and reported back to Eatelligents backend while the recipe is viewed by the user. The user then rates the recipe, say 4 star rating. In Table 3.3, a 4 star rating corresponds to a weight of 0.5 times every ingredient tag. Table 3.7 shows the updated weights after the feedback is given.

Now, when the user have given both implicit and explicit feedback to the recommendation engine, we have some more content to base the next recommendations on. This information helps the other users (collaborative part), and the user itself the next time some user asks for a recipe list recommendation. In the last part of the example, the next recommendation list is aimed more at the users preferences. Table 3.7 shows that the tags chicken, animal product and composed ingredients is something this user likes.

Listing 3.10 shows the next recommendation after the actions described above is carried out. The CF recommender proposes a chicken-based recipe, due to similar users and their rating history. From the NURR method, a fish dish is suggested. The most similar users ratings explain this suggestion. The CBF algorithm presents another chicken dish due to the positive weight to the chicken and animal product tags. The items ranking in the list are randomized, where different parts of the list are the top. This randomization is because we want to expose recommendations from all recommender methods

Ingredient tag	New value
Spicy	$-1.0 + 0.5 = -0.5$
Salty	$0.1 + 0.5 = 0.6$
Spice	$-0.5 + 0.5 = 0$
Green	$-0.3 + 0.5 = 0.2$
Extras	$0 + 0.5 = 0.5$
Milk product	$0.3 + 0.5 = 0.8$
Chicken	$1.3 + 0.5 = 1.8$
Animal product	$0.6 + 0.5 = 1.1$
Composed	$1.0 + 0.5 = 1.5$

Recipe tag	New value
Healthy	$0.3 + 0.5 = 0.8$
Asian	-0.6
Quick	0

Table 3.7: The updated weights after a rating of “Paprika Chicken with asparagus”

Listing 3.10: New recommendation after interactions

```

{
  ok: true,
  recommendations: [
    {
      __meta__: { source: "CF", predictedRating: 4.45 },
      recipe: { name: "Curry chicken with banana and fennel" }
    },
    {
      __meta__: { source: "NURR", similarityToUser: 1.0 },
      recipe: { name: "Fishburger" }
    },
    {
      __meta__: { source: "CBF", score: 4.1 },
      recipe: { name: "Chickenwok with sweet and sour" }
    },
    ...
  ]
}

```

as the first recipe. Furthermore, this ensures that the user is unbiased as to which recommender is employed in retrieving the various recipes.

3.6 Making Eatelligent Public

As described in Section 3.3.1, architectural choices is made to make the publication to most users a relatively simple process. The 25th of March 2015, the mobile client was published in Google Play and Apple App Store. To have both Android and iPhone users, help gather diversity in the user mass to reveal possibly differences [22]. The project's homepage at <http://eatelligent.no> promoted the application along with social media and the word of mouth. Chapter 4 describes the results of the gathered data.

Chapter 4

Experiments and Results

In this chapter, we describe our experiments along with the corresponding results. We start by listing the experimental plan, followed by the experimental setup, and in the end we show the results.

4.1 Experimental Plan

Multiple ways exist to evaluate the recommender system. The most general one is to look up the difference between the user ratings on recipes picked by a random generator, and compare it with the ratings on recipes given by a recommender algorithm. With enough data, this approach makes it easy to say if the algorithms are intelligent at all, and it is easy to compare the different recommendation approaches. Since the logs of the recommended recipes always contain the name of the algorithm promoting it, this can be aggregated easily. For both the star ratings and the “yes/no” ratings, this strategy will be done. The stars contain the strongest indication of whether the user liked the recipe or not, and the “yes/no” ratings tell us if the user found that recipe interesting.

For the collaborative filtering algorithm, the recommended recipes contain a predicted rating, which also enables us to evaluate the difference between the predicted and the actual user rating, once a user rates a recipe. This evaluation is measured by the mean absolute error (MAE). This error is defined as the absolute value of predicted rating, p , subtracted by the actual

user rating, a . The mean of these errors is the MAE and explains how far the algorithm is from the optimal predictions.

$$\text{MAE} = \frac{1}{n} \sum |p - a| \quad (4.1)$$

For all the different recommender algorithms, it is interesting to plot the performance over time since the accuracy should increase. Both the timeline for a new user until he has several ratings, but also how an algorithm performs with few vs. many users in the system. It could also be interesting to evaluate the cold-start item problem after a while, by releasing some new ones, and see if they are picked up by the recomender system or not. However during our test period, we do not have enough data to answer all these questions, and a deeper evaluation must wait until later work.

Since a recommender system needs much data to get better, the application is released on the open market in both App Store and Google Play. Therefore, we did not know beforehand how many users we were going to include in our tests, but we report the numbers in the experimental setup.

When the user gets a recommendation, the server gives ten for each request, but the user only sees one. He can swipe “yes” or “no”, and if “no”, he gets the next one. This approach enables us to evaluate how many recipes the user on average looks at before he chooses a recipe. A lower number would be better.

4.2 Experimental Setup

The data for this chapter describes the application after ten weeks on the open market, available for anyone to test. Explicit feedback from the users, either by rating a recipe 1-5 or by answering “yes” or “no”, forms the basis of our results.

The number of items (recipes) the recommender can choose from is 100. The number of users is 109. These users have together used the application for 1126 minutes, giving each user 10.5 minutes in average. The users have given 5071 “yes/no” ratings and 87 star ratings. All users have answered the 5 cold-start questions described in Section 3.4.1. They have together viewed 413 recipes and a list of ten recommended recipes is generated 924 times.

4.3 Experimental Results

This section will present the results collected during our test period.

4.3.1 Yes/No ratings

When presenting the recommendations, the users answer either “yes” or “no” to the recipe. Figure 4.1 shows the distributions of answers between the algorithms. The different algorithms do not have an equal number of responses. Out of the ten recommendations given each time, some of the algorithms will not always output recipes, and on average collaborative filtering outputs more. This outcome happens because both CBF and NURR requires the recommendations to above a certain threshold so that the algorithms rather outputs nothing than bad recommendations.

