NTNU
Norwegian University of
Science and Technology

A Hybrid Recommender System for Context-Aware Recommendations of Restaurants

Matias Pettersen
Adrian Kristoffer Tvete

Master of Science in Computer Science
Submission date: June 2016
Supervisor: Herindrasana Ramampiaro, IDI

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

Problem Description

The primary aim of the thesis is to design, implement and evaluate a personalized recommender system in the touristic domain. In order to do this, it is necessary to get an overview of existing work done in the field of recommendation systems, both in general and within the touristic domain.

Assignment given: January 18th, 2016
Supervisor: Heri Ramampiaro, IDI

Preface

This Master's thesis is written by Kristoffer Tvete and Matias Pettersen from January 2016 to June 2016 at the Norwegian University of Science and Technology (NTNU). The thesis completed our Master of Science (MSc) degree in Computer Science, with specialization in Intelligent Systems.

We would like to thank our supervisor, Heri Ramampiaro, for constructive feedback and discussions, and for always being in a pleasant mood throughout the semester. We would also like to thank our friends and families for their encouragement and support throughout our five years at NTNU.

Trondheim, June 22, 2016

Kristoffer Tvete

Kristoffer Tvete

Matias Pettersen

Matias Pettersen

Abstract

Recommender systems have cemented themselves in the daily online activities of most people, and they have been successfully applied across a range of different domains. However, they have yet to make the big breakthrough in complex areas such as tourism and gastronomy. A reason for this is that the attributes of items within these domains are seldom readily quantifiable, and people's opinions on items like hotels and restaurants are dependent on a large number of factors. A successful recommender system in such a complex domain could have a great impact on further advancements of this technology.

In this thesis, we present *RestRec*: a novel, personalized, context-aware, hybrid recommender system for restaurants. We perform a literature review showing that even though contextual information has huge potential, it has been largely ignored in research. In order to learn more about what factors affect people's choices in restaurants, and the nature of their social settings when attending them, we conduct a survey. The subsequent results are then incorporated into our system with the purpose of improving recommendations. Furthermore, we address the cold-start user problem which pertains to the difficulty of providing high quality recommendations to new users where little information exists. This problem is particularly dominant in the restaurant domain where the majority of users are cold-start users. To address the effects of a cold start, we employ a combination of collaborative filtering with demographic information and content-based filtering.

To evaluate *RestRec* we perform user-based evaluation conducted on friends, family, and fellow students. Our experiments show that we are indeed successful in identifying patterns with regards to social setting and use this to make better recommendations. The overall predictive accuracy of the system exceeds 70 %, showing the feasibility of our approach.

Sammendrag

Anbefalingssystemer har etablert seg i de daglige nettaktivitetene til folk flest og har blitt innført med suksess i en rekke domener. Likevel har de fortsatt til gode å gjøre det store gjennombruddet i komplekse områder, som for eksempel turisme og gastronomi. En årsak til dette er at egenskapene til elementer innenfor disse domenene sjelden er lett kvantifiserbare, og folks oppfatninger av hoteller og restauranter er avhengige av en rekke faktorer. Et vellykket anbefalingssystem i et så komplekst domene kan ha betydning for videre fremskritt for denne type system.

I denne masteroppgaven presenterer vi *RestRec*: et personlig, kontekstbevisst anbefalingssystem for restauranter. Vi utfører en litteræranalyse som viser at selv om kontekstuell informasjon har stort potensial, har det i stor grad blitt ignorert innen forskningen. For å identifisere hvilke faktorer som spiller inn i folks valg av restauranter og deres sosiale setting under besøket, utfører vi en spørreundersøkelse. Resultatene fra denne undersøkelsen blir brukt til å forbedre anbefalingene våre. Videre prøver vi å takle kaldstart-problemet som er knyttet til utfordringen med å tilby gode anbefalinger til nye brukere som man vet lite om. Dette problemet er spesielt fremtredende i restaurant-domenet hvor de fleste av brukerne er kaldstart-brukere. For å takle problemet, bruker vi en kombinasjon av samarbeidsfiltrering med demografisk informasjon og innholdsbasert filtrering.

For å evaluere *RestRec* utfører vi brukertester på venner, familie, og medstudenter. Eksperimentene våre viser at vi lykkes med å identifisere mønstre med hensyn til sosial setting, og at vi klarer å bruke dette til å lage bedre anbefalinger. Systemet har en treffsikkerhet på over 70 %, noe som viser anvendbarheten av vår metode.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

In this chapter we give an introduction to some of the basic premises that the rest of this thesis builds upon. We provide motivation for the work to be done, followed by a presentation of the problem description along with research goals, scope, and limitations.

1.1 Motivation

Since the Internet saw the light of day, it has become an integral part of our society and how we lead our lives. Statistics presented by the International Telecommunication Union (2015) show that in the period 1997 to 2014, the number of worldwide Internet users has increased from 2 % to 40 % (11 % and 78 % in the developed world). This, added with the commercialization of the Internet and the digitalization of our society, results in what we today refer to as Big Data — *large pools of data that can be captured, communicated, aggregated, stored, and analyzed* (McKinsey & Company, 2011).

As we spend more and more of both our professional and private lives connected to the Internet, the amount of available data is exploding. According to Mervis (2012), 1.2 zettabytes of new data is generated each year, and recent studies suggest that we use as much as 6 hours online every day via PCs, laptops, smartphones, and tablets (Mander, 2015). Companies all over the world are collecting vast amounts of data on their customers in the hope of being able to improve their user experiences and ultimately increase their revenues. One would perhaps expect this to be beneficial to everyone, as companies will be able to target their customers more accurately, and users will have access to more information. However, this is not the whole truth.

Indeed, in the era of Big Data, it is becoming increasingly difficult for the regular Internet-user to successfully navigate the sea of information on the World Wide Web effectively. For example, a user browsing an online movie service does not wish to go through tens, or perhaps hundreds, of uninteresting movies before finding an acceptable one. The overwhelming amount of options makes it harder for the user to find exactly the wanted item, a state termed *information overload* (Adomavicius & Tuzhilin, 2005) (see Figure 1.1). This is already a reality in several domains, for example multimedia and tourism. In order for actors in these areas to be successful and competitive, they need their users to be satisfied with their services. Fueled by this need to combat information overload, recent years have witnessed an increase of research in the area of recommender systems (Resnick & Varian, 1997).

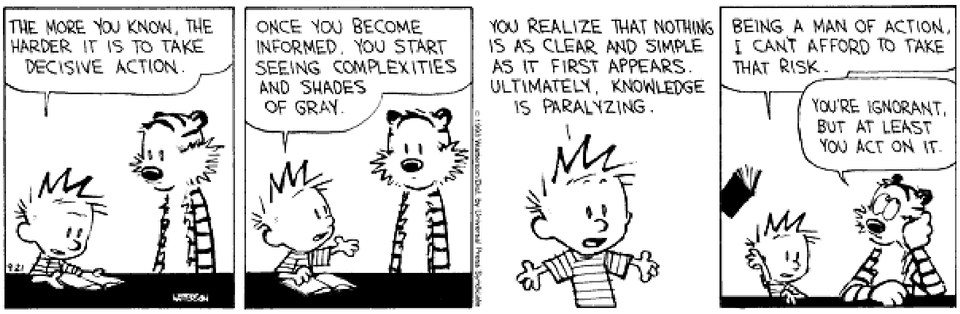

THE MORE YOU KNOW, THE HARDER IT IS TO TAKE DECISIVE ACTION.
ONCE YOU BECOME INFORMED, YOU START SEEING COMPLEXITIES AND SHADES OF GRAY.
YOU REALIZE THAT NOTHING IS AS CLEAR AND SIMPLE AS IT FIRST APPEARS. ULTIMATELY, KNOWLEDGE IS PARALYZING.
BEING A MAN OF ACTION, I CAN'T AFFORD TO TAKE THAT RISK.
YOU'RE IGNORANT, BUT AT LEAST YOU ACT ON IT.

Figure 1.1: Calvin and Hobbes' take on information overload (Watterson, 2005).

Recommender systems are software tools that aim at helping their users to make the best choice within a certain domain. Some recommender systems are personalized, meaning that different users or user groups receive diverse recommendations, while others are non-personalized. An example of the former is the system employed in the online bookstore Amazon[1], where each user sees a personalized store based on their interests (Linden, Smith, & York, 2003). Non-personalized recommendations are much simpler to generate and typical examples include top ten selections of books and CD's.

In the field of tourism most people know sites like Expedia[2], Hotels.com[3], and TripAdvisor[4]. These sites all include some form of recommender system to present the user with popular products and good deals, but currently they do not provide the user with truly personalized recommendations. Rather than seeing the user as an individual with its own interests, they assume that all users are similar. A reason for this is the inherent complexity of items within this domain. People's opinions on items such as hotels and restaurants are dependent on a large number of factors, making it hard to offer good personal recommendations.

Some reasons on why established recommendation techniques cannot be directly applied to the tourism domain are mentioned by Felfernig, Gordea, Jannach, Teppan, and Zanker (2006). Collaborative filtering is a widely used approach in recommender systems, but works best when there exists a broad user community and each user has already rated a significant number of items. Given the fact that travel planning activities are noticeably less common than for example buying a book or watching a movie, and the items themselves may have a far more complex structure, it is difficult to establish reasonable user profiles. In addition, a single trip arrangement may consist of several, independently configurable services. Typically only pre-defined packages are available online, for example "all-inclusive" or "flight and hotel". There is also the concept of *context* (Anand

[1]Amazon: www.amazon.com
[2]Expedia: www.expedia.com
[3]Hotels.com: www.hotels.com
[4]TripAdvisor: www.tripadvisor.com

& Mobasher, 2007), or what situation the user is in when requesting recommendations. Users interact with systems within a particular context, and ratings for an item in one context may be completely different in another. Consider for example a male user going to a sports bar with a group of friends versus going with a date. It is likely that the user would rate the bar higher in the first scenario.

A successful recommender system in such a complex domain could make a big difference in the way we plan vacations and reduce the amount of resources needed.

1.2 Problem description

Based on the information provided in the previous section, we make the following observation: *Recommender systems have been successfully introduced in many systems across a range of domains, but they have yet to make a breakthrough in complex areas such as tourism and gastronomy.*

People travel more than ever and according to a survey done by Gesellschaft für Konsumforschung (2014), around 90 % of travel bookings today involve going online. When in a new country or location people tend to make use of the Internet for tips on what to do or where to go, and recommender systems play a huge part in this process. In view of this, we can see that a working recommender system in tourism could have a significant impact on the industry. Thus, we formulate the main research question for this thesis as follows:

> *How can we build an intelligent, personalized recommender system that works in the touristic domain by making use of established recommendation techniques?*

To reach a conclusion regarding this main question, it is helpful to first split it into several smaller problems that will be addressed. The idea is that the individual solutions to each of these subquestions can be put together to form a complete solution to the main question.

RQ1: What challenges are there, and what methods have been developed to meet them?

RQ2: What kind of systems already exist, and what are their strengths and shortcomings?

RQ3: How can we find and use data to describe the items to be recommended?

RQ4: How can we identify the user's situation and use it to improve recommendations?

1.3 Scope and limitations

Tourism is a big and complex domain, and due to the time constraints for this work, we have to narrow the scope down to a more manageable size. Going out for a meal at a restaurant is very common when being a tourist, thus we consider restaurant recommendation to be a proper subset of touristic recommendation. Choosing a restaurant to eat at, whether it is an important dinner with business associates, or simply a casual dinner with friends and family, is a familiar situation for most people. Sometimes the amount of variables to consider can be quite large and the decision-making process can quickly lay claim to valuable time, time perhaps better spent on other affairs.

We stress that even though restaurant recommendation is a small part of the tourism domain, we do not believe it to be an oversimplification of the original problem statement. Restaurants are complex items requiring complex user profiles, and it is not something the average person uses every day. The problems with touristic recommendation mentioned in Section 1.1 still very much apply, and just like people travelling more than ever, the culture of going out for food is increasing day by day. This has contributed to the fact that restaurant recommendation is currently an important research field.

The focus of the work described in this thesis is on the recommendation of restaurants, and how to handle a cold start system. A cold start is the term used to describe a system in its start-phase where it is difficult to make recommendations due to lack of data. Furthermore, we examine the possibility of including contextual information in restaurant recommendations. We survey the field of recommender systems and how they are applied with regards to restaurants. Finally, we design and implement our own system called *RestRec*, and we gather data with the purpose of testing the system. The results of this process are analyzed and explained.

For simplicity, only restaurants in New York City are considered. The data collected for the testing phase is provided by persons in the local community and mostly students. As a consequence of this we will not be able to test the system for various demographic groups. Text analysis and sentiment analysis of for example textual reviews will not be done.

1.4 Thesis structure

The rest of this thesis is organized as follows:

In Chapter 2 we give an introduction to the field of recommender systems and the theory that lies beneath. The cold start problem is introduced, and much of what is written here will be brought up in later chapters.

Chapter 3 is a study of related work and current state-of-the-art within recommender systems. Projects, research, and solutions that are of importance to the work described in this thesis are presented and explained.

Chapter 4 introduces the *RestRec* system. Architecture and communication between the different components are expounded, design and technological choices are justified, and the data domain is explained.

In Chapter 5 we evaluate the system by verifying that it provides sound recommendations. The data is examined and various statistics are shown and explained. Furthermore, we discuss our findings in relation to the research questions described in Chapter 1.

Finally, Chapter 6 forms the conclusion of the work described in this thesis, along with suggestions for possible future work.

Chapter 2

Background

This chapter will go into more detail on how recommender systems work. We begin in Section 2.1 by presenting some of the reasons as to why one would want to implement these systems. Sections 2.2 and 2.3 introduce similarity models and classification. These are core concepts and vital to any working recommender system. In Sections 2.4 and 2.5 we describe how feedback from the users and contextual information can be used to improve future recommendations. Next, in Section 2.6, we introduce some of the challenges related to recommender systems. Finally, in Section 2.7, we present various methods of evaluating recommender systems.

2.1 Recommender systems

Recommender Systems (RSs) are software tools for providing a user with suggestions on how to solve a specific problem (Lops, de Gemmis, & Semeraro, 2011). They help users deal with information overload, and have become an important field of research since they were introduced in the early 90s (Goldberg, Nichols, Oki, & Terry, 1992). A few examples of what RSs can help us with are to decide what music to listen to, movies to watch, news articles to read, items to buy, and what to eat. An item in the context of RSs refer to what the system recommends to users.

Most RSs focus on the recommendation of items within a specific domain and their implementations are geared and customized towards being as useful as possible for those items. They are mainly directed towards users that are not familiar with or have insufficient knowledge about the type of items being recommended to make an informed decision.