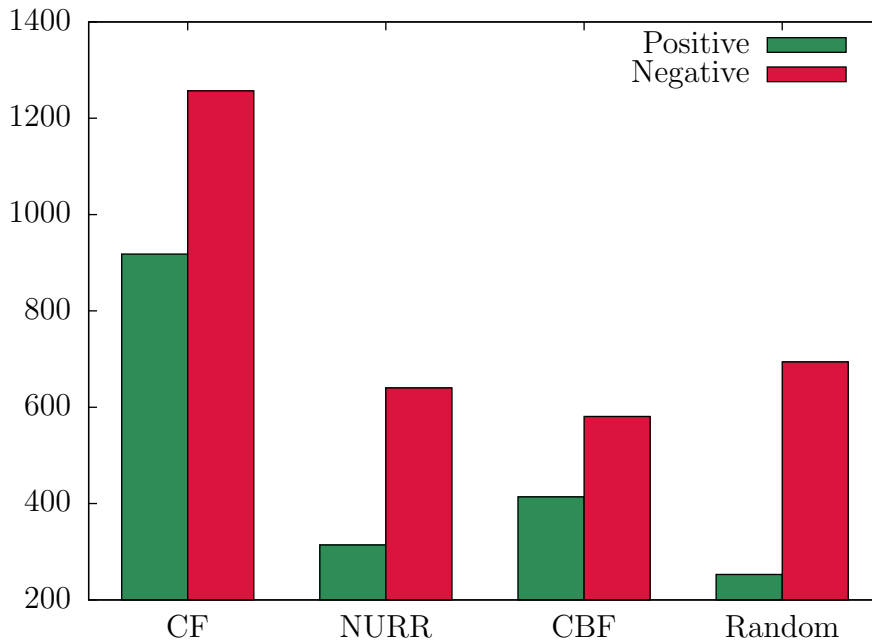


Figure 4.1: “Yes/no” ratings per algorithm

To compare the different algorithms on a normalized level, Figure 4.2 shows the same data as a stacked histogram.

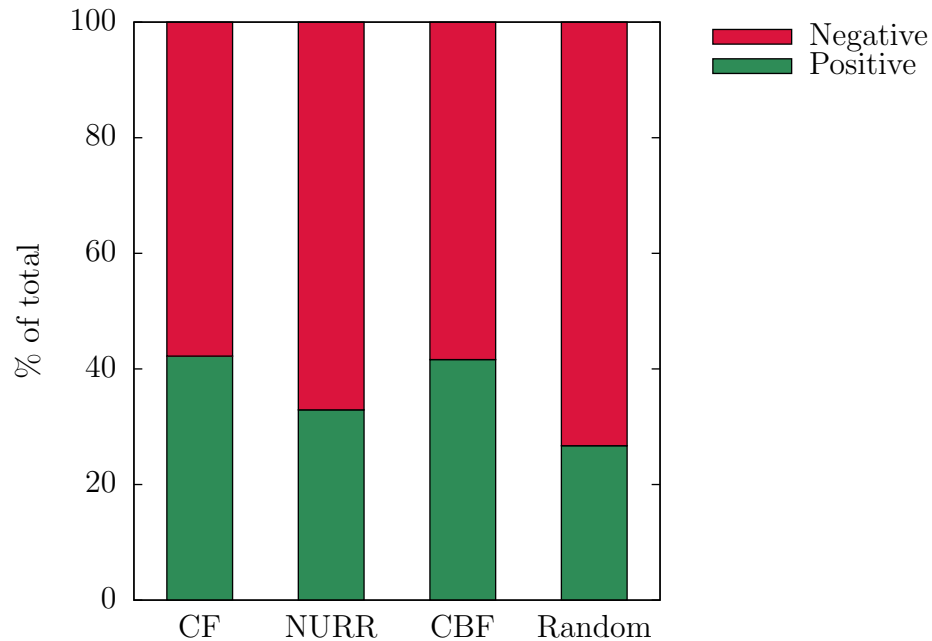


Figure 4.2: Normalized “Yes/no” ratings per algorithm

We can see that CF and CBF do a relatively good job in recommending recipes that the user wants to check out further. Compared to the random items given, all three procedures performs better (ratio between positive and negative answers).

4.3.2 Star Ratings

Figure 4.3 shows the distribution in rating of stars and this reveals the data basis of star ratings. As we can see, it is a lot more of 4 and 5 ratings compared to the others. We assume this is because people are more likely to make the recipes they think they will like, and, therefore, most people will filter out the recipes before they consider rating them.

Figure 4.4 shows the normalized distribution of stars ratings per algorithm. We can see that random have a relatively large amount of low ratings compared to the other algorithms. It is also small differences between the different algorithms, but we should be careful about analyzing it too much since the

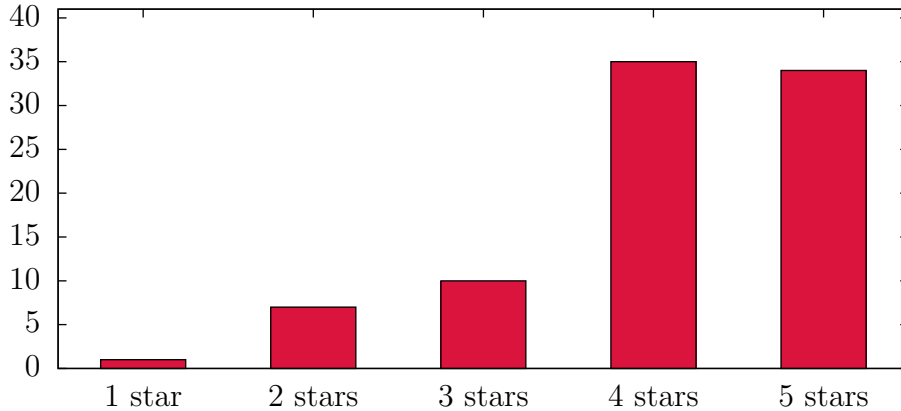


Figure 4.3: Star ratings

Algorithm	No. ratings
NURR	8
CF	38
CBF	27
Random	12

Table 4.1: The number of ratings for each algorithm

number of ratings is low as shown in Table 4.1.

2.38 is the average number of recommended recipes the user checks out before he select one. This average number reflects that the users both want to explore the recommendations deeply, but at the same time receives recipes that are interesting in a reasonable time. A number that is high (closer to 10, because of the limitation of recommendations in one request), would affect the users judgment of the recommendations. A small number would describe a user mass with a sense of “yes to everything” attitude.