In addition to being useful for consumers, Lops et al. (2011) list several reasons as to why service providers may want to implement these systems:

- *Increase sales*. This is perhaps the most obvious reason. When the system gets to know the user, it can recommend more specific items, thus increasing the chance of the user buying.

- *Increase diversity*. The system can recommend items that the user has not previously looked into, but is likely to be interested in.
- *Increase satisfaction*. A well designed system can affect the user experience by providing relevant and interesting suggestions.
- *Better understand what customers want*. The service provider may decide to use the acquired user information for other purposes, like improving the management of the production. For instance, in the travel domain the management can advertise a specific region to a new group of users.

When implementing an RS, there are a few different options to choose from with regards to exactly how the recommendations are made. Most of today's systems employ content-based filtering or collaborative filtering, or even a mixture of the two, resulting in what we call *hybrid systems*.

2.1.1 Content-based filtering

In content-based filtering (CBF) both item and user profile are described by a set of attributes, and the recommended items are ranked based on how similar they are to the user's attribute profile (Lops et al., 2011). This type of RS is very popular due to its simplicity, and it can be used to recommend items such as websites, music, movies, books, restaurants and hotels.

The steps one has to consider when building a content-based system are the following (assuming the designer has access the data needed):

1. Building the item profile of a set of attributes describing the items to be recommended.
2. Building the user profile using the same set of attributes.
3. Devising a measure to compute the similarity between user and item.
4. Ranking the items with respect to similarity, and provide them to the user.
5. Devising a method for refining the user profile based on feedback.

Due to this method being based solely on content, it has several desirable properties such as good scalability, that is the system works well independent of how many users there are in the system. Furthermore, it works well for predicting items that are new to the system, as the only thing needed is to calculate the similarity between the item and the user profile. This presupposes that the item profile is already built. CBF is easy to understand and the user can easily comprehend why a specific item is recommended, making the system more trustworthy to the user.

On the other hand, there are some drawbacks. As a consequence of the recommendations being based exclusively on the user's profile, they may become overspecialized. For example if a user has liked several movies of the same movie series, the system will lean towards recommending the rest of the movies from that series and nothing else. This is an example where there is too little variation in the output of the system, and the recommendations become boring. Another drawback is the difficulty of building the item profile. To extract the features of an item and get all the different aspects can be a demanding task. For example when constructing an item profile for a website it is difficult to represent the user experience or the aesthetic details, which are essential attributes to consider when rating websites.

2.1.2 Collaborative filtering

Collaborative filtering (CF) is fundamentally different from CBF (Balabanović & Shoham, 1997). Instead of creating recommendations based on the similarity between user and item, they recommend items that users of similar preferences have rated favorably in the past. There are three steps to this process of recommending:

1. Users are clustered based on their user profiles.
2. A ranked list of the most popular items in the cluster is made.
3. The top item that the user has not seen before is recommended to the user.

The main advantage of CF is that it does not need to know anything about the items in order to make recommendations. Recommendations are based exclusively on the similarities between users, meaning that it is possible to recommend very complex items.

However, the lack of content information can be a disadvantage. Consider for example a travel website recommender where one user likes Expedia and another user likes TripAdvisor. The system will not cluster those users in the same group even though Expedia and TripAdvisor are very similar items. A pure CF system will only group users if they have rated some of the same items. Another disadvantage with this method is if there is a user with a rare taste of items compared to the other users in the system. This will result in recommending unsuitable items for that user since it is badly clustered. Lastly, there is the problem of introducing new items in the database. There is no way a new item can be recommended to a user until another user has rated it. This is known as the cold start problem which we will explain in more detail later.

2.1.3 Hybrid solutions

Pure CF is able to solve all of the problems related to CBF. By making use of other users' ratings, we can deal with any kind of items and recommend items dissimilar to those seen in the past. Similarly, by doing content analysis we can deal with the problems specific to CF mentioned earlier.

This brings us to the concept of hybrid RSs, systems that combine multiple techniques together to achieve some synergy between them (Burke, 2002). It is for example very popular to combine collaborative and content-based techniques as they complement each other very well. A system like this can base the recommendations on both the individual user's profile, and groups of users with similar rating history. In special cases, the system can even swap between collaborative recommending and content-based recommending. If there are only one user in the system it can use content-based methods, and when the RS fails to extract distinguishing features from items, it falls back on collaborative filtering. Burke (2002) has identified seven types of hybridization methods:

- *Weighted* - Outputs from the chosen techniques (in the form of scores or votes) are combined with different degrees of importance to offer final recommendation.
- *Switching* - The type of situation affect which technique the system uses.
- *Mixed* - Recommendations from several techniques are presented simultaneously to the user.
- *Feature combination* - Features from different recommendation sources are combined as input to a single technique.
- *Feature augmentation* - The output from one technique is used as an input feature to another.
- *Cascade* - One recommender refines the recommendations given by an other.
- *Meta-level* - The model learned by one recommender is used as input to an other.

2.1.4 Other methods

Two other techniques of creating RSs worth mentioning are demographic systems and knowledge-based systems. They are not as widely used as CF or CBF, but they are still useful and are more often than not included in hybrid RSs.

Demographic systems

A demographic RS rely on demographic information about users, such as gender, age, geographical location, marital status, occupation and income, etc (Krulwich, 1997). Given this input, it is possible to calculate the set of demographic clusters to which the user are most likely to belong, and have the data available for the resulting clusters serve as the basis for the user profile. Recommendations are then done for each cluster, like in CF.

Knowledge-based systems

Knowledge-based RSs use the features of the items and knowledge about how these meet the user's needs to try to match an item to the the user's need (Burke, 2000). This knowledge will sometimes contain explicit functional knowledge about how certain product features meet user needs. For example when a user buys a camera in an online store, the system knows that the user may be interested in a camera bag as well.

2.2 Similarity models

In order for systems who employ CF or CBF to work, it is necessary to be able to calculate various similarities between users and items. In information retrieval systems, similarity models are used to find relevant documents based on a query. RSs carry out the same action, except with more variations in relation to similarity models. A query is not restricted to being either a user profile or an item, and the same goes for the documents. The selected approach depends on the purpose and implementation of the specific system. For example in collaborative systems, we may be interested in finding users that are similar to a specific user. And in content-based systems, we may want to find items that are similar to another item.

Consider U the set of users where $u \in U$, and D the set of items where $d \in D$. The similarities we may be interested in are $sim(u, d)$, $sim(u_i, u_j)$, and $sim(d_i, d_j)$. There are a variety of ways to calculate these, but the most popular approaches are Euclidean distance, Pearson correlation, and cosine similarity (Adomavicius & Tuzhilin, 2005).

Euclidean distance This is the simplest way of calculating similarity between two data points. It sums up the distance between every pair of attributes in the data x and y. In Equation 2.1, n is the dimensionality of the data points.

$$sim(x,y) = \sqrt{\sum_{k=1}^{n} (x_k - y_k)^2} \tag{2.1}$$

Pearson correlation Given the covariance $\Sigma(x,y)$ of data points x and y, and their standard deviations σ_x and σ_y, we compute the Pearson correlation as:

$$sim(x,y) = \frac{\Sigma(x,y)}{\sigma_x \times \sigma_y} \tag{2.2}$$

The Pearson correlation is a measure of the linear correlation between two variables, giving a value in the range of [-1,1] where 1 is total correlation and -1 is total negative correlation.

Cosine similarity In the cosine-based approach (Equation. 2.3) the two items x and y are treated as two vectors in n-dimensional space. The similarity between x and y can then be measured by calculating the cosine of the angle between them.

$$sim(x,y) = \frac{x \cdot y}{||x|| ||y||} \tag{2.3}$$

The cosine similarity also gives a value in the range [-1,1] and can be interpreted the same way as the Pearson correlation in Equation 2.2.

Weighting Traditionally, these are the most used methods in RSs, but there are many different variations on how to weight the terms. Consider a system where we want to predict the rating $r_{c,s}$ for user c and item s. This value is usually computed as an aggregate of the ratings of some other group of users, usually the N most similar. The simplest is

$$r_{c,s} = \frac{1}{N} \sum_{\hat{c} \in C} r_{\hat{c},s} \tag{2.4}$$

where *C* is the set of users that have rated item *s*. Intuitively, this equation translates into ”the predicted rating *r* of user *c* for item *s* is the average rating of some set of other users for item *s*”.

However, Equation 2.4 has an obvious limitation: it places the same amount of importance on every user, something that is not always desirable. We can account for this by including a similarity measure:

$$r_{c,s} = k \sum_{\hat{c} \in C} sim(c, \hat{c}) \times r_{\hat{c},s} \tag{2.5}$$

One problem with using the weighted sum in Equation 2.5 is that users may be using the rating scale differently. The *adjusted* weighted sum

$$r_{c,s} = \overline{r}_c + k \sum_{\hat{c} \in C} sim(c, \hat{c}) \times (r_{\hat{c},s} - \overline{r}_{\hat{c}}) \tag{2.6}$$

tackles this by making use of the user’s deviation from his or hers average rating, instead of using the absolute values of ratings. In both Equation 2.5 and Equation 2.6 *k* serves as a normalizing factor.

2.3 Classification

Classification is a machine learning technique to predict group membership for data instances. In other words, it is a mapping between feature space and label space. Without these algorithms we would not be able to predict whether a user would like a certain item or not, and are consequently a key component to RSs.

Classification has two distinct meanings. We may be given a set of observations with the aim of establishing the existence of classes or clusters in the data. Or we may know for certain that there are so many classes, and the aim is to establish a rule whereby we can classify a new observation into one of the existing classes. The former type is known as *unsupervised* learning (or clustering) (Jain, Murty, & Flynn, 1999), the latter as *supervised* learning (Michie, Spiegelhalter, & Taylor, 1994). For example in a content-based RS, a supervised method can be used to calculate an estimate of the probability that a user will like an item. This probability can then be used to sort a list of recommendations. In a collaborative system, clustering can be used to form groups of users with similar preferences.

The following section will go through a few popular classification algorithms, both supervised and unsupervised.

2.3.1 Naive Bayes

Bayesian classifiers are supervised and based on the definition of conditional probability and Bayes theorem (Duda & Hart, 1973). Given a record *R* with *N* features, the goal is to predict the class *C* by finding the value *C* that maximizes the probability of the class given the data. When applying Bayes theorem, we get

$$P(C|R_1, R_2, ..., R_N) = P(R_1, R_2, ..., R_N|C)P(C) \tag{2.7}$$

The naive Bayes classifier makes strong independence assumptions between the features. For example, a fruit may be considered to be a banana if it is yellow, bent, and approximately 20 cm long. The classifier considers each of these features to contribute independently to the fact that the fruit is indeed a banana, regardless of any correlations between the features (thus the name *naive* Bayes). This assumption lets us rewrite the conditional probability as

$$P(R_1, R_2, ..., R_N|C) = P(R_1|C)P(R_2|C)...P(R_N|C) \tag{2.8}$$

Naive Bayes is frequently used due to being fast and easy to implement, even though it places last or near last in many head-to-head classification papers (Rennie, Shih, Teevan, & Karger, 2003). The biggest drawback of naive Bayes is the assumption that features are independent, and in many domains this is simply not true. For example in text classification, a word is not independent of the word in front of it.

2.3.2 Support vector machines

Support Vector Machines (SVM) are supervised learning models that analyze and recognize patterns (Hearst, Dumais, Osman, Platt, & Scholkopf, 1998). Unlike naive Bayes, SVM is a non-probabilistic classifier. Given a set of training data belonging to one of two categories, the SVM builds a model that assigns new input to one category or the other. In other words, it is a binary classifier.

In its simplest, linear form, an SVM is a hyperplane that separates a set of positive examples from a set of negative examples with maximum margin. See Figure 2.1 for an illustration of the concept. The formula for the output is

$$u = \vec{w} \cdot \vec{x} - b \tag{2.9}$$

where $\vec{w}$ is the normal vector to the hyperplane and $\vec{x}$ is the input features vector. We then want to maximize the following margin, *m*:

$$m = \frac{1}{||\vec{w}||} \tag{2.10}$$

SVM have increased in popularity in recent years and they have been shown empirically to perform well on problems such as handwritten character recognition (Cortes & Vapnik, 1995), face detection (Osuna, Freund, & Girosit, 1997), and text categorization (Joachims, 1998).

2.3.3 k-Nearest neighbor

This algorithm works by memorizing the training data and using them to predict the label of unseen cases. Given a point to be classified, *k*NN finds the *k* closest cases from the training set and assigns it the predominant label amongst the neighbors (Cover & Hart, 1967) (see Figure 2.2). Closeness can for example be calculated by use of similarity models as shown in Section 2.2. The *k*NN classifier is amongst the simplest of all machine learning algorithms, and the most challenging part is how to determine the value of *k*. If it

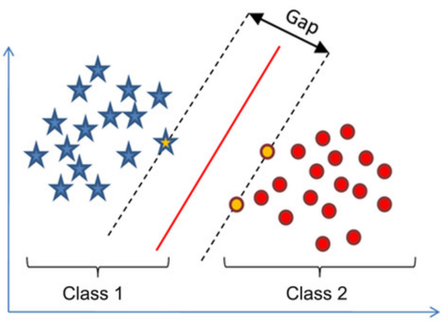


Figure 2.1: Illustration of SVM ("Illustration of SVM," n.d.).

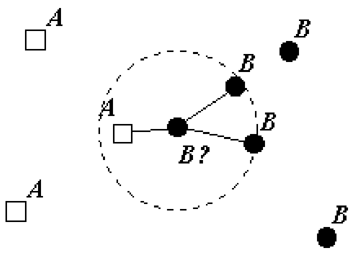


Figure 2.2: Illustration of *k*NN with *k*=3. As 2 out of the 3 closest cases are labeled B, the new case is labeled B ("Illustration of KNN," n.d.).

is too low the classifier will be sensitive to noise points, but if it is too big the neighborhood might include too many points from other classes. *k*NN is what we call a *lazy learner*, meaning that it does not construct a model on which to base future classifications. Every time a new point is to be classified it has to calculate the *k* nearest neighbors, implying that *k*NN does not scale well. This algorithm is frequently used in CF due to its simplicity and conceptual relation to CF.