The recommended recipes from CF that ended up with a rating had an MAE of **0.71**. This number is only possible to calculate for this algorithm because the other algorithms does not predict a rating, but calculates a score or a similarity. This number does not have a rock solid ground due to the user mass, but it describes a good prediction at least in an early stage of the

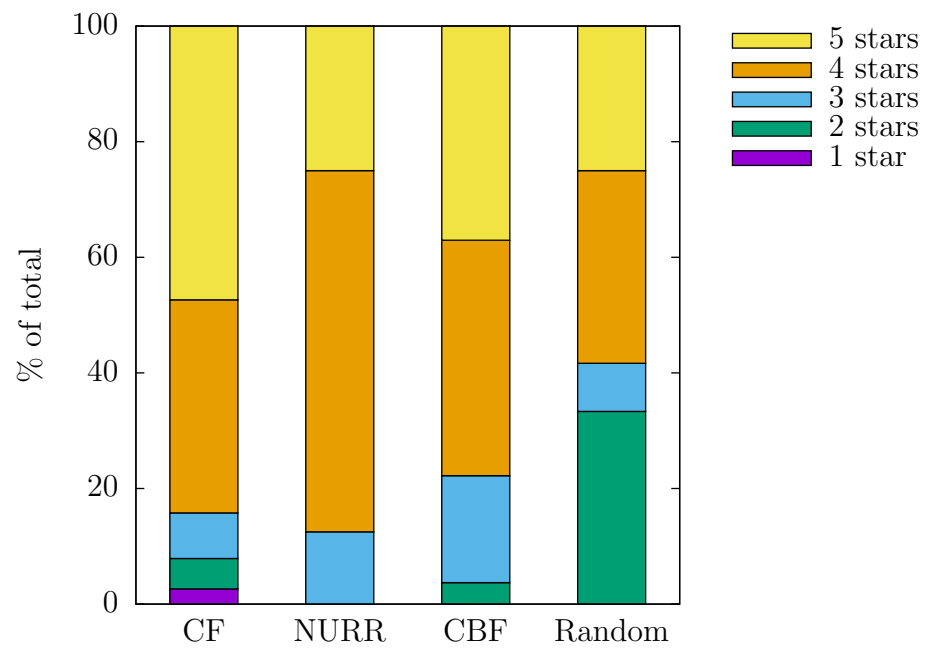


Figure 4.4: Percentage of star ratings for each algorithm

system.

The results presented is from a tiny data set and, therefore, these results do not necessarily describe the correct nature of the algorithms. Still Figure 4.3 and 4.4 shows a difference between the intelligent methods vs. randomly given recipes.

4.3.3 Comparing Yes/No and Star Ratings

The recommendations are based upon three types of user feedback; answer to cold-start questions, “yes/no” ratings, and star ratings on the recipes. However, the different algorithms do not consider all the types of feedback themselves. NURR and CF both focus on finding similar users and their highly-rated recipes, but they lack learning by the “yes/no” feedback. The “yes/no” ratings can be aggregated for each recipe, and if we consider “yes” as 1 and “no” as -1, we can calculate a sum for each recipe. Based on results from Figure 4.1, we see there is an overweight of “no” ratings. Therefore, a negative sum is common. However, as shown in the results in the upper chart in Figure 4.5 (which illustrates this aggregation), some of the recipes are doing significant worse than the average. These recipes all have relatively good average rating (bottom chart). We have reason to believe that this is caused by people not giving star ratings on the recipes they are not interested in. Therefore, the number of stars will remain high while people say “no thanks” to the recipe. Since both NURR and CF do not learn from the “yes/no” feedback, they continue to promote the recipes with good rating, and this explains why a “bad” recipe is recommended many times.

4.3.4 Cold-Start Questions

The GINI index can be used to say something about the cold-start questions ability to split the user mass into different buckets. This technique calculates a number between 0 and 0.5 where 0.5 is a perfect split on a boolean question. Equation 4.2 shows how to calculate the GINI index for each question.

$$\text{GINI} = 1 - \left(\frac{\text{No. yes}}{\text{No. users}} \right)^2 - \left(\frac{\text{No. no}}{\text{No. users}} \right)^2 \quad (4.2)$$

Table 4.2 lists the number of positive and negative responses to the cold-start questions. The three of the questions with GINI score below 0.25 is relatively

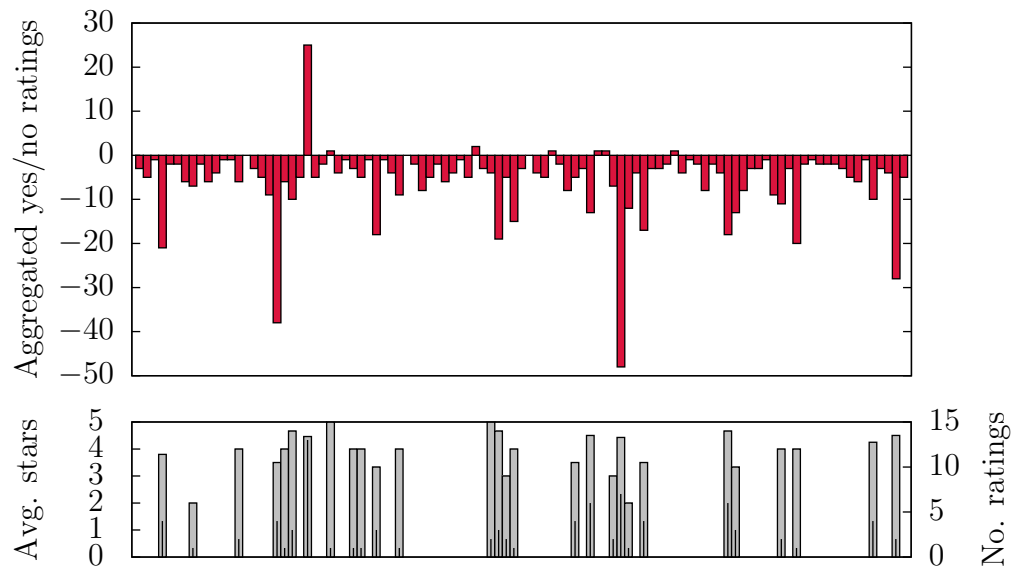


Figure 4.5: Comparison between yes/no and star ratings. The data for each recipe corresponds vertically between the two charts. In the bottom chart, the impulses represent the number of ratings, and the bars show the average star rating.

bad questions compared to the others. Still, these questions help the few of the users to get in the right direction according to the weights. The question regarding spicy and skills in the kitchen does, on the other hand, separate the user mass to a greater extent. However, this numbers does reflect the impact of each question to the final recommendations.