2.3.4 k-Means

k-Means is an unsupervised classification algorithm, meaning that it tries to assign items to a group such that items within a group is more similar than items in another group (Jain et al., 1999). Simply put, the algorithm aims to partition *n* observations into *k* clusters in which each observation belong to the cluster with the nearest mean.

k-Means is an iterative algorithm. It begins by randomly selecting *k* centroids which is to be the centers of the clusters, and then assigns each item to the centroid that is closest to it. In every step it tries to minimize the distance from each item to its corresponding centroid as shown in Equation 2.11. This process is continued until a convergence criterion is met, for example until no item changes cluster or the squared error within each cluster is sufficiently low.

$$E = \sum_{k} \sum_{n \in S} d(n, k) \tag{2.11}$$

The k-Means algorithm is popular because it is easy to implement, and its time complexity is *O(n)*, where *n* is the number of items. Clustering is sure to increase efficiency of the RS, but it is unlikely to help improve accuracy and it has several shortcomings. For instance, the user needs to select an appropriate *k*, indicating the need for some prior knowledge of the data. It is also very sensitive to the initial placings of the centroids. An example of a result of clustering is shown in Figure 2.3.

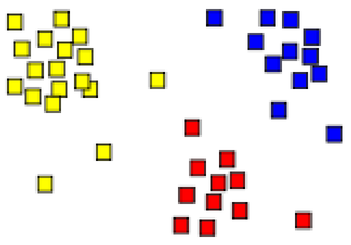


Figure 2.3: The result of a cluster analysis shown as the coloring of the squares into three clusters ("The result of a cluster analysis," n.d.).

2.4 Feedback

Feedback is an important part of any good RS. In order to make useful and reliable recommendations to the user, the system must learn to know the user. This is achieved by observing and storing how the user interacts with the system and the items. All feedback can be stored in the recommender database and used for generating new recommendations in the next session.

There are two different techniques for recording a user's feedback: explicit feedback and implicit feedback (Lops et al., 2011). Furthermore, it is possible to divide between two kinds of feedback: positive feedback (inferring features liked by the user) and negative feedback (inferring features the user does not like). Most systems implement both of these methods. Table 2.1 shows a few examples of common types of feedback.

2.4.1 Explicit feedback

When a system requires the user to explicitly evaluate items, it makes use of what we call explicit feedback. There are mainly three different kinds of explicit feedback:

Table 2.1: Examples of explicit and implicit feedback.

Explicit feedback	**Implicit feedback**
Rating	Dwell time
Like/dislike	Bookmarking
Review	Mouse input/keyboard input

- *Binary* - items are classified by a binary scale, as either relevant or non-relevant. A well-known example of this is the Facebook[1] like/dislike button.
- *Ratings* - a discrete numeric scale is used to rank items. Examples of this can be to have the user rate items on a scale from one to 10, roll a dice, or give an amount of "stars". Alternatively, the numeric scale may represent symbolic ratings. In questionnaires, each response have a numeric value attached to them.
- *Text comments* - text comments are widely used amongst e-commerce providers. They enable users to write textual comments about items, thus helping new users in the decision-making process. They contain much information, but are hard for an automated system to process. Advanced sentiment analysis software is needed to analyse whether the comment is positive or negative towards the item, and to what degree.

These methods are easy to implement, but explicit feedback generally place an increased cognitive load on the user, and may not even be able to capture the user's feelings about an item. Another catch is that most users do not bother with providing explicit feedback unless they have to, making it hard for the system to provide accurate recommendations.

2.4.2 Implicit feedback

Implicit feedback does not require any active user involvement. The way the user interacts with the system reveals much about the user's preferences, and the system can take advantage of this. Implicit feedback works by assigning a score to different user actions on an item, such as saving/deleting the item, reading time, and mouse movements. Purchase history, browsing history, and search patterns are also important types of implicit feedback. For example if a user listens to many songs by the same artist, the user probably likes that artist.

This kind of feedback is more prone to noise because the users are generally unaware of the process taking place, but over time the system learns more and more about the user's true preferences.

2.5 Contextual recommendation

Lieberman and Selker (2000) define context as ”everything that affects the computation except its explicit input and output”. Should for example a travel RS provide the same

[1]Facebook: www.facebook.com

recommendations in both winter and summer? And should a restaurant RS recommend the same restaurant to both groups and single individuals?

Context has been largely ignored in research into RSs (Anand & Mobasher, 2007). Most existing approaches focus on recommending the most relevant items to users without taking into account any additional contextual information, such as time, location, or the company of other people (Lops et al., 2011). They ignore the notion of ”situated actions”, the fact that the user may interact with the system within a particular context and ratings within one context may be completely different from ratings within another context.

Consider a user that buys and rates books of science fiction for himself, work-related books on computer science topics, and books for his children. Combining this user's interest in books into a single representation that aggregates all of these genres is clearly a bad idea, yet this is what most systems will do. Any children's books the user may have bought should not have an impact on recommendations of computer related books. The ideal contextual RS would therefore be able to reliably label each user action with a context.

Context-aware RSs (CARS) try to address these problems. They model and predict user tastes and preferences by incorporating available contextual information into the recommendation process as explicit additional categories of data. This contextual information can be obtained in a number of ways, including:

- *Explicitly* - The user explicitly provides contextual data by answering direct questions or eliciting this information through other means. For example, a website may obtain contextual information by asking the user to fill out a form.

- *Implicitly* - Context is obtained implicitly from the data or the environment. For example, the location of the user can be obtained through the GPS device in the mobile phone, or temporal information can be obtained from the timestamp of a transaction. The advantage of this approach is that nothing needs to be done in these cases in terms of interacting with the user.

- *Inferred* - Contextual information can be inferred by use of statistical or data mining methods. For example, the household identity of a person flipping through the TV channels (husband, wife, son, etc.) may not be explicitly known to a cable TV company, but it can be inferred by observing the TV programs watched and the channels visited by use of data mining methods.

2.6 Challenges

As with any other research field, there are many and various challenges to consider when building an RS. Some have been there from the start, while others are emerging as the technology improves. In this section we will introduce two of the more prominent challenges, namely that of cold start and scalability. We also mention two emerging challenges: privacy and proactive recommendations.

2.6.1 Cold start

A big problem for RSs is the so-called cold start problem, and it pertains to the sparsity of information (Lika, Kolomvatsos, & Hadjiefthymiades, 2014) about users and items. To adapt to a user, the system needs to know what the user liked in the past. However, when a new user joins the system, nothing is known about the user, thus it is not possible to make recommendations.

Systems based on CF are built on community preferences, such as ratings. Consequently, if an item does not have any ratings it will never be recommended to anyone. This problem occurs mostly when the user population is small compared to the item base or it is a new item in the system. It is also problematic to do proper clustering when the amount of ratings in general are low, and bad clustering leads to bad recommendations.

RS designers tend to solve this problem by either getting users to rate items at the start, or by getting them to answer some demographic questions (and then using stereotypes as a starting point, for example elderly people like classical music).

2.6.2 Scalability

The biggest problem for scaling an RS is the amount of operations involved in computing distances. One possible way to solve this is by use of clustering algorithms, or we can reduce the dimensionality of the data.

It is common in RSs to have datasets that are both high-dimensional and sparse (Lops et al., 2011), so it is essential to be able to reduce the dimensionality of the data. This is necessary for clustering algorithms because of noise in the features, and the computational complexity associated with high dimensionality. It is also helpful to reduce the dimensions when the feature vectors contain a limited number of values for each object and there is much density between each value. Most of the values will be zero when for example describing the interest relation between an item and a user because most users have just made up their mind about a very small portion of all the objects.

Two popular dimensionality reduction methods are Principal Component Analysis (PCA) (Jolliffe, 2002) and Singular Value Decomposition (SVD) (Golub & Reinsch, 1970). These methods map users and items into a dense and reduced latent space that captures their most prominent features. They provide better recommendations than traditional neighborhood methods (Herlocker, Konstan, Borchers, & Riedl, 1999) as they reduce the level of sparsity and improve scalability (Koren, 2008).

Principal Component Analysis

This is a classical statistical method to find patterns in high dimensionality datasets, such as the ones used in RSs. The method points out the covariance structure of the set of variables in the data and identifies the principal directions in which the data varies. This allows us to obtain an ordered list of components that account for the largest amount of variance in the data. The dimensions can then be reduced by neglecting those components with a small contribution to the variance.

The variance is computed by finding the eigenvectors and the corresponding eigenvalues to each dimension. The dimensions with the highest eigenvalues are the dimensions

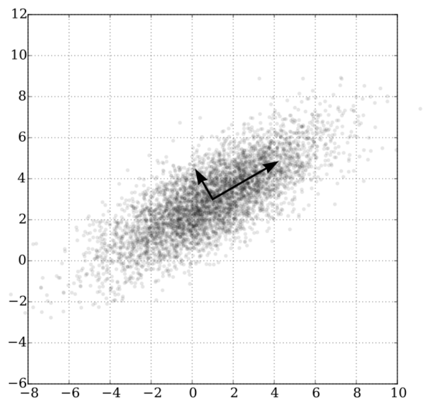


Figure 2.4: PCA applied to a Gaussian distribution. The two vectors are the principal components of the data (“PCA applied to a Gaussian distribution,” n.d.).

with most variance and thus are most important (see Figure 2.4).

The positive aspect with PCA is that it is powerful and can be used on very large datasets. On the other hand, non-linear structures are hard to model with PCA, and the original data also needs to be drawn from a Gaussian distribution. When this assumption does not hold true, there is no warranty that the principal components are meaningful.

Singular Value Decomposition

This method is very similar to PCA in the way that both methods seek to find the principal components. SVD, however, uses a slightly different approach. Instead of using the eigenvalues to compute the variance in each dimension, it uses the singular value for each dimension. The variances are given by squaring the singular values. The singular values can be seen as the square roots of the eigenvalues of the ”squared matrix” AA^T.

The advantage of this method is that it computes optimal dimension reduction. There are also incremental algorithms to compute an approximated decomposition. In that way, when new users or ratings arise in the system, it does not need to compute the decomposition from scratch. The drawback is that it is computationally hard and is sensitive to outliers.

2.6.3 Emerging challenges

Most RSs developed so far follow a ”pull” model, meaning that the user has to explicitly request recommendations. However, in the modern society where computers are ubiquitous and smartphones are everywhere, it seems natural to imagine that an RS should be able to detect implicit requests, resulting in *proactive* RSs (Yeung & Yang, 2010). The challenge then consists of not only predicting what to recommend, but also when to recommend it.

Another challenge is that of *privacy*. Privacy and security are increasing concerns regarding RSs (Lam, Frankowski, & Riedl, 2006). The very fundament of the technology is based on knowing as much as possible about the users. In the attempt to increase the quality of these systems, they collect as much user data as possible. This will clearly have a negative impact on the privacy of the users, and they may start to feel that the system knows too much about them. Therefore, it is important to address this issue in the research community. There is need for systems that sensibly use user data, while ensuring that malicious users cannot get their hands on private data.

2.7 Evaluation

How to evaluate an RS is extensively studied in the literature (Herlocker, Konstan, Terveen, & Riedl, 2004). It is essential to know the performance of the system after it is implemented in order to find out if the work has been successful or not. There are many properties that can be considered when evaluating an RS, and in this section we introduce the most common evaluation methods. We list and explain the principal properties of an RS.

2.7.1 Offline evaluation

This type of evaluation is done by using pre-collected data of user behavior, for example user ratings. It is assumed that the user behavior after the system is deployed is the same as in the pre-collected dataset. Experiments with this dataset are easy to carry out because it does not require any interaction with any user and we can then evaluate an RS with very low cost.

The negative factor of this is that the dataset can be too narrow and not see all the aspects of the system as a whole. This type of evaluation will in most cases just calculate the prediction score of the system. With that in mind, one can see that this method is fit to quickly determine if a system performs good or not with a low cost. In order to do these evaluations it is necessary to simulate the behavior a user has online where the user gives feedback on recommendations.

2.7.2 Online evaluation

Online evaluation, also known as A/B testing, tries to measure the change in user behavior when the user interacts with different RSs using different settings. It is then easier to evaluate the system against the others since they are tested in the same conditions. In online evaluation the system is used by real users performing real tasks. This method can be used to see how the user behavior is affected by the different property changes done in the evaluating process.

Many real world systems employ an online testing system where multiple algorithms can be compared. Typically, such systems redirect a small percentage of the traffic to different alternative recommendation engines, and record the users interactions with the different systems.

2.7.3 User studies

A user study is conducted by making test subjects perform various tasks requiring an interaction with the RS. The subjects will be asked questions about their experiences during the tasks, and quantitative measurements will be collected. With the questions asked the system can be evaluated on properties that are difficult to measure, such as user experience.

Unlike offline evaluation methods, user studies give the opportunity to observe the user in action. In that way, this method does not need to do any assumptions about user behavior. It will also be collected qualitative information about the behavior of the user, in difference to the two other methods. The drawback of this method is that it is very expensive to conduct. To get a reliable evaluation many subjects need to do several tasks repeatedly, resulting in a costly affair.

2.7.4 The principal properties of Recommender Systems

There are a range of properties to consider when evaluating an RS. As different RSs try to meet different needs, they have to be built with focus on different properties, some of which are trade-offs for one another. For example accuracy is at trade-off when the system wants to focus on diversity.

User preference This property is the most basic one. If a user prefer one system over another, it can be said that the preferred system is better. This property does not need any measurements and it is easy for a test subject to have an opinion.

Prediction accuracy There are different types of prediction accuracy. In ratings prediction accuracy, each item in the catalog is given a predicted rating. After the user has given these items their ”true” scores, the accuracy can be measured based on how similar the predicted rating was to the true rating. The most common metric used for this is the Root Mean Squared Error (RMSE).

Usage prediction measures how many of the recommendations that are successful. This can be done by computing the precision and recall of the recommendations. In order to elaborate on precision and recall, some terms need to be explained:

- *True Positives (TP).* The items recommended to a user that the user classifies as interesting.
- *True Negatives (TN).* The items not recommended to a user that the user classifies as not interesting.
- *False Positives (FP).* The items recommended to a user that the user classifies as not interesting.
- *False Negatives (FN).* The items not recommended to a user that the user classifies as interesting.

The simplest measure of accuracy is the ratio between the correctly predicted instances and the total number of instances as defined by Equation 2.12.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.12}$$

To get a more informative measure of accuracy, precision and recall is used. Precision is defined in Equation 2.13 and is the measure of how many of the predicted items that were successful.

$$P = \frac{TP}{TP + FP} \tag{2.13}$$

Recall is a measure of how good the system is to recommend the items that the user is interested in. It is defined in Equation 2.14.

$$R = \frac{TP}{TP + FN} \tag{2.14}$$

Coverage This term can have several meanings, but the most common is the Item Space Coverage. The simplest measure of this is the percentage of all items that may be predicted by the system. Another one is the User Space Coverage. This is the percentage of the users that the system can recommend items to.

Serendipity This is a measure of how surprising the successful recommendations are. It is important for the user to have good prediction accuracy, but it must not come at the cost of novelty. Serendipity is a property that favor the predictions that can seem a bit surprising. This property needs to be be balanced with the accuracy of the system. If a system has good serendipity, it is more likely to successfully recommend more diverse items to the user.

Diversity This is the opposite of similarity. It is not very useful for a user to only get recommended similar items all the time. The property is often measured by using item-item similarity, and diversity comes at expense of for example accuracy. When the diversity goes up, the system is less likely to recommend items the user is fond of, thus decreasing accuracy. The cold start problem is highly correlated to the diversity property of an RS.

Robustness Robustness can be seen as the ability of the system to handle fake information. Considering how many people rely on RSs, it can be tempting to influence the system to predict items that comes one to favor. For example a rating of an item can be increased by making several fake users rate the item favorably. It is, however, unrealistic to create a system that is immune to these type of attacks. Robustness can also be seen as the stability of the system under extreme conditions, such as many requests to the system at the same time.

Adaptivity This property describes how well the system adapts to quick changes in the item set and on trends in the interest of these items. Another type of adaptivity is how fast the user preferences adapt to new user ratings. If a user history only contains comedy movies and the user suddenly rate an action movie high, the user preferences is expected to change.

Scalability It is essential for an RS to work well with large datasets. Ideally, the algorithms would perform well on both small and large sets of items, but even though the algorithms work well on a small dataset it does not necessarily mean they are free of flaws on a big one. Often the RSs trade properties such as accuracy or coverage, to get rapid results on large datasets. Dimensionality reduction will lead to faster computation of recommendations on very large datasets while not sacrificing too much accuracy.

The scalability of an RS can be measured by observing the speed and resource consumption when testing the system on different sized datasets.

Chapter 3

Related work

In this chapter, previous work and research into the area of RSs will be presented in more detail in order to place the work of this thesis in relation to existing solutions and to find comparable results. We start by introducing a set of existing solutions in Section 3.1, before we in Section 3.2 present research on the cold start problem, and how external effects can impact recommendations. Section 3.3 is a conclusion where we compare the introduced systems to each other and state their differences.

3.1 Related systems

There exist many systems that seek to provide a user with recommendations about restaurants or tourism in general. As we cannot hope to cover them all, we have chosen a set of five systems that we will explain in more detail. The systems are called *R-Cube*, *I'm feeling LoCo*, *REJA*, *OpenTable*, and *TripBuilder*. We chose these systems to show some of the various approaches that are possible and to give a broad idea of existing solutions in both academia and industry.

3.1.1 R-cube

R-cube is a hybrid dialogue system for restaurant recommendation and reservation (Kim & Banchs, 2014). The authors opted for a dialogue system, as this kind of systems are gaining popularity nowadays through applications like SIRI[1], Google Now[2], and Watson[3]. The three main factors contributing to this trend are the increased quality of speech recognition systems, the increasing availability of data for supporting data-driven applications and technologies, and the ubiquitous accessibility to information and services from mobile platforms.

[1]Apple Siri: http://www.apple.com/ios/siri/
[2]Google Now: https://www.google.com/landing/now/
[3]IBM Watson: http://www.ibm.com/smarterplanet/us/en/ibmwatson/

The proposed system, *R-cube*, is composed by a combination of three subsystems: a restaurant recommendation sub-system, in which the system collects information about the user's preferences in order to shortlist restaurant options; a restaurant selection sub-system, in which the user can ask questions about the shortlisted venues with the objective of making a final selection; and a booking sub-system, in which the system collects the required additional information to complete the restaurant booking process.

The restaurant recommendation sub-system considers the user's explicit preferences through five variables: type of food, price range, area of city, and restaurant-name. The system will keep asking the user questions about these variables until it is able to reduce the list of restaurant candidates to four or less. At this point the system moves on to the next phase: the restaurant selection sub-system. The user now has the possibility to learn more about the recommended restaurants through variables such as address, phone number, general description, reviews, etc. When the user has selected the desired restaurant, the booking system takes over. In a process equal to the one in the first phase, the user is asked to provide information about number of guests, booking date, and sitting time. Once this is done, the reservation is completed and the user receives a booking confirmation.

Each of the sub-systems consists of a sequence of four process levels (see Figure 3.1): preprocessing, natural language understanding (NLU), dialogue management (DM), and natural language generation (NLG). To increase robustness of the system, each level has multiple components based on different approaches. Even though results obtained from different components at the same level might differ, they are all provided as input for the next level. In this way, the damages caused by erroneous output from one component can be diminished.

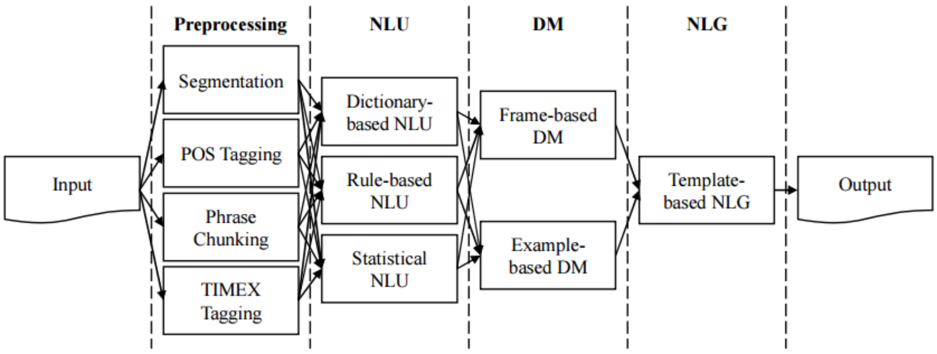


Figure 3.1: The system architecture for each sub-system in *R-cube*. Illustration by Kim and Banchs (2014).

To show that the system does indeed work as intended, a transcription of a real interaction between the system and a given user is included towards the end of the paper. For future work, the authors wish to integrate additional components into the DM and NLG levels.

3.1.2 I’m feeling LoCo

I’m feeling LoCo is a location based context-aware RS described by Saiph Savage, Baranski, Elva Chavez, and Höllerer (2012). Research on RSs has paid little attention to the integration of contextual information, and the systems that do, often require the user to complete extensive surveys. The motivation for creating this system springs from earlier research suggesting that when responding to long questionnaires, individuals are more likely to give identical answers to most or all of the questions. It is clear that a system which could automatically and accurately infer an individual’s preferences would dramatically boost the user experience.

I’m feeling LoCo is implemented as a mobile application, and aims at presenting more complete recommendations by considering temporal and spatial information. Instead of including a survey phase, the system mines a person’s Foursquare[4] profile and maps this data into user preferences. Furthermore, the application automatically infers a user’s current mode of transportation and utilizes this information to determine an upper bound for how far a person would be willing to travel to visit a location. The only explicit input the user has to provide is what type of venue they are searching for.

To detect the form of transportation (stationary, walking, biking, or driving), the system employs a decision tree followed by a first-order Hidden Markov Model (HMM). The HMM helps reducing noise by utilizing temporal knowledge of the previous transportation mode that was detected. The system will for example not misclassify situations where the user is driving and slowed down due to traffic or red lights.

User preferences are acquired through Foursquare. The Foursquare application makes use of the user’s GPS coordinates and returns a list of possible places the user could be in. Each item on the list has an associated name, category, and relevant tags that describe the place. The user simply selects the place they are currently in, and the application will upload this to the user’s profile along with the associated tags. The Foursquare API then permits the retrieval of all these user check-ins with associated data. This retrieved information is what constitutes the user model in *I’m feeling LoCo*. Every time a user visits a place, its name, category, and tags are added to the user model, which in essence is a document holding a series of words.

Due to the sparse nature of the data and the wish to be able to recommend places that have not been visited by other users, a CBF recommendation algorithm is used. The recommendation process can be explained as follows:

1. The user selects a category for the recommendations. Example categories are ”artsy”, ”nerdy”, and ”hungry”.

2. Given the current location, all places within a certain radius of the user is retrieved. The radius depends on the user’s mode of transport.

3. The list of places is filtered to only keep those of the selected category, and associated tags for each of the remaining places are obtained.

4. For each place, a set of words containing the intersection between the tags of the user and the tags of the particular place is created.

[4]Foursquare: https://foursquare.com/

5. The log frequency weight is calculated for each term in a set of words. This is a measure of how many times a certain term is mentioned in the user's profile and is calculated for every set of words.

6. A summation over all the weights for every place is done. This summation represents the score of a particular place.

7. The *K* places with highest scores are recommended to the user.

In cases where the user has not provided enough check-ins through Foursquare (cold start), the authors have included a metric that searches the current city's wikitravel page for iconic places or landmarks. These places are then searched for on Foursquare to retrieve category and address. If the landmark is close to the user and the requested category, it is suggested to the user.

The system is tested with user studies (see Section 2.7.3) in different US cities (Portland, Beaverton, Santa Barbara, and Goleta). All users made positive comments about the system's ability to flawlessly detect their current transportation mode. Furthermore, the authors noted that the Foursquare usage of all participants increased through the testing period. They believe the users felt motivated to check-in to more places they visited in order to obtain more precise recommendations.

3.1.3 REJA

REJA (REstaurants of JAén, Spain) is a restaurant RS that hybridize a CF and a knowledge-based system (Martinez, Rodriguez, & Espinilla, 2009). The main advantage of *REJA* is the use of incomplete preference relations in the knowledge-based system in order to overcome the cold-start problem of the collaborative component.

The collaborative system used by *REJA* is implemented by using the CF engine *CoFE* in combination with a database of restaurants and users. It seems the *CoFE* project is no longer maintained as we were not able to find any information about it, thus we cannot say anything certain about the inner workings of the CF algorithm.

When it comes to dealing with cold start, the authors reach the same conclusion as Saiph Savage et al. (2012), that requiring the user to explicitly provide information about their preferences is unlikely to be popular. But unlike Saiph Savage et al. (2012), however, Martinez et al. (2009) decide to use a knowledge-based approach that require just a minimal amount of information in order to provide suitable recommendations. It works by means of a case-based reasoning method as follows:

1. The user selects a restaurant similar to their needs.

2. The system provides three well known restaurants and requires the user to rate them such that the system can compare them to the selected restaurant.

3. With these three pieces of data the system can create a simple user profile that will be used to create recommendations.

REJA also provides the user with geographical information via Google Maps as it helps avoiding textual data overloading. With this, the user can find the restaurant location on a map, search for the shortest path, and find popular tourist places (museums, parking spaces, etc).

3.1.4 OpenTable

OpenTable[5] is the world's leading provider of online restaurant reservations (Das, 2015). With the capability to recommend over 32 000 restaurants worldwide, they seat more than 17 million diners each month. In addition to the diner-restaurant interaction history they use click and search data, the metadata of restaurants, as well as insights gleaned from reviews, together with any contextual information to make meaningful recommendations.

Many of the restaurants have thousands of reviews. By making use of Word2vec[6], these reviews are used to create a vector space with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words sharing common contexts in the corpus are located in close proximity to one another in the space. This enables *OpenTable* to learn for example what kind of wine goes with what food, and find synonyms for various words.

Expecting diner reviews to be broadly composed of a handful of themes (such as food, drinks, ambiance, service, etc.), they also use the reviews for topic modeling (see Figure 3.2). This helps reveal the unique aspects of each restaurant without having to read the reviews. In other words, it is possible to identify the top topics for any restaurant.

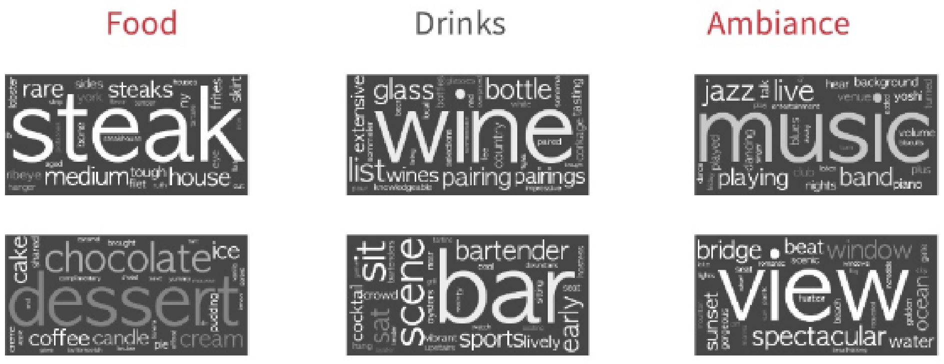


Figure 3.2: Topic modeling at OpenTable.

3.1.5 TripBuilder

TripBuilder takes as input the target travel destination, the time available for the visit, the user's profile, and builds a personalized tour of various Points of Interest (PoIs) (Brilhante, Macedo, Nardini, Perego, & Renso, 2013). It takes into account both the time needed to enjoy an attraction, and the time needed to get to the next when calculating the route.

[5]OpenTable: www.opentable.com
[6]Word2vec: http://deeplearning4j.org/word2vec

The knowledge is mined entirely in an unsupervised way from two publicly available collaborative services: Wikipedia[7] and Flickr[8]. Flickr is used to gather photos and metadata from users all over the world, and Wikipedia is used to gather information on PoIs in a specific city.

More specifically, given the geographic location of a city, all Wikipedia pages related to an entity within the area is downloaded and considered a PoI. Description, geographic coordinates, and the set of categories relevant to the PoI is retrieved and stored in a touristic database. Flickr is used to collect a set of users and metadata of pictures taken within the city for the given time period. The assumption is that photo albums made by Flickr users implicitly represent touristic itineraries within the city. A photo is associated with a PoI if it was taken within a distance of 100 meters of the PoI, and the time needed for visiting a PoI is assumed to be the time between the first and last picture a user took of the PoI. Finally, the popularity of each PoI is calculated as the number of distinct users that take at least one picture of the PoI.

Given the preference vectors for user and PoI, the user-PoI interest is defined as a combination of user-PoI similarity, and the popularity of the PoI. Similarity is calculated with the cosine formula.

With the collected data, it is possible to calculate a set of trajectories for the users. A trajectory is defined as the sequence of PoIs visited consecutively. They now formulate the *TripCover* problem: the problem of generating an optimal personalized itinerary given the tourist's preferences and time budget. Given a tourist u, PoIs P, trajectories S, and the user-PoI interest function Γ, find a subset of S that maximizes total user-PoI interest while still upholding the user's time budget (Equation 3.1).

$$maximize \sum_{i=1}^{|S|} \sum_{j=1}^{|P|} \Gamma(p_j, u) \tag{3.1}$$

This is an instance of the *Generalized Maximum Coverage* problem which is proved to be NP-hard, so a greedy approximation algorithm is used. The results are promising, showing that *TripBuilder* outperforms two strong baselines for all considered metrics.