Question	Yes	No	GINI
Do you like meat?	96	13	0.21
Do you like spicy food?	82	27	0.37
Do you view yourself as a good chef?	75	34	0.43
Do you like fish?	94	15	0.24
Do you like chicken?	94	15	0.24

Table 4.2: Answers to the cold-start questions

Table 4.3 lists the number of recommendations made based on the answers from the cold-start questions. Positives are recommendations given that supports the response from the cold-start questions. Negatives are recommendations where the user have said no to a category in the cold-start questions, but the recommenders have disregarded this input. Note that the number of recommendations that is associated with spicy is relatively high. We have reason to believe that this is because some recipes got a good rating at an early point in time, and, therefore, got snapped up by the collaborative algorithms.

The numbers in Table 4.3 are not surprising due to the implementation of the algorithms but are relevant to demonstrate how they behave regarding the cold-start questions. CF is only considering star ratings and does not take the cold-start questions into consideration at all. Therefore, this algorithm recommends more recipes disregarding the answers. NURR is similar to CF since it looks at the ratings, but it looks at the cold-start questions for finding the similar users. However some users might have given a high score to recipes even if they do not correspond to their answers for the questions, and this might make the data noisy. CBF is not surprisingly doing best (less negative recommendations) here since that approach looks at the content, and the weights are initially set by the cold-start questions.

It would also be interesting to evaluate the ratings the users give the recipes

	CF		NURR		CBF	
	Pos.	Neg.	Pos.	Neg.	Pos.	Neg.
Meat	575	171	429	163	882	131
Spicy	4537	1867	1809	897	2623	0
Skills	262	108	50	22	70	0
Fish	23	13	0	0	71	9
Chicken	535	83	47	18	496	21

Table 4.3: Recommendations made by each algorithm based on cold-start questions

that are recommended negative vs. positive regarding the answers to the initial questions. However, we were not able to do this due to a tiny dataset.

Chapter 5

Evaluation and Conclusion

This chapter will evaluate the work covered in this thesis. First we present a discussion regarding the link to the research objectives presented in the introduction, followed by summarizing our contributions. In the end, we present some thoughts about what future work can do both with this project and recommendations of food recipes.

5.1 Evaluation and Discussion

Chapter 2 covers previous research regarding recommender systems, along with techniques for gathering feedback and methods to approach the cold-start problem. Previous ideas have inspired us in the making of the algorithms used in Eatelligent. We covered more theory and approaches to recommender systems than we had the time and data to test in our application, so in Section 5.3 we discuss some of the ideas for future research.

5.1.1 RO1 - Gain an understanding of what it means to recommend food recipes

We have done a deep dive into what features that are truly important in the food domain, with respect to recommendations. Our studies have unveiled that the basic primitives (ingredients) can be generalized into higher abstractions to gain more knowledge about the true model. This thesis does

not cover the nutrition element that is clearly a dimension of relevance for some groups of users, which can discover new interesting abstractions. We would also like to mention context as a factor in understanding what it means to recommend food recipes. The lack of understanding the context of a user is a weakness because we believe context is a significant factor of what a user choose to make for dinner.

5.1.2 RO2 - Study and evaluate different strategies to recommend items for a new user using both collaborative and content-based approaches

Since we wanted to evaluate different recommendation strategies we chose to implement this as a hybrid recommender system, by using the hybrid *mixed* approach which uses different algorithms separately. This approach enables us to compare the different algorithms easily with each other. Since collaborative filtering is the most common machine learning algorithm in the recommender systems today, we found it obvious to include this technique in our system. We also wanted to have one content-based approach, and this was achieved by grouping ingredient and recipes in different tags, and update a user profile continuously for these tags. These two approaches turned out to have very similar results for various reasons. Both CF and CBF have different shortcomings and strengths. CF recommends popular recipes and is doing worse on exploring the whole dataset. CBF gives recommendations based on learned personalized weights and pays attention to the cold-start questions and “yes/no” ratings, but it does not care about the star rating and hence can recommend less popular recipes.

The last approach, NURR, was created to have a technique that looks only at the answers to the five cold-start questions presented to a new user. This method performs worse than the other two but still better than random. The biggest problem with this approach is that NURR sometimes find one user that matches on the cold-start questions but if his best recipes are considered by most people to be bad, the algorithm does not consider any new feedback. As shown i Section 4.3.4, many people gives the same responses to the cold-start questions, and if one user with a popular response vector and good rating on a “bad” recipe, NURR will recommend this recipe to many people. To improve NURR, the algorithm needs to take advantage of “yes/no” ratings.

All three algorithms perform better than just giving out random recipes, so the results are promising, and all the approaches show to be useful in recommending recipes. However, all of them are still recommending more recipes which the user is not interested in compared to the recipes the user look at closer. One explanation is that people seems to have a browsing habit when looking at food recipes, and they want to look at multiple recipes before they decide to take one.

5.1.3 RO3 - Study existing solutions to the cold-start problem

In Chapter 2 we presented existing solutions to the cold-start problem. On this subject, much research is done, and we picked out the most common strategies and the ones that we believed could be useful in Eatelligent.

Recommender systems are continuously evolving to be more accurate and are widely used and researched by the industry. In the search for literature, it is always possible that we have been biased and that some methods are not discovered. However, we think we have discovered the most common and essential approaches forming a good foundation for this thesis.

5.1.4 RO4 - Study what kind of data we can collect from the application used by a user over a period of time that is relevant to build more knowledge about the user

We found that the collection of data from a user in the application context was a fairly easy task within limitations. To acquire more knowledge about *why* a user answers “yes” or “no” to a dish, a better model of the context needs to be understood. The context is not only important for a domain where food recipes are in focus, but also in other domains of recommendation.

Upon three types of user feedback, the recommendations are based; the answers to cold-start questions, “yes/no” ratings, and star ratings on the recipes. Our results show that all the types are relevant to learn from, but the different algorithms lack by not taking advantage of all of them. Improvements can be done by combining the “yes/no” and star ratings. Implicit ratings

often have roots in the usage patterns of the user mass, and due to the small amount of usage data, these type of recommendation enhancements was not implemented.

We have gained a deep knowledge of what types of feedback that are important, and how this feedback can be used to build up knowledge about the user.

5.1.5 RO5 - Study the challenges that arise when building an application with a recommender system from scratch

The thesis have faced challenges regarding the development of a full featured application with a mobile application and a server with a recommender system as a main focus. Evaluation of the system has been a challenge due to the number of users and usage within our time limits.

Not being able to evaluate the accuracy of the recommender systems during the development process made it hard to adjust parameters and the design of the recommender algorithms throughout the process. In the design of recommender systems with a dataset, it is easy to evaluate every step by testing the accuracy on the test set, but we had to wait until we had released the application. By releasing the application earlier we could have collected more feedback, and then improvements could be done to improve the recommendations. Within our time limits, this was not possible, and in future work the algorithms need to be tuned when more evaluation is done.