3.2 Related research

RSs are becoming more and more common in e-commerce and there are much research dedicated to improve the technology. Every year ACM holds a RSs conference (RecSys[9]) with the purpose of bringing together the main international research groups and companies working on RSs. It has become the most important annual conference for the presentation and discussion of RSs research, and RecSys2016 even has a workshop on RSs in tourism[10].

In this section we present some research pertaining to the cold start problem, how external factors can have an influence on reviews, and we do a small case study of one of

[7]Wikipedia: www.wikipedia.org

[8]Flickr: www.flickr.com

[9]RecSys: https://recsys.acm.org/

[10]Tourism in RecSys2016: http://www.ec.tuwien.ac.at/rectour2016/

the more sophisticated RSs in existence today, namely the Netflix[11] RS.

3.2.1 Cold start

A paper from Princeton University in the U.S. propose a method that combines CF and CBF to overcome the cold start problem (Wang & Blei, 2011). They wanted to develop a machine learning algorithm for recommending scientific articles to users in an online community. Their algorithm uses two types of data: the libraries of articles of the users of the community, and the content of each article. The goal of the system is to both recommend older papers that are important to others in the community with similar article taste, and at the same time recommend new papers that would be relevant to the user.

When recommending articles from other users with similar taste, they use CF based on latent factor models. This method works well for recommending popular articles, but cannot be used to recommend papers that has not been read yet. To deal with this they use content-analysis based on probabilistic topic modelling, and can consequently recommend articles with similar content as the articles enjoyed by the user in the past. The topic modelling of the articles provide a topic representation of the items to discover the main themes in each article, and in turn helps the system make intelligent recommendations for articles before anyone has rated them.

These two methods are subsequently combined in a probabilistic model where the decision of which item to recommend is based on the conditional expectation of hidden variables. The expectation is equally influenced by the content from the articles and the libraries of all the users, but in cases where the item is new, the recommendations are based on the content.

This method deals with new items and the problems associated. However, it does not address the problems of new users. Stern, Herbrich, and Graepel (2009) propose to solve the challenge by using meta-data about each user to recommend items popular in a user's demographic group such as age, gender, and occupation.

3.2.2 Demographics, weather and online reviews

Research shows that pleasant weather improves mood and memory, and demography has been associated with people's spending time online. In the paper *Demographics, Weather and Online Reviews: A Study of Restaurant Recommendations*, Bakhshi, Kanuparthy, and Gilbert (2014) ask the question of whether these phenomena documented in psychology also affect large-scale online behavior. Could weather and local demographics of restaurants drive how we rate them online? This study is the first to look at external factors and how they affect online ratings.

By studying a large amount of restaurants, reviews, demographic data, and weather data spanning from the period of 2002-2011 they find the following:

- Restaurants that are marked online as "low-price" tend to get fewer reviews and lower ratings.

[11]Netflix: http://www.netflix.com

- Service related factors such as ”take-away” are strongly tied with the population density of the neighborhood.
- Restaurants located in areas with higher education levels are more likely to be reviewed, but it does not seem to affect ratings.
- There is a seasonal pattern among rating and reviews, and weather conditions are significantly associated with ratings.

These findings have implications for the design of recommendation systems by accounting for, and correcting bias, that is systematically related to demographics and weather.

3.2.3 Netflix case study

When writing about the state of the art in RSs, there is no way to bypass Netflix. Netflix is the world’s leading Internet television network with over 69 million members in over 60 countries (Netflix, 2015) and is presently in possession of the most well-known RS. Their users watch more than 100 million hours of TV series and movies every day. They can watch as much as they want, when they want, for a fixed amount monthly, uninterrupted by advertisements.

The key to Netflix’s success lies in their ability to predict what the user likes, as every two out of three hour of playtime in Netflix are from recommendations. In Netflix’s early days, they used information about gender and age to determine what movies to recommend. Now, they use for example what the user has watched before, searched for, how much time the user spends watching, and which type of device being used. The predictions from the Netflix recommender engine is not only based on user behavior, but also context (such as time of the day), title popularity, novelty, diversity, and how recent the item was released are important factors. With all these features, Netflix gets enormous amounts of data. Their goal is to use it to discover patterns they can use to recommend items to users. In recent time they have used these patterns to find out what users want that does not exist already. For example in 2012, Netflix bought a script for a series called House of Cards (Barnes, 2013). They had figured since there was a demand for political dramas, the actor Kevin Spacey, and the successful director David Fincher, they needed to create a series with all of those factors. Fortunately, the series have been a success.

There are many important elements to Netflix’s personalization. One of them is the awareness the user has about the system adapting to their tastes. This builds up trust to the system and the system gets more ratings, which again leads to better recommendations. The same is encouraged when the user receive explanations of why the user is predicted a given item. An example of this is shown in Figure 3.3

In 2006, Netflix announced the Netflix prize, a machine learning and data mining competition for movie rating prediction. In that context, they released a dataset of 100,000,000 movie ratings. The task was to improve the accuracy of their existing system called Cinematch by 10 %. The accuracy at that time had a Root Mean Squared Error (RMSE) of 0.9525. The winner would be rewarded with one million dollar. This led to an increase of research on rating prediction by minimizing the Mean-Squared Error. After a time, it also led to a lawsuit against Netflix, once somebody managed to de-anonymize their data.

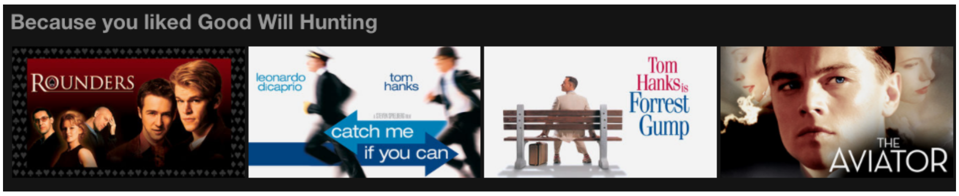


Figure 3.3: An example of the recommendations from Netflix.

A year into the competition a group won the Progress Prize with a score of 8.43 % improvement. The group reported that they had used over 2000 hours and the solution consisted of a combination of 107 algorithms. The two main algorithms of these was Matrix Factorization (which the community generally call SVD, Singular Value Decomposition) and Restricted Boltzmann Machines (RBM) which is a neural network.

The form of Matrix Factorization used is basically an asymmetric form of SVD, called SVD++, that can make use of implicit information (just as RBMs also do). With only the SVD version, the system performed with a RMSE of 0.8914 and with only RBM, the system performed with a 0.8990 RMSE. Combined in a linear blend, the algorithms had reduced the error down to 0.88. To implement these two algorithms Netflix had to make them deal with a higher amount of data and the ability to adapt to new ratings in the system everyday. Today, these two algorithms are an essential part of the Netflix recommendation engine. The algorithms developed by the winning team of the Netflix Prize 2009, three years after they started, were too computationally intensive to scale and the expected improvement after the upscaling was not worth the engineering effort.

3.3 Conclusion

In this chapter we have presented five different recommendation systems from the touristic domain, and we have showed that there are many different ways of creating recommendations (see Table 3.1). Even though they differ in approaches, they have many similarities when it comes to dealing with cold start and data.

R-Cube is in fact a very simple system considering the recommendations mechanism. Instead of calculating recommendations based on a user-profile, it asks the user to provide rules until it can filter out all but a small number of restaurants. However, this method pose challenges with scalability. As the underlying restaurant database grows, it will be increasingly difficult to filter out restaurants based on rules provided by the user. *R-Cube* does not consider any contextual information to address the scalability problem.

Both *I'm feeling LoCo* and *TripBuilder* make use of social media to overcome cold start and to build the user-profiles. This proves a very effective and user friendly approach due to the user's only task being to connect the system to their social media profiles. Depending on the chosen social media site and the items to recommend, it can be either hard or easy to extract information about the user's preferences. However, social media almost always provide some sort of demographic information about the user, and research has shown that this can improve the quality of recommendations. Out of the five systems, these two are the only ones using contextual information in their recommendations. An interesting aspect of *TripBuilder* is how it aims to provide recommendations consisting of several items. In Section 1.1 we mention exactly this as a challenge in touristic recommendations.

REJA takes a different approach in dealing with cold start users. Instead of connecting to social media or asking explicit questions to get personal preferences, it asks the user to select an item which is similar to their needs. This serves as a starting point for the system to circle in on the true user preferences.

Lastly, we have *OpenTable* which is a successful commercial system. As a consequence it is, naturally, hard to find details on their recommendations system. It is, however, interesting to see that it is in fact possible to make successful recommendations in the restaurant domain. As far as we know, *OpenTable* relies heavily on their user-generated data such as reviews. It does not look like they make use of any other contextual information, but based on the research presented by Bakhshi et al. (2014), they should look into incorporating weather-data in their RS.

Table 3.1: A comparison of the systems presented in this chapter.

System	**Recommendation method**	**Domain**
R-Cube	Dialogue	Restaurants
I’m feeling LoCo	Content-based	Locations
REJA	CF and knowledge-based	Restaurants
OpenTable	*Unknown*	Restaurants
TripBuilder	Collaborative-/content-based	Tours of locations

Chapter 4

Approach

In Chapter 1 we defined the main research question for this thesis as "*how can we build an intelligent, personalized recommender system that works in the touristic domain by making use of established recommendation techniques?*". Chapters 2 and 3 provided theory and related work, as well as motivation, towards building a solid foundation on which we can base our answer.

In this chapter we present *RestRec*, a context-aware, personalized, hybrid RS for restaurants. We start by defining a set of requirements in Section 4.1. Then, in Section 4.2, we show an abstract overview of our proposed design, along with an introduction to the data domain we will be working with and some challenges this domain poses to RSs. Sections 4.3 and 4.4 are dedicated to the implementation of the recommender engine and how feedback from the users is used to change their future recommendations. In Section 4.5 we explain the hybrid aspect of *RestRec* and how we use this to tackle the cold start problems. Section 4.6 describes how we use contextual information in *RestRec* to make better recommendations. Finally, in Section 4.7, we analyze the system with respect to the requirements laid out in Section 4.1.

4.1 Requirement Analysis

In any software engineering process, it is of utmost importance to create a Software Requirements Specification (SRS) before starting the development. A good SRS helps to lay the foundation and guidelines of the work to be done. An SRS following the standards as defined in IEEE specification 29148:2011 (IEEE Standards Association, 2011) lays out both functional and non-functional requirements, such as response time, availability, maintainability, etc. However, for the purpose of *RestRec* and this thesis, we limit ourselves to functional requirements only.

We present the system requirements for *RestRec* as two separate lists. The first contains all requirements that *must* be implemented in order to get a system that can help us answer the research questions listed in Section 1.2. The second contains requirements that *should* be implemented, but are not crucial to the outcome of this thesis and can be implemented at a later time. They are on a relatively high level and they do not specify any methods or techniques that can be used to achieve them.

1. Requirements that are crucial and *must* be implemented:
   (a) The system *must* allow new users to sign up and log in so that their personal information may be used to create more accurate recommendations in the future.
   (b) The system *must* allow authenticated users to log out of the system.
   (c) The system *must* allow authenticated users to set, and update, their personal information such as age, nationality and gender.
   (d) The system *must* provide a search mechanism to enable authenticated users to retrieve any specific restaurant.
   (e) The system *must* allow authenticated users to rate restaurants to indicate whether they would enjoy the restaurant or not.
   (f) The system *must* allow authenticated users to rate restaurants in relation to a given situation (context).
   (g) The system *must* allow authenticated users to improve their recommendations at any time by presenting a set of restaurants that the user can rate.
   (h) The system *must* provide authenticated users with a personalized list of top recommendations for a given context.
2. Requirements that are of lesser importance and can be implemented at a later time:
   (a) The system *should* allow unauthenticated users to search for restaurants.
   (b) The system *should* allow unauthenticated users to see a list of the best rated restaurants.
   (c) The system *should* be able to show authenticated users an overview of their rating history.
   (d) The system *should* provide an administrative interface where users with elevated privileges can see more detailed information about users in the system.
   (e) The system *should* be able to run on a mobile device.

Furthermore, as we are building a system that is going to handle personal data for a potentially large set of users, we have to focus on security. It is important that we can assure users that there are no risks of their information being stolen and used by malicious users.

4.2 System overview

In Figure 4.1 we show an abstract overview of how we intend the system to work. Simply put, it can bee seen as a combination of three components:

- *Restaurant data extractor*. This is where the system connects to an Internet endpoint in order to extract restaurant data that can be useful in the recommendation process. This component does not have a direct impact on recommendations, but we choose to include it in the figure as it may be desirable to update the local database at a fixed interval, as data can be both outdated and updated.
- *Recommendation engine*. The heart of the system. Given enough information on both users and data, it produces a list of recommendations to the active user. Section 4.3 will go into more detail about the inner workings of this module, so for now we consider this a black box.
- *Profile learner*. The entire goal of this system is to make personalized recommendations, which makes this a key component. Its function is to keep track of all the user profiles and update them whenever feedback is provided. We hope that, in time and given enough feedback, the user's digital profile will match the user's "true" profile as closely as possible. We will elaborate on this in Section 4.4.

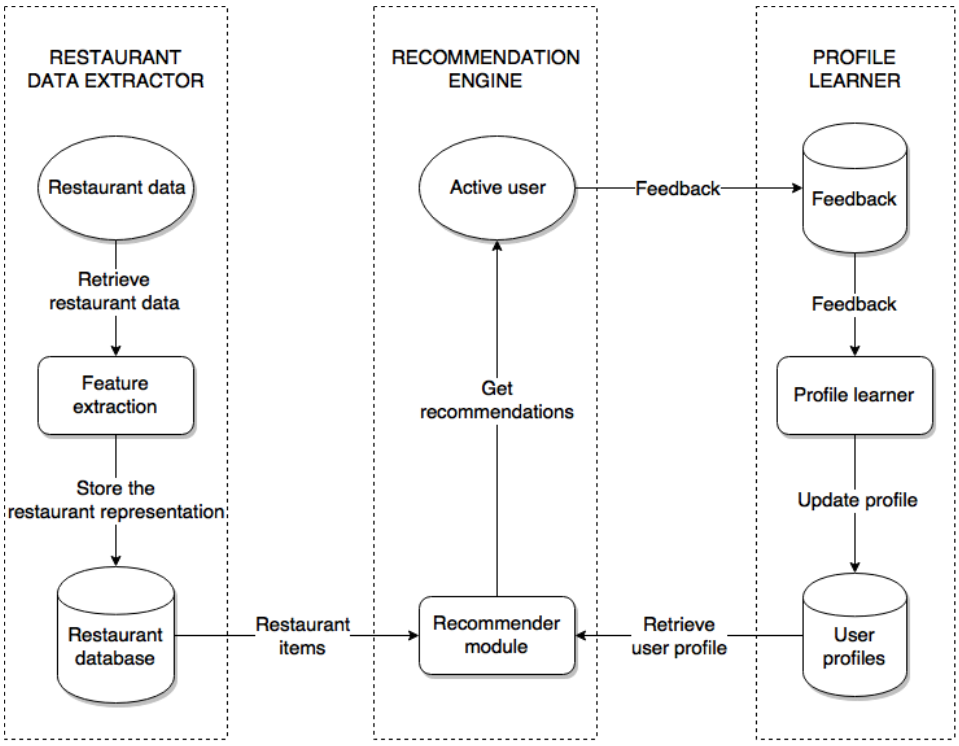


Figure 4.1: The architecture of the system, divided into three components.

From the perspective of a user, we envision a complete user session could for example unfold in the following way: A user will either have to register as a new user or provide authentication, depending on whether or not they are new in the system. The user will then get some recommendations based on his user profile. Finally, the user can provide feedback for the recommended restaurants.

4.2.1 Data domain

RestRec will be a *restaurant* RS. Ergo, we need to find suitable restaurant data. We aim for *RestRec* to be able to overcome the cold-start problems, and to be as self-sufficient as possible when it comes to user-generated content. Simply put, we aim to provide recommendations where no user-generated data exist. However, this will pose challenges. Many potential restaurant visitors will to some degree base their choices on information like quality of food, quality of service and cleanliness, which presupposes an existing user community.

One also has to consider the dynamic nature of restaurants, and how this sets the problem apart from for example movie-recommendation. It is not uncommon for restaurants to change their menus, renovate their locales, or get a new manager. These are changes that could possibly have a huge impact on the public opinion towards a restaurant. Even small changes, like getting a new chef, could affect online ratings. For a restaurant RS to be successful in the long run, it will have to be able to detect and adapt quickly to such changes.

In general, it is harder for restaurants to get consumer feedback compared to movies, and is a consequence of restaurants being more complex and the fact that more effort is required from the user to eat at a restaurant than watching a movie. Therefore, it is harder for restaurants to know whether they should be doing something different. When new movies get released to the public, the producers can easily compare how their movies are doing by looking at box office numbers. Restaurants do not have an equivalent ”portal” where they can compare themselves to others.

4.2.2 Technology

RestRec is a complete, web-based, personalized restaurant recommendation system that can be reached at http://www.restrec.no. The rest of this section is dedicated to clarifying the technical aspects of the system.

The website and its content are self-hosted on a laptop running Ubuntu Server 14.04 LTS[1] making use of the following stack: Django, MySQL/MongoDB, uWSGI, nginx.

Framework

Django[2] is a high-level Python[3] web framework that encourages rapid development of complex database-driven websites. It has good support for communicating with relational databases and takes care of many common security pitfalls behind the curtains, such as Cross Site Request Forgery and SQL injection.

[1]Ubuntu Server: http://www.ubuntu.com/download/server
[2]Django: https://www.djangoproject.com/
[3]Python: https://www.python.org/

Data storage

We are using MongoDB[4] for storage of restaurant data. MongoDB is a so called NoSQL/no-relational database, meaning that it avoids the traditional table-based relational database structure in favor of JSON-like documents. This kind of database is increasingly used in big data applications as they are easier to scale than classic relational databases.

For storage of user data, however, we are using MySQL[5] as it integrates nicely with Django.

Web server

Nginx[6] is a free, open-source, high performance HTTP/HTTPS webserver and reverse proxy. It can serve static files directly from the file system, and we use it as a port of entry to the *RestRec* application. However, Nginx cannot talk directly to Django applications. It needs something that will run the application, feed it requests from web clients and return responses. A Web Server Gateway Interface (WSGI) does this job, and we use a specific implementation called uWSGI[7].

Due to *RestRec* handling data such as usernames and passwords we decided to force all traffic through HTTPS, ensuring that all communication between client and server is encrypted.

4.3 Making recommendations

We have finally reached the point where it is natural to move more into the inner workings of the *recommendation engine* from Figure 4.1. Based on the theory provided in Chapter 2 and the related work in Chapter 3, we opted for a hybrid approach. A hybrid system is best equipped to tackle the challenges pertaining to a cold system, and it allows for more diversity in the recommendations. More specifically, we are building *RestRec* as a hybridization of CF and CBF.

The idea is simple: each component works independently from the other and produces a list of recommendations. These two lists are then merged by a *mixed* method as explained in Section 2.1.3. Figure 4.2 illustrates the concept.

However, before we can dive into the implementation of the recommendation engine, we need to explain how we represent users and restaurants in the system.

4.3.1 Representation

Both users and restaurants are represented by N-dimensional vectors where each element corresponds to one attribute. Boolean attributes are fairly uncomplicated, and each of them is represented by one element in the feature vector. Multivalued attributes on the other hand, have one element for each value in their domains.

[4]MongoDB: https://www.mongodb.com
[5]MySQL: https://www.mysql.com/
[6]Nginx: https://www.nginx.com/
[7]uWSGI: https://uwsgi-docs.readthedocs.io/en/latest/

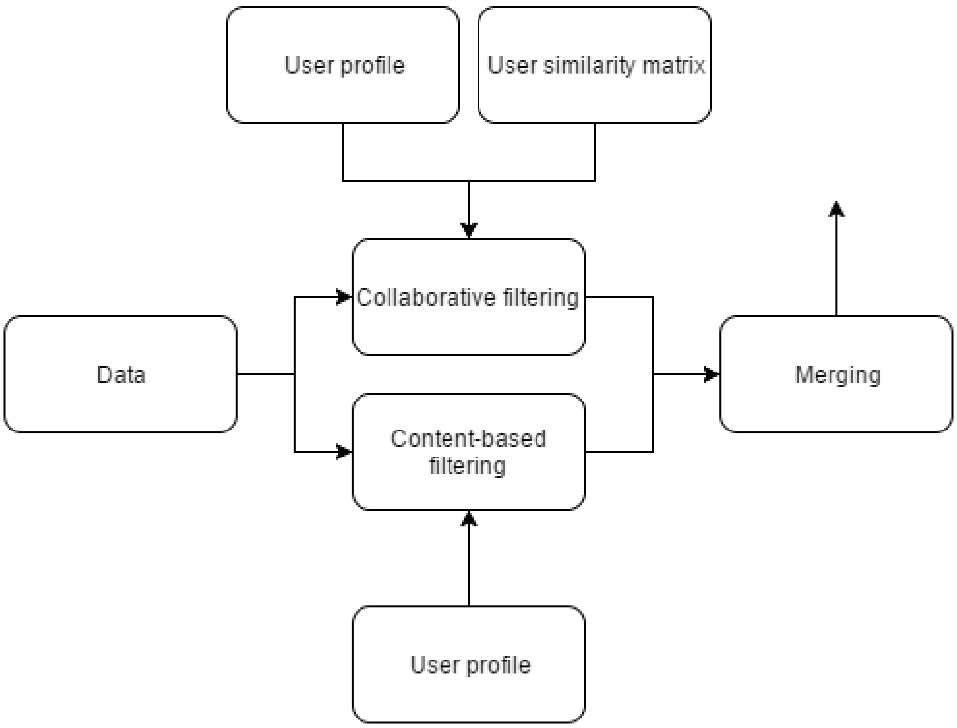


Figure 4.2: The recommender module consisting of both CF and CBF.

A user's vector consists of demographic information and a set of values in the range [0,1] describing the affinity for the given attributes. A value of 0 symbolizes a negative attitude towards the attribute, while a value of 1 symbolizes a positive attitude. If the user is indifferent to a certain aspect of the restaurant, this will be represented by a value of 0.5.

A restaurant is similarly represented by a vector. The only difference is that each of the boolean attributes are fixed as either a 1 or a 0, depending on the restaurant.

4.3.2 Content-based filtering

The content-based recommendation approach we use is called user-item recommendation. Conceptually, it works by calculating the set of restaurants that are most similar to a given user's profile. For example, a user that has rated many Asian restaurants favorably in the past will have a profile that more closely resembles an Asian restaurant, and the recommendations will reflect this.

To decide which restaurant the user most likely will enjoy, we need a way of calculating the similarity between user and restaurant. In the end, it is the restaurant with the highest similarity value that will be recommended to the user. In Section 2.2 we presented several similarity models that can be used to calculate this similarity, and for the purposes of this system we have decided to use a weighted euclidean distance measure. The reason for this is that we want to be able to assign a level of importance varying with each attribute, and we would not be able to do this with, for example, a cosine-based measure.

$$sim(u,r) = \sqrt{\frac{\sum_{i=1}^{n}(u_i - r_i)^2 \times u_i}{\sum_{i=1}^{n} u_i}} \tag{4.1}$$

Equation 4.1 describes the process, given a restaurant *r* and user *u* as input. We use the degree of preference the user has for attribute r_i as a weight for the *i*th component of the distance. The difference between each user-restaurant attribute pair is multiplied by the weight and summarized. Hence, if there is a perfect match between *u* and *r* the similarity value will be 0.

However, if the user has a low value (close to 0) for an attribute it means the user has a negative preference towards that specific attribute, and in its current form Equation 4.1 does not account for this. A negative preference should have the same impact on the final similarity score as a positive preference. Therefore, we modify Equation 4.1 to have the weight as a function of the user's preference instead of the preference itself.

$$sim(u,r) = \sqrt{\frac{\sum_{i=1}^{n}(u_i - r_i)^2 \times w_i}{\sum_{i=1}^{n} u_i}} \qquad \text{where } w_i = \begin{cases} u_i, & \text{if } u_i > 0.5 \\ 1 - u_i, & \text{otherwise} \end{cases} \tag{4.2}$$

Equation 4.2 considers a low preference value equally important as a high preference value. Without this small addition to Equation 4.1 *RestRec* would care very little if a user for example has a strong dislike for restaurants of a certain type. Listing 4.1 describes how we implement the CBF algorithm with Equation 4.2 as similarity measure.

```
def get_cb_recs(user_vector, restaurants):
    results = {}

    #for each restaurant, calculate the similarity between the user and
    the restaurant
    for restaurant in restaurants:
        results[restaurant] = similarity(user_vector, restaurant)

    results.sort()
    return results
```

Listing 4.1: Pseudocode for providing recommendations by means of CBF.

4.3.3 Collaborative filtering

The CF algorithm is conceptually quite simple: given a user *u*, find a set of other users, *U*, that have rated some of the same restaurants as *u* and recommend restaurants that are popular in *U*. In light of this, we decide to implement the algorithm ourselves. It is a *user-based* recommendation approach, meaning that the algorithm does not need to know anything about the restaurants themselves, except their identifying id's. The idea is to recommend restaurants that are popular among users that like the same restaurants as a specific user.

In practice, this is done by first building a *user-restaurant* rating matrix like the one shown in Table 4.1. The matrix is an overview of all the ratings in the system, making it easy to find the set of users that share ratings with any given user. In Listing 4.2 we show the algorithm for calculating similarity between two users, given the user-restaurant matrix.

For example if two users do not share any ratings, like *U1* and *U8*, their similarity score will be 0. Users having the exact opposite opinions about restaurants, like *U3* and *U6*, will have a similarity score approaching 0. However, users that rate the same restaurants equally, like *U2* and *U4*, will get a score of 1. The similarity value can thus range from 0 to 1, where 1 represents perfect agreement.

Table 4.1: An example user-restaurant rating matrix.

		Users										
		U1	**U2**	**U3**	**U4**	**U5**	**U6**	**U7**	**U8**	**U9**	**...**	**UN**
	R1	1	1		1		1	1				
	R2	1			1	1	-1	1		1		
Restaurants	**R3**		1		1		1		1			
	R4	-1	-1			-1	-1	1				
	R5		1	1		1	-1	1	-1			
	...											
	RN											

```
def sim_score(matrix, person, other):
    #assume the users have no ratings in common
    si = False
    for restaurant in matrix[person]:
        if restaurant in matrix[other]:
            si = True
            break

    #if they have no rating in common, return 0
    if not si:
        return 0

    #add up the squares of all differences
    sum_of_squares =
        sum([pow(matrix[person][restaurant]−matrix[other][restaurant],2)
            for restaurant in matrix[person]
                if restaurant in matrix[other]])
    #return the similarity as a value between 0 and 1
    return 1 / (1 + sum_of_squares)
```

Listing 4.2: Pseudocode for calculating the similarity-score between two users.

The complete CF algorithm is presented in Listing 4.3. For a given user, we compute the similarities to all other users, and if a similarity is greater than 0, we iterate through all restaurants rated by the other user. A total score describing the restaurant's popularity is calculated for every restaurant, and the more positive ratings a restaurant has received, the better it will score in relation to other restaurants. The restaurant-scores are calculated by multiplying the rating with the similarity-score between the two users. This approach allows more similar users to have a greater influence on the final recommendations compared to less similar users. In the end, it is the restaurant with the highest total score that is recommended to the user.

For example if we were to recommend a restaurant to *U2*, we would first calculate similarity-scores for the set of users that has rated some of the same restaurants as *U2*. In this case, users *U1*, *U3*, *U4*, and *U5* would all get similarity-scores of 1 and thus matter the most when calculating the top restaurant. However, as *U3* has not rated any other restaurants than the one shared with *U2*, we need only consider *U1*, *U4*, and *U5*. Restaurant *R2* is popular with these users, whilst *U2* has not rated it yet. Therefore, restaurant *R2* is the restaurant that will ultimately be recommended.

```
def get_cf_recs(matrix,person):
    results = {}

    for other in matrix:
        #don't compare me to myself
        if other == person:
            continue
        sim = sim_score(matrix,person,other)

        #ignore scores of zero or lower
        if sim <= 0:
            continue
        for restaurant in matrix[other]:
            #only score restaurants the person haven't rated yet
            if restaurant not in matrix[person]:
                #Similarity * score
                results.setdefault(restaurant,0)
                results[restaurant] += matrix[other][restaurant] * sim

    #return the sorted list
    results.sort()
    return results
```

Listing 4.3: Pseudocode for the CF algorithm.