Since we developed this system as a new application and collected all the data in a short period with new users, the results are very sparse. The experiments should continue for a longer period, to derive more definitive conclusions.

5.2 Contributions

This section summarizes our contributions from this thesis.

Chapter 2 provides an overview of theory and previous research related to recommender systems, including problems like the cold-start problem and

user feedback. CBR is also described there as a problem-solving strategy to make recommender systems.

Eatelligent is available for any user (only recipes in Norwegian so far) in App Store and Google Play and is free to download. Improvements can be done to make the application more user-friendly, but we are happy to see that people already can have use of our product.

While star ratings or other similar ratings on items are the typical feedback of many recommender systems, we provide two types of explicit feedback; stars and a binary “yes/no” rating. The latter is less explicit and extracts knowledge about more items, not only the ones that the user likes. Our algorithms do not take enough advantage of the combination of these two types of ratings, but we provide the results and analysis that will help future implementations to do so.

It is not revolutionary to use tags to structure the data and use this information for recommendations, and we found this to be an efficient data structure for machine readable recipes. The tagging on ingredients is especially good since the number of ingredients does not increase significantly after adding enough recipes, and the tags can be entered manually without too much work. If we were to let users add recipes in the future, they would not need to add tags to the ingredients since those already exist.

With more user data, better evaluation can be done, but we provide an initial evaluation of the results in this thesis. We are careful to make too strong conclusions, but the results are already helping us to see what kind of improvements that need to be done.

All code written for Eatelligent is open source¹ and released under the MIT license. Hopefully, some people can take advantage of this in future research, and the different parts of the system can easily be modified to solve other types of problems than the recommendation of food recipes. The mobile application is self-contained, and can easily be modified to retrieve other types of data. Greater changes need to be done for the server and recommendation code to recommend in other domains, but for food recommendation it is a good framework to start with.

¹Source code: <https://github.com/eatelligent>

5.3 Future Work

While developing this recommender system, and also when studying the previous research, we discovered many ideas and useful features. It is not possible to make everything at once, and we had to prioritize what is the key functionality to answer the research objectives. In this section, we present the ideas that we think should be tried out in future research and new versions of Eatelligent.

A similar evaluation of the system needs to be done after collecting more data. A new evaluation will help to draw stronger conclusions and to tune the algorithms more precisely.

As we mentioned in Section 2.6.4, contextual information can be very interesting to consider in the composition of recommendations in Eatelligent. This approach was not implemented and tested, and should be tried out in future work. Some demographic info could be extracted by letting the user sign up with a social network profile. Geographic location can be used to track where the user is, and see if it can learn any patterns. If the user is on the way to the grocery store, he might care less about going for a recipe with many ingredients that he needs to buy. However, if he is home and want to make something with the ingredients he already has, the system should be able to take advantage of this. Either by taking some ingredients as input and output a recipe consisting of those or it could also assume some ingredients that he is likely to have.

This system focuses on the recommendation to one user. However, people are often making dinner for a party, and extracting this contextual information, and implement group recommendations, is an interesting problem. This feature is also likely to be something that many people would find useful in a food recommender system.

More work is needed with respect to which parameters are important in a food recipe. For example, in this thesis, nutrition is not emphasized and could help boost the personalization both for one user and group recommendations. This improvement could make it possible to use the number of calories as a feature, and also be able to recommend recipes that fit in a user's diet. For example, a user on a low-carbohydrate diet does not want recommendations containing many carbohydrates.

In this study, only the *mixed* hybrid method is used, and other approaches

should also be tried out. We think it is best to start to test the algorithms separately like we did, to evaluate them. Once we discover useful algorithms, combinations of these should be tried out and might result in better accuracy.

Mixtures of other hybrid approaches along with other algorithms such as model-based collaborative filtering could be an interesting comparison to the results covered in this thesis.

The cold-start weights listed in Table 3.2 is taken by our best guess based on our knowledge about food. From a data set to optimize the score for each user on the recipe and ingredient tags, these weights could be learned.

As mentioned in Section 3.4.3, methods for promoting new items and items without ratings are not implemented. This aspect of recommender systems is important for real world problems when new items are added frequently, and to find good strategies towards this issue more research needs to be done.

Appendix A

SQL-scheme

This appendix lists the PostgreSQL scheme used in this project.

```
CREATE TABLE language (  
    id serial8 PRIMARY KEY,  
    locale text NOT NULL UNIQUE,  
    name text NOT NULL UNIQUE  
);
```

```
CREATE TABLE recipe_tag (  
    id serial8 PRIMARY KEY,  
    name text NOT NULL UNIQUE  
);
```

```
CREATE TABLE unit (  
    id serial8 PRIMARY KEY,  
    name text NOT NULL UNIQUE  
);
```

```
CREATE TABLE ingredient (  
    id serial8 PRIMARY KEY,  
    name text NOT NULL UNIQUE,  
);
```

```
CREATE TABLE users (  
    id serial8 PRIMARY KEY,  
    first_name text ,
```

```

    last_name text ,
    email text NOT NULL UNIQUE,
    image text ,
    role text ,
    created timestamp ,
    recipe_language int8 REFERENCES language(id),
    app_language int8 REFERENCES language(id),
    city text ,
    country text ,
    sex text ,
    year_born int
        CONSTRAINT age_check
        CHECK (year_born > 1900 AND year_born < 2016),
    enrolled boolean ,
    metric_system boolean
);

CREATE TABLE recipe (
    id serial8 PRIMARY KEY,
    name text NOT NULL,
    image text ,
    description text ,
    language int8 NOT NULL REFERENCES language(id)
        ON DELETE CASCADE,
    procedure text ,
    spicy int
        CONSTRAINT spicy_check
        CHECK (spicy > 0 AND spicy < 4),
    time int
        CONSTRAINT time_check
        CHECK (time > 0),
    difficulty text ,
    source text ,
    created timestamp NOT NULL,
    modified timestamp NOT NULL,
    published timestamp ,
    deleted timestamp ,
    created_by int8 NOT NULL REFERENCES users(id)
        ON DELETE CASCADE
);

```



```
CREATE TABLE favorites (  
    user_id int8 REFERENCES users(id)  
        ON DELETE CASCADE,  
    recipe_id int8 REFERENCES recipe(id)  
        ON DELETE CASCADE  
);  
  
CREATE UNIQUE INDEX favorites_idx  
ON favorites(user_id , recipe_id);  
  
CREATE TABLE recipe_in_tag (  
    recipe_id int8 REFERENCES recipe(id)  
        ON DELETE CASCADE,  
    tag_id int8 REFERENCES recipe_tag(id)  
        ON DELETE CASCADE  
);  
  
CREATE UNIQUE INDEX recipe_in_tag_idx  
ON recipe_in_tag(recipe_id , tag_id);  
  
CREATE TABLE ingredient_tag (  
    id serial8 PRIMARY KEY,  
    name text NOT NULL UNIQUE  
);  
  
CREATE TABLE ingredient_in_tag (  
    ingredient_id int8 REFERENCES ingredient(id)  
        ON DELETE CASCADE,  
    tag_id int8 REFERENCES ingredient_tag(id)  
        ON DELETE CASCADE  
);  
  
CREATE UNIQUE INDEX ingredient_in_tag_idx  
ON ingredient_in_tag(ingredient_id , tag_id);  
  
CREATE TABLE ingredient_in_recipe (  
    recipe_id int8 REFERENCES recipe(id)  
        ON DELETE CASCADE,  
    ingredient_id int8 REFERENCES ingredient(id)
```

```

        ON DELETE CASCADE,
    unit_id int8 REFERENCES unit(id)
        ON DELETE CASCADE,
    amount real
);

CREATE TABLE user_star_rate_recipe (
    user_id int8 REFERENCES users(id)
        ON DELETE CASCADE,
    recipe_id int8 REFERENCES recipe(id)
        ON DELETE CASCADE,
    rating real
    CONSTRAINT rating_check
    CHECK (rating >= 0.0 AND rating <= 5.0),
    created timestamp,
    created_long int8,
    source text,
    data json
);

CREATE UNIQUE INDEX user_star_rate_recipe_idx
ON user_star_rate_recipe (user_id, recipe_id);

CREATE TABLE user_yes_no_rate_recipe (
    user_id int8 REFERENCES users(id)
        ON DELETE CASCADE,
    recipe_id int8 REFERENCES recipe(id)
        ON DELETE CASCADE,
    rating int,
    last_seen timestamp
);

CREATE UNIQUE INDEX user_yes_no_rate_recipe_idx
ON user_yes_no_rate_recipe(user_id, recipe_id);

CREATE TABLE user_viewed_recipe(
    user_id int8 NOT NULL REFERENCES users(id)
        ON DELETE CASCADE,
    recipe_id int8 NOT NULL REFERENCES recipe(id)
        ON DELETE CASCADE,

```

```

        duration int8 NOT NULL,
        last_seen timestamp NOT NULL
    );

```

```

CREATE UNIQUE INDEX user_viewed_recipe_idx
ON user_viewed_recipe(user_id , recipe_id);

```

```

CREATE TABLE logininfo (
    id serial8 PRIMARY KEY,
    provider_id text ,
    provider_key text
);

```

```

CREATE TABLE userlogininfo (
    user_id int8 NOT NULL,
    login_info_id int8 NOT NULL
);

```

```

CREATE TABLE passwordinfo (
    hasher text ,
    password text ,
    salt text ,
    login_info_id int8
);

```

```

CREATE TABLE cold_start (
    id serial8 PRIMARY KEY,
    image text NOT NULL,
    identifier text UNIQUE NOT NULL,
    description text NOT NULL,
    ingredient_tags hstore ,
    recipe_tags hstore
);

```

```

CREATE TABLE user_cold_start (
    user_id int8 NOT NULL REFERENCES users(id)
        ON DELETE CASCADE,
    cold_start_id int8 NOT NULL REFERENCES cold_start(id)
        ON DELETE CASCADE,
    answer boolean ,

```

```
        answer_time timestamp,
    );

CREATE TABLE user_ingredient_tag (
    user_id int8 REFERENCES users(id)
        ON DELETE CASCADE,
    ingredient_tag_id int8 REFERENCES ingredient_tag(id)
        ON DELETE CASCADE,
    value real
        CONSTRAINT value_check
        CHECK (value >= -5.0 AND value <= 5.0)
);

CREATE UNIQUE INDEX user_ingredient_tag_idx
ON user_ingredient_tag(user_id , ingredient_tag_id);

CREATE TABLE user_recipe_tag (
    user_id int8 REFERENCES users(id)
        ON DELETE CASCADE,
    recipe_tag_id int8 REFERENCES recipe_tag(id)
        ON DELETE CASCADE,
    value real
        CONSTRAINT value_check
        CHECK (value >= -5.0 AND value <= 5.0)
);

CREATE UNIQUE INDEX user_recipe_tag_idx
ON user_recipe_tag(user_id , recipe_tag_id);

CREATE TABLE given_recommendation (
    id serial8 PRIMARY KEY,
    user_id int8 REFERENCES users(id)
        ON DELETE CASCADE,
    recipe_id int8 REFERENCES recipe(id)
        ON DELETE CASCADE,
    type text ,
    ranking int ,
    created timestamp ,
    data json
);
```

Appendix B

Endpoints

The backend described in Section 3.3.2 consists of endpoints. These endpoints are used by the client to interact with the backend through HTTP. Every endpoint is prefixed with the server host.

B.1 GET

Endpoints that is reached with the GET method uses the auth token (current user) and returns either one entity or a list of entities. Below is a list of this projects endpoints that goes under the GET method.

- **/api/users** - Returns a list of all the users
- **/api/recommendation/recipes** - Returns a personalized a list with recipes
- **/api/users/:user_id** - Takes a `user_id` as a URL parameter and returns the user with its fields
- **/api/user** - Returns the current user entity
- **/api/ingredients** - Returns a list of all ingredients
- **/api/ingredients/:ingredient_id** - Takes a `ingredient_id` and returns a verbose version of the ingredient entity

- **/api/ingredients/tags** - Returns a list of the ingredient tags
- **/api/recipes** - Returns a list of all recipes
- **/api/recipes/:recipe_id** - Takes a `recipe_id` as a URL parameter and returns a verbose version of the recipe entity
- **/api/recipes/tags** - Returns a list of the recipe tags
- **/api/favorites/recipes** - Returns a list of the favorite recipes for the current user
- **/api/recipes/viewed** - Returns a list of the ten last viewed recipes for the current user
- **/api/coldstart** - Returns the cold-start questions

B.2 PUT

PUT endpoints are used to update an entity. These endpoints input a payload of what to update on the entity.