4.4 Feedback

Feedback is necessary to understand the user's preferences. As explained in Section 2.4, there are two kinds of feedback: explicit and implicit. Optimally, one would want to take advantage of both techniques, but we have only implemented explicit feedback in *RestRec*. There are two reasons to this: first, implicit feedback such as dwell time is very noisy. Second, the benefit of implicit feedback is low compared to explicit feedback.

We use ratings as a means of explicit feedback in this system, predominantly because reviews call for sentiment analysis which is not a part of our scope. *RestRec* gives users the possibility to rate restaurants on a binary scale, either up or down, and we use this rating to update the user's profile. In what way we update the profile based on the user's rating as well as the restaurant's profile. Table 4.2 shows how we increase/decrease the degree of preference for the attributes that are relevant for the evaluated restaurant.

Table 4.2: The amount of change to the preferences depending on feedback.

Rating	**Change in preference**
Up	+*v*/-*v*
Down	-*v*/+*v*

This is better explained by example. Consider a restaurant *R* and a user *U* that are represented by the following vectors:

$$R = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \qquad U = \begin{bmatrix} 0.6 & 0.4 & 0.5 \end{bmatrix}$$

The user gives the restaurant a negative rating in this case and therefore dislikes the restaurant. Based on this, we can infer that the user prefers restaurants where the 1*st* and 3*rd* attributes are 1, and the 2*nd* attribute is 0. Thus, we can update the user's profile as follows:

$$\hat{U} = \begin{bmatrix} 0.6 + v & 0.4 - v & 0.5 + v \end{bmatrix}$$

In *RestRec*, the user-profiles have values ranging from 0 to 1, so we will have to choose the value of *v* accordingly. If the value is too high, the user-profile will ”jump” too much and never settle around a value, but if it is too low, the system will be slow to learn user-preferences. We cannot expect our users to rate enough restaurants that a change of 1-2 % per attribute is sufficient. Therefore, we set the value of *v* to be 0.03. If a user rates 20 restaurants with this value, there should be enough span in the user-profile to make recommendations.

4.5 Hybridization and cold start

Figure 4.2 shows that both filtering components output separate sets of possibly different recommendations. We need some way to combine those two sets into one set which can be presented to the user. In Section 2.1.3 we introduced a few possible solutions for how this can be done, and we choose to use the *mixed* scheme for our purposes. Based on the state of the system and the profile of a given user, we may wish to compose the list of recommendations in different ways. In general, we let CBF and CF have an equal impact of 40 % each on the final list. The last 20 % of the recommendations will be random restaurants to give the list some novelty. The recommendations are presented to the user in three groups: CBF, CF, and random restaurants, with headlines ”Based on your ratings”, ”Based on similar users”, and ”Random picks”, respectively.

One of the biggest motivations for a hybrid system is the level of robustness it provides with regards to cold-start problems. Cold start comes in two forms: cold user, and cold item.

4.5.1 Cold user

There are two problems to consider when a new user is introduced in an already established RS with a high amount of both users and ratings. First, a new user does not necessarily have a fine-tuned preference-vector, thus calculating similar restaurants to the user vector is unlikely to give good results. Second, the CF approach needs ratings in order to make recommendations. In an attempt to rectify these shortcomings, we introduce what we call *cold rules* in the recommender engine. One of these rules states that if a user has made less than ten ratings, we consider it a cold start. The reason behind this specific amount is based on the value of v explained in the previous section. As each rating can change the value of a field by 3 % the effect of ten ratings can potentially be 30 %, making the user's preference-vector in the range of 0.2 to 0.8. We consider this a large enough span for making recommendations based on CBF.

CBF When making cold start content-based recommendations, the engine will compute the average of the few restaurants the user has liked thus far and recommend the restaurants that are closest to this average. This variation of CBF is called item-item CBF and is different from the user-item approach that is normally used when recommendations are based on the user's preference vector. The algorithm for doing this is presented in Listing 4.4

```
def get_cold_cb_recs(rated_restaurants ,all_restaurants):
    results = {}
    #take an average of the few restaurants the user has rated
    average_restaurant = average(rated_restaurants)
    for restaurant in restaurants:
        results[restaurant] = similarity(average_restaurant , restaurant)
    #return the most similar restaurants to the average restaurant
    results.sort()
    return results
```

Listing 4.4: Pseudocode for content-based recommendation when cold start.

CF To overcome the cold user problem in the CF approach, we use demographic data. Instead of using the user-restaurant rating matrix defined in Section 4.3.3 to find similar users, we use personal information such as nationality, age, and gender. The idea is that users in the same demographic group have the same taste in restaurants, and we use these stereotypes to generate recommendations.

After finding the set of most similar demographic users, we compute the most popular restaurants in the set and recommend the restaurants the user has not rated yet. For example, if cheap restaurants are popular among young Norwegians it is likely that newly registered young Norwegians will enjoy these restaurants as well. When the new user after a while develops a more accurate profile by means of feedback, CF will be based on ratings rather than demographic information. The steps for this approach are described in Listing 4.5.

```
def get_cold_cf_recs(user, other_users, ratings):
    similar_users = {}

    #calculate the similarity between the user with the others based on
    demography
    for other_user in other_users:
        similar_users[other_user] = similarity(user.demography(),
    other_user.demography())

    #get the ten most similar users
    most_similar_users = similar_users.sort()[:10]

    #return the most popular restaurants for the most similar users to
    this_user
    most_popular_restaurants = get_most_popular_restaurants_by_users(
    most_similar_users)
    return most_popular_restaurants
```

Listing 4.5: Pseudocode for cold start CF based on demographic information.

4.5.2 Cold item

The problem with cold items is that they have a lesser chance of being recommended. With a classical user-based CF approach, they would never be recommended at all since no one has rated them. CBF, however, is based only on item-content and would always be able to recommend cold items. Furthermore, randomization among some of the recommended items will help promote cold items. We mentioned earlier that 20 % of the final recommendations in *RestRec* will be random restaurants.

In Section 2.7.4 we speak of *the principal properties of RSs*, and *coverage* is one of these properties. It is important for actors employing RSs that they have good coverage, meaning that they are able to recommend the whole set of items, and recommend to the whole set of users.

4.6 Context in *RestRec*

Section 2.5 introduced the notion of context and context-aware RSs. Research shows that incorporating contextual information in RSs can improve recommendations, and in accordance with RQ4 (Section 1.2), we try to do this in *RestRec*. Some examples of contextual data that *RestRec* would benefit from are:

- *Timestamps* - The time of day may say something about the user's intentions.
- *Weather* - For example if it is a nice day, we may want to emphasize recommendations for restaurants with outdoor seating.
- *Location* - If the user is using the mobile interface, we can rule out restaurants that are far away.
- *The social setting of the restaurant visit* - If a user is, for example, planning a romantic date, we may be able to filter out certain types of restaurants.

In *RestRec* we will only focus on the last one, the social setting of the restaurant visit. The reason we omit the first one is because we will need to gather data from a set of users that use the system regularly, and the time-frame of this project makes that unlikely. We omit weather for the same reason, in addition to the need for connecting to some weather-service to get forecasts. Location is omitted because *RestRec* is first and foremost a web-based system and not a mobile application.

In Section 2.5 we explained three different methods for obtaining contextual information: explicitly, implicitly, and inferred. It would be possible to infer the social setting, but this will require a large amount of user-data and advanced machine-learning algorithms, thus we will not use this approach. And as we do not see a way to obtain this information implicitly, we are left with explicit. This means the active user will explicitly have to provide this information when rating restaurants and receiving recommendations.

To ease the implementation, we have chosen a set of 4 different ”situations” the user can select from when rating and receiving recommendations. The situations are: *business*, *romantic*, *casual*, and *special*. It is our opinion that users either go to a restaurant with colleagues, in a romantic setting, casual setting with friends or family, or on a special occasion. We do not impose on the users any definitions for the contexts as we feel it is up to the individual user to decide on the meaning for each situation. We do, however, hope that users rate consequently when rating in a given context as this will have a direct impact on the subsequent recommendations.

Up until now we have described a user as a single vector where each value represents the user's preference for an aspect of a restaurant, but this is a somewhat simplified explanation. In reality, we store a user as four different vectors — one complete user-preference vector for each context. So when we speak of a user's preferences in the algorithms listed previously (and presently), this is actually the user's preferences within a given context.

4.7 How it works

It is now finally time to go through the system in practice to see if we have met the requirements outlined in Section 4.1. We start by presenting the mechanisms under the surface in the application and continue with a walk-through of the system from the user's perspective.

4.7.1 Server-side

Considering that the goal of the work presented in this thesis is making recommendations, we do not go into technical details on other aspects such as authentication, presentation, and updating information. Instead, we describe the general flow of the system when a user requests recommendations. Up until this point we have described each function of the system separately, but in *RestRec* they are ultimately combined into a hybrid system. Listing 4.6 shows how the the different parts of the recommendation process described in earlier sections is interlaced to make one fully functioning recommender engine.

```
def recommender_engine(user, other_users, ratings, restaurants):
    cold_start = False

    #get the user's rated restaurants
    rated_restaurants = ratings[user]

    #if the user has less than ten ratings we consider it a cold start
    if len(rated_restaurants) < 10:
        cold_start = True


    cb_recs = {}
    cf_recs = {}
    random_recs = get_random_recs(restaurants)

    if cold_start:
        #can only get content-based recommendations if there exist any
    ratings
        if len(rated_restaurants) < 1:
            cb_recs = get_cold_cb_recs(rated_restaurants,restaurants)

        #get the recommendations based on demography
        cf_recs = get_cold_cf_recs(user, other_users, ratings)
    else:
        #get the recommendations for a more developed user
        cb_recs = get_cb_recs(user, restaurants)
        cf_recs = get_cf_recs(ratings, user)

    return cb_recs,cf_recs,random_recs
```

Listing 4.6: Pseudocode describing the flow of the *RestRec* recommender engine.

4.7.2 Client-side

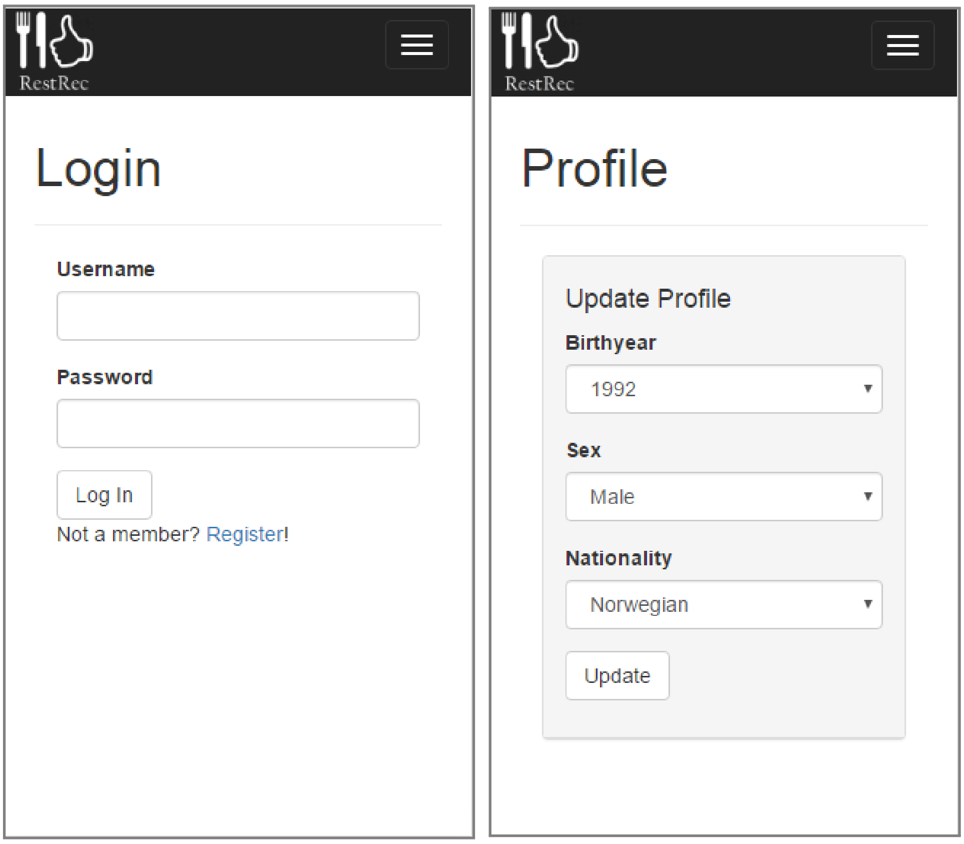
The client-side of the system is best described by presenting a selected set of screenshots from the perspective of a new user (Figures 4.3 to 4.6). *RestRec* has been implemented with a responsive front-end (requirement 2e), thus the screenshots are taken from the mobile site to save space.

When new users go to http://www.restrec.no in their browsers, they are met by the landing-page with information about *RestRec*. From there, it is easy to navigate to the login-page seen in Figure 4.3a. Users who do not have accounts have to register before they can log in. When logging in for the first time, users are taken directly to their profile pages as shown in Figure 4.3b where they can fill inn personal information that will be used in the recommendation processes. At this point, we have met requirements 1a, 1c, and 2e.

By tapping the top-right icon, the users get an overview of the system and what their possible actions are (Figure 4.4a). The top three options are what constitutes the restaurant part of the system and will be explained in more detail now. The last two options take the users to their profile pages or logs them out of the system, respectively.

On the right, in Figure 4.4b, we see the interface for the ”Rate restaurants” page where the users can both search for and rate restaurants. A list of 20 restaurants unrated by the user is shown below the search-fields. Every time a user rates a restaurant, the user-profile is updated to reflect the user’s preferences. We can therefore mark requirements 1b, 1d, 1e and 1g to the list of fulfilled requirements.

Figure 4.5 shows two different views of a restaurant. The first one is an overview from the ”Rate restaurants” page showing a small amount of information, while the other is a more detailed view that the user can see by clicking the name of the restaurant. Here the users can rate restaurants for the different situations we explained in Section 4.6. At the bottom of the view in Figure 4.5b is a Google Map showing the location of the restaurant (not shown in the figure). We can now mark requirement 1f as fulfilled.