- **/api/user** - Inputs new user fields and returns the current users updated entity.
- **/api/recipes/:recipe_id** - Takes a `recipe_id` in the URL and new recipe fields and returns the updated recipe entity.

B.3 POST

Endpoints that use the POST method sends a payload to the server. This payload often used to create a new entity.

- **/api/users** - Creates a new user entity
- **/api/recipes** - Creates a new recipe along with new ingredients if they are not already stored in the database

- **/api/authenticate** - Inputs the user credentials and authenticates the user. This sets the current user for this session with a cookie
- **/api/ratings/recipes** - Inputs the `recipe_id` and a `star_rating` to give a star rating on a recipe from the current user
- **/api/favorites/recipes** - Inputs the `recipe_id` to save the recipe to the favorites list for the current user
- **/api/ratings/recipes/binary** - Inputs `recipe_id` and a boolean to give a “yes/no rating” to a recipe
- **/api/recipes/viewed** - Inputs the duration of how long a recipe was viewed for the current user
- **/api/coldstart** - Inputs the `cold_start_question_id` and a boolean to give feedback to the cold start questions
- **/api/knowledge/_reset** - Resets the weights for the current user

B.4 DELETE

DELETE endpoints delete an entity. The deletion does not permanently delete the entity but sets a flag in the database, marked is as deleted.

- **/api/recipes/:recipe_id** - Takes a `recipe_id` in the URL and deletes the recipe with this id
- **/api/favorites/recipes/:recipe_id** - Takes a `recipe_id` in the URL and removes the recipe from the favorites list for the current user.

Appendix C

Recipe Features

This appendix lists important features regarding recipes and how a they are structured.

C.1 Ingredient Tags

All the ingredients are mapped to either zero or some of the listed tags:

- Animal product
- Beef
- Berries
- Cheese
- Chicken
- Composed
- Extras
- Fish
- Flour

- Fruit
- Green
- Meat
- Milk product
- Pasta
- Pork
- Potatoes
- Salty
- Seafood
- Spice
- Spicy

Most of the ingredient tags is self-explanatory, and what ingredients goes under this category. Composed and extras can be more difficult to understand and is described below:

The **composed** tag is ingredients that are made up from different other ingredients. Pizza sauce is typically a composed ingredient due to the composition crushed tomatoes and different types of spices. Skillful chefs may use few of these ingredients.

Extras can be referred to as ingredients that are used as side dishes. Pasta and corncob are associated with this ingredient tag.

C.2 Recipe Tags

All recipes are associated with at least one recipe tags. Below is the list of all recipe tags used in the set of recipes:

- Family

- Fast
- Healthy
- Comfort food
- Guests
- Asian food
- Italian food
- Mexican food
- Indian food
- Norwegian food

C.3 Full Recipe Description

Listed below is three recipes used in the example in Section 3.5.

C.3.1 Paprika Chicken with Asparagus

This recipe contains the following ingredients for one portion:

- Salt (0.5 tablespoons)
- Pepper (0.5 tablespoons)
- Green fresh asparagus (4 pieces)
- Sour cream (1 tablespoon)
- Chicken filet (1 pieces)
- Corn flakes (2 tablespoons)
- Paprika powder (1 tablespoon)

These ingredients resolves to the following ingredient tags:

- Salt → Spicy, Salty, Spice
- Pepper → Spice
- Green fresh asparagus → Green, Extras
- Sour cream → Milk product
- Chicken filet → Chicken, Animal product
- Corn flakes → Composed
- Paprika powder → Spice

C.3.2 Babi Asam Manis

This recipe contains the following ingredients for one portion:

- Onion (0.25 pieces)
- Garlic (0.75 cloves)
- Pork into strips (150 grams)
- Chili powder (0.25 teaspoon)
- Canned pineapple chunks (0.25 can)
- Paprika (0.5 pieces)
- Ready to cook wok sauce (0.25 bag)
- Pineapple juice (0.5 deciliter)

These ingredients resolves to the following ingredient tags:

- Onion → Green
- Garlic → Green, Spice

- Pork into strips → Pork, Meat
- Chili powder → Spice, Spicy
- Canned pineapple chunks → Composed, green
- Paprika → Green
- Ready to cook wok sauce → Composed
- Pineapple juice → Fruit

C.3.3 Asian Omelet

This recipe contains the following ingredients for one portion:

- Egg (1.5 pieces)
- Oyster sauce (0.5 tablespoons)
- Beef meat in strips (100 grams)
- Bean sprouts (0.25 can)
- Oyster mushrooms (50 grams)

These ingredients resolves to the following ingredient tags:

- Egg → Animal product
- Oyster sauce → -
- Beef meat in strips → Beef
- Bean sprouts → Extras
- Oyster mushrooms → Green

Bibliography

- [1] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59, 1994.
- [2] G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless networks*, 3(5):421–433, 1997.
- [3] G. Adomavicius, R. Sankaranarayanan, S. Sen, and A. Tuzhilin. Incorporating contextual information in recommender systems using a multidimensional approach. *ACM Transactions on Information Systems (TOIS)*, 23(1):103–145, 2005.
- [4] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005.
- [5] G. Adomavicius and A. Tuzhilin. Context-aware recommender systems. In *Recommender systems handbook*, pages 217–253. Springer, 2011.
- [6] D. W. Aha. Case-based learning algorithms. In *Proceedings of the 1991 DARPA Case-Based Reasoning Workshop*, volume 1, pages 147–158, 1991.
- [7] X. Amatriain, J. M. Pujol, and N. Oliver. I like it... i like it not: Evaluating user ratings noise in recommender systems. In *User modeling, adaptation, and personalization*, pages 247–258. Springer, 2009.
- [8] M. J. Berry and G. Linoff. *Data mining techniques: for marketing, sales, and customer support*. John Wiley & Sons, Inc., 1997.

- [9] J. Bettman, M. Luce, and J. Payne. Consumer decision making: A constructive perspective. *Consumer Behavior and Decision Making*, pages 1–42, 1991.
- [10] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is “nearest neighbor” meaningful? In *Database Theory—ICDT’99*, pages 217–235. Springer, 1999.
- [11] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 43–52. Morgan Kaufmann Publishers Inc., 1998.
- [12] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware applications: from the laboratory to the marketplace. *Personal Communications, IEEE*, 4(5):58–64, 1997.
- [13] R. Burke. Hybrid web recommender systems. In *The adaptive web*, pages 377–408. Springer, 2007.
- [14] P. G. Campos, I. Fernández-Tobías, I. Cantador, and F. Díez. Context-aware movie recommendations: An empirical comparison of pre-filtering, post-filtering and contextual modeling approaches. In *E-Commerce and Web Technologies*, pages 137–149. Springer, 2013.
- [15] F. Cena, L. Console, C. Gena, A. Goy, G. Levi, S. Modeo, and I. Torre. Integrating heterogeneous adaptation techniques to build a flexible and usable mobile tourist guide. *AI Communications*, 19(4):369–384, 2006.
- [16] D. Cosley, S. K. Lam, I. Albert, J. A. Konstan, and J. Riedl. Is seeing believing?: how recommender system interfaces affect users’ opinions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 585–592. ACM, 2003.
- [17] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.
- [18] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

- [19] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction*, 16(2):97–166, 2001.
- [20] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.
- [21] J. Freyne and S. Berkovsky. Intelligent food planning: personalized recipe recommendation. In *Proceedings of the 15th international conference on Intelligent user interfaces*, pages 321–324. ACM, 2010.
- [22] T. J. Gerpott, S. Thomas, and M. Weichert. Characteristics and mobile internet use intensity of consumers with different types of advanced handsets: An exploratory empirical study of iphone, android and other web-enabled mobile users in germany. *Telecommunications Policy*, 37(4):357–371, 2013.
- [23] K. J. Hammond. Chef: A model of case-based planning. In *AAAI*, pages 267–271, 1986.
- [24] T. R. Hinrichs and J. L. Kolodner. The roles of adaptation in case-based design. In *AAAI*, volume 91, pages 28–33, 1991.
- [25] M. Hosseini-Pozveh, M. Nematbakhsh, and N. Movahhedinia. A multi-dimensional approach for context-aware recommendation in mobile commerce. *arXiv preprint arXiv:0908.0982*, 2009.
- [26] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 263–272. IEEE, 2008.
- [27] D. Kelly and J. Teevan. Implicit feedback for inferring user preference: a bibliography. In *ACM SIGIR Forum*, volume 37, pages 18–28. ACM, 2003.
- [28] H.-N. Kim, A.-T. Ji, I. Ha, and G.-S. Jo. Collaborative filtering based on collaborative tagging for enhancing the quality of recommendation. *Electronic Commerce Research and Applications*, 9(1):73–83, 2010.
- [29] J. Kolodner. *Case-based reasoning*. Morgan Kaufmann, 1988.

- [30] G. Linden, B. Smith, and J. York. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.
- [31] R. Lopez De Mantaras, D. McSherry, D. Bridge, D. Leake, B. Smyth, S. Craw, B. Faltings, M. L. Maher, M. T. Cox, K. Forbus, et al. Retrieval, reuse, revision and retention in case-based reasoning. *The Knowledge Engineering Review*, 20(03):215–240, 2005.
- [32] F. Lorenzi and F. Ricci. Case-based recommender systems: a unifying view. In *Intelligent Techniques for Web Personalization*, pages 89–113. Springer, 2005.
- [33] D. A. Lussier and R. W. Olshavsky. Task complexity and contingent processing in brand choice. *Journal of Consumer Research*, pages 154–165, 1979.
- [34] P. Massa and B. Bhattacharjee. Using trust in recommender systems: an experimental analysis. In *Trust Management*, pages 221–235. Springer, 2004.
- [35] K. Oku, S. Nakajima, J. Miyazaki, and S. Uemura. Context-aware svm for context-dependent information recommendation. In *Mobile Data Management, 2006. MDM 2006. 7th International Conference on*, pages 109–109. IEEE, 2006.
- [36] C. Palmisano, A. Tuzhilin, and M. Gorgoglione. Using context to improve predictive modeling of customers in personalization applications. *Knowledge and Data Engineering, IEEE Transactions on*, 20(11):1535–1549, 2008.
- [37] U. Panniello, A. Tuzhilin, M. Gorgoglione, C. Palmisano, and A. Pedone. Experimental comparison of pre-vs. post-filtering approaches in context-aware recommender systems. In *Proceedings of the third ACM conference on Recommender systems*, pages 265–268. ACM, 2009.
- [38] S.-T. Park, D. Pennock, O. Madani, N. Good, and D. DeCoste. Naïve filterbots for robust cold-start recommendations. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 699–705. ACM, 2006.

- [39] M. J. Pazzani and D. Billsus. Content-based recommendation systems. In *The adaptive web*, pages 325–341. Springer, 2007.
- [40] D. Poo, B. Chng, and J.-M. Goh. A hybrid approach for user profiling. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 9–pp. IEEE, 2003.
- [41] C. Prahalad. Beyond crm: Ck prahalad predicts customer context is the next big thing. *American Management Association MwWorld*, 2004.
- [42] A. M. Rashid, I. Albert, D. Cosley, S. K. Lam, S. M. McNee, J. A. Konstan, and J. Riedl. Getting to know you: learning new user preferences in recommender systems. In *Proceedings of the 7th international conference on Intelligent user interfaces*, pages 127–134. ACM, 2002.
- [43] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994.
- [44] F. Ricci, D. R. Fesenmaier, N. Mirzadeh, H. Rumetshofer, E. Schaumlechner, A. Venturini, K. W. Wöber, and A. H. Zins. Dietorecs: a case-based travel advisory system. *Destination recommendation systems: behavioural foundations and applications*, pages 227–239, 2006.
- [45] C. K. Riesbeck and R. C. Schank. *Inside case-based reasoning*. Psychology Press, 2013.
- [46] N. S. Ryan, J. Pascoe, and D. R. Morse. Enhanced reality fieldwork: the context-aware archaeological assistant. In *Computer applications in archaeology*. Tempus Reparatum, 1998.
- [47] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 253–260. ACM, 2002.
- [48] B. N. Schilit and M. M. Theimer. Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, 1994.

- [49] M. Van Setten, S. Pokraev, and J. Koolwaaij. Context-aware recommendations in the mobile tourist application compass. In *Adaptive hypermedia and adaptive web-based systems*, pages 235–244. Springer, 2004.
- [50] Y. Wang, S. C.-f. Chan, and G. Ngai. Applicability of demographic recommender system to tourist attractions: A case study on trip advisor. In *Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology-Volume 03*, pages 97–101. IEEE Computer Society, 2012.
- [51] Z. Yu, X. Zhou, D. Zhang, C.-Y. Chin, X. Wang, et al. Supporting context-aware media recommendations for smart phones. *Pervasive Computing, IEEE*, 5(3):68–75, 2006.