Birthyear: 1992
Sex: Male
Nationality: Norwegian

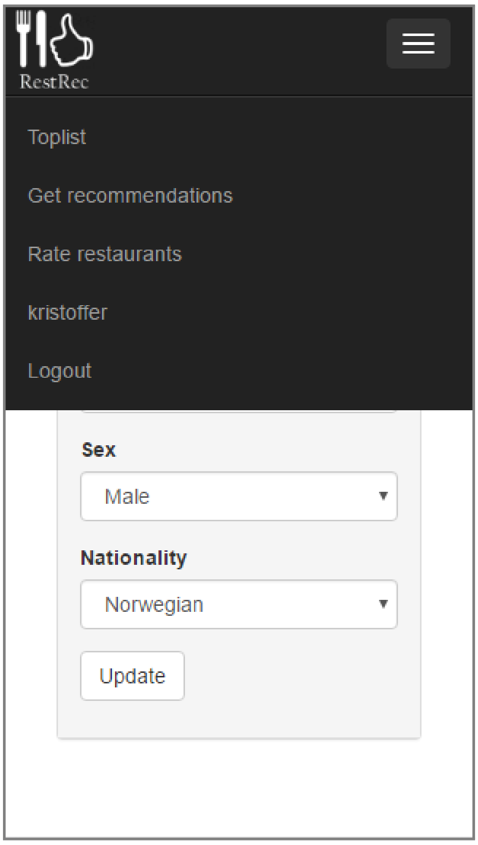
(a) The login page. **(b)** The profile page for an authenticated user.

Figure 4.3: *RestRec*.

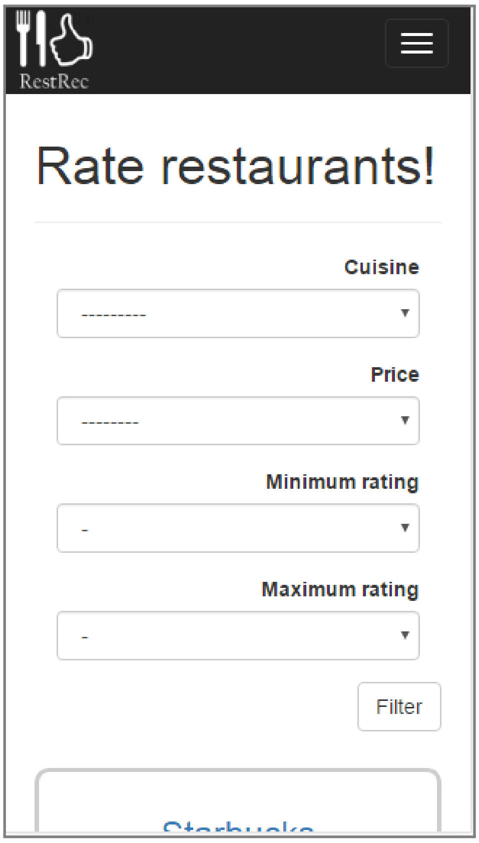
Finally, there are the ”Get recommendations” (Figure 4.6a) and ”Toplist” pages (Figure 4.6b). To get recommendations, the users have to select the wanted context before pressing the button. In this specific example the user has not rated any restaurants yet, so there are no results in the section for CBF recommendations. The second section shows results from the CF algorithm. Due to the fact that this specific user has not rated anything, the recommendations are based on demographic information only.

The ”Toplist” page is exactly what it sounds like. *RestRec* shows a list of the most popular restaurants for every context, with a number next to it describing how many users have upvoted the restaurant. With this said, we can consider requirement 1h as fulfilled.

To conclude this chapter, in Tables 4.3 and 4.4 is an overview of the requirements and whether we have met them or not. Requirement 2a, ”*the system should allow unauthenticated users to search for restaurants*”, has not been implemented by design. To test the system and gather data, we wanted as many people as possible to register. A way of achieving this was through restricting the access of unauthenticated users.



(a) By tapping the menu-icon, a dropdown shows a list of possible actions.



(b) *RestRec*'s "rate restaurants"-page with search functionality.

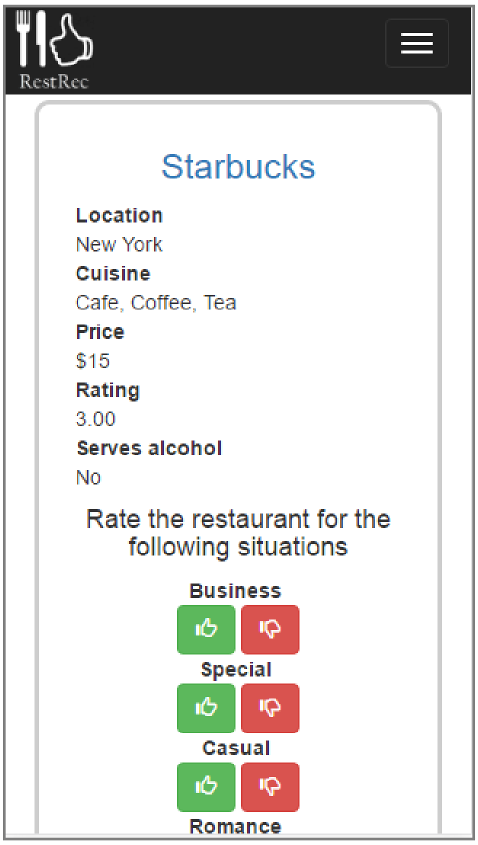
Figure 4.4: *RestRec*.

Requirement	Fulfilled	Not fulfilled
1a	✓	
1b	✓	
1c	✓	
1d	✓	
1e	✓	
1f	✓	
1g	✓	
1h	✓	

Table 4.3: Shown in this table is the list of *must implement* requirements from Section 4.1 and whether we have been able to fulfill them.

Requirement	Fulfilled	Not fulfilled
2a		✓
2b	✓	
2c	✓	
2d	✓	
2e	✓	

Table 4.4: Shown in this table is the list of *should implement* requirements from Section 4.1 and whether we have been able to fulfill them.

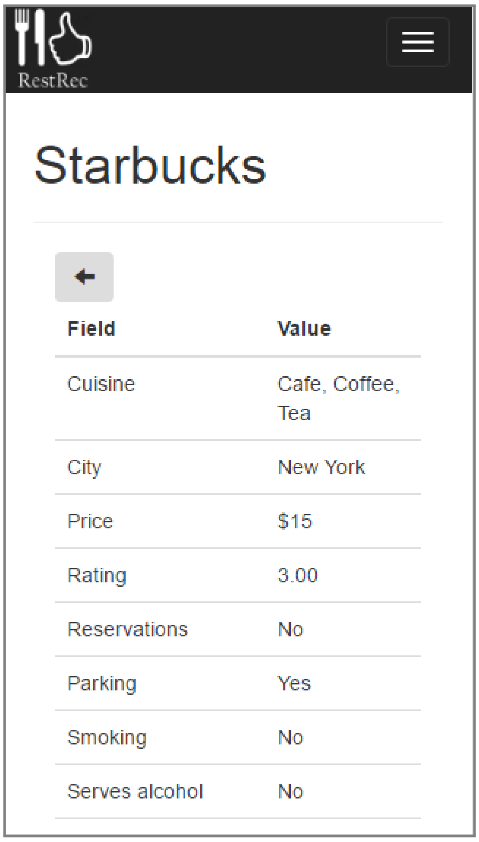


Starbucks

Location
New York
Cuisine
Cafe, Coffee, Tea
Price
$15
Rating
3.00
Serves alcohol
No

Rate the restaurant for the following situations

(a) *RestRec* presents an overview of a set of restaurants for the user to choose from.

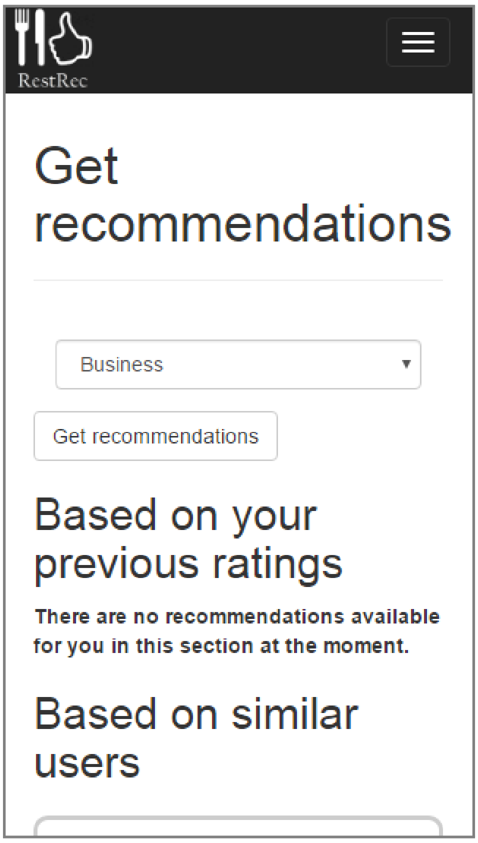


Starbucks

Field	Value
Cuisine	Cafe, Coffee, Tea
City	New York
Price	$15
Rating	3.00
Reservations	No
Parking	Yes
Smoking	No
Serves alcohol	No

(b) *RestRec* presents detailed information about a selected restaurant.

Figure 4.5: *RestRec*.



RestRec

Get recommendations

Business

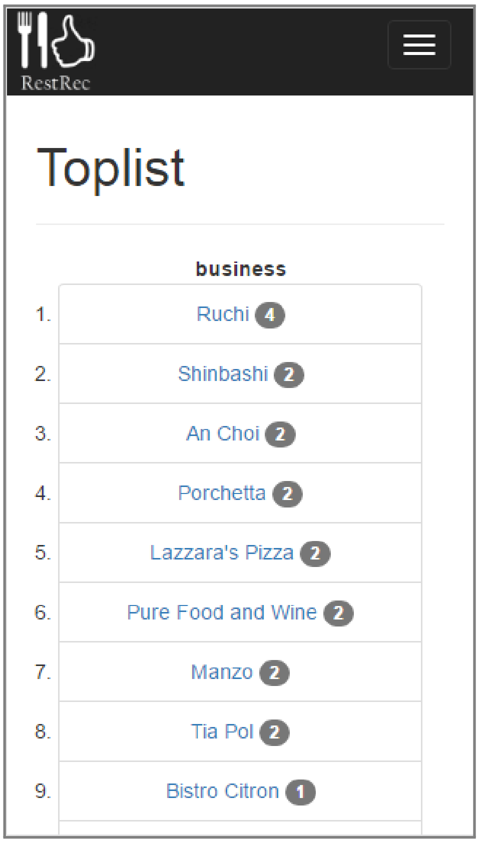
Get recommendations

Based on your previous ratings

There are no recommendations available for you in this section at the moment.

Based on similar users

(a) *RestRec* presents personal recommendations for a given situation.



RestRec

Toplist

business

1. Ruchi 4
2. Shinbashi 2
3. An Choi 2
4. Porchetta 2
5. Lazzara's Pizza 2
6. Pure Food and Wine 2
7. Manzo 2
8. Tia Pol 2
9. Bistro Citron 1

(b) *RestRec* shows a global toplist for every situation.

Figure 4.6: *RestRec*.

Chapter 5

Evaluation

In this chapter we evaluate *RestRec* and attempt to reach a conclusion regarding the research questions defined in Chapter 1. We begin by presenting an experimental plan in Section 5.1 where we state what type of experiments we will perform. However, to perform these experiments we need restaurant data, and this is the topic of Section 5.2. Following in Sections 5.3 and 5.4 are the experimental setup and results. These results are what forms the basis of the discussion taking place in Section 5.5. Finally, we revisit the original research questions in Section 5.6.

5.1 Experimental plan

In Section 2.7 we addressed the importance of evaluating an RS after it is implemented, and some of the ways this can be done. In brief, the evaluation of an RS has to measure whether real people are willing to act based on the recommendations. One of the simplest ways of doing this is to compare the difference in ratings between restaurants picked by a random generator and a recommender algorithm. With enough data this method will quite easily give an indication of the overall performance of the system and, furthermore, it is easy to compare the performance of the different recommendation approaches. It is also in our interest to plot performance as a function of ratings, as logic dictates that the prediction accuracy should increase with the number of ratings. As all recommendations in *RestRec* contain the name of the algorithm promoting them, we will perform experiments pertaining to this.

RestRec's ability to tackle a cold user is an area we have mostly focused on, so naturally we wish to ascertain whether or not we have been successful in this effort. There is only one way to reach a conclusion — to test the quality of the recommendations for cold users. In Chapter 4 we defined a cold user as a user with less than 10 ratings per context, so we will have to isolate and analyze these users before we attempt to make a conclusion.

RQ4 relates to situational information and how we can make use of this to improve recommendations. In *RestRec* we consider four different situations: *business*, *casual*, *romance*, and *special*. To learn how this extra piece of information impacts recommendations, we must analyze ratings with focus on the selected context. It would for example be interesting to see whether or not we can establish a connection between social setting and certain restaurant-attributes. But before we can do any of this, we need both user- and restaurant-data gathered by using *RestRec*.

5.2 Data domain analysis and data collection

Before we can truly call *RestRec* a *restaurant* RS, we need to find data on restaurants that can be recommended. This is also a crucial step if we are to perform any experiments and be able to evaluate the system.

In Section 4.2.1 we introduced the restaurant data domain and some of the thoughts we have regarding the data *RestRec* will be working with. However, we cannot base this project solely on our thoughts. We need solid quantitative data to substantiate our choices. Therefore, to learn more about the restaurant diners' preferences, we perform a survey with the intention of guiding us in the search for data to use in *RestRec* and our experiments.

5.2.1 Restaurant diners' preferences

The primary motivation for performing this survey is twofold. First, we wish to learn more about the users' social setting when visiting restaurants. Second, in order to meet the users' needs, we must learn how the users make their choices and what information they base them on. The last one is crucial for creating a system that has value and is capable of providing successful recommendations.

The survey is composed of the questions listed in Table 5.1. Considering that we want *RestRec* to be as self-contained as possible with regards to community preferences, we mostly include objective questions (questions that can readily be answered by the restaurant). This could be whether or not the restaurant has free WiFi, or if they serve alcohol. However, we do include questions related to community preferences as well as it is still interesting to see how much weight they carry with restaurant goers.

We posted the survey on Reddit[1] to attract attention from a broad set of people and to get as many responses as possible. Reddit is an entertainment and news website where registered community members can submit content, such as text posts or direct links, making it essentially an online bulletin board system. After 48 hours we had gotten 78 responses, and the rate of which we received new responses had dwindled down to zero. We therefore figured that this was the highest response rate we could expect. The results are mostly as expected. Because Reddit has primarily American users, 50 % of the respondents reported that they were from the U.S., while the remaining half is a mixture between Norwegian, U.K., and Canadian users. Some results of the questions are shown graphically in Figure 5.1 while the answers to the "how important are..." questions are grouped in categories shown in Table 5.2.

We can see that, as one perhaps would expect, opinion-based attributes are important to the average diner. However, factual information such as type of food and price is also considered important, and this is the kind of data we can use in *RestRec*.

[1]Reddit: http://www.reddit.com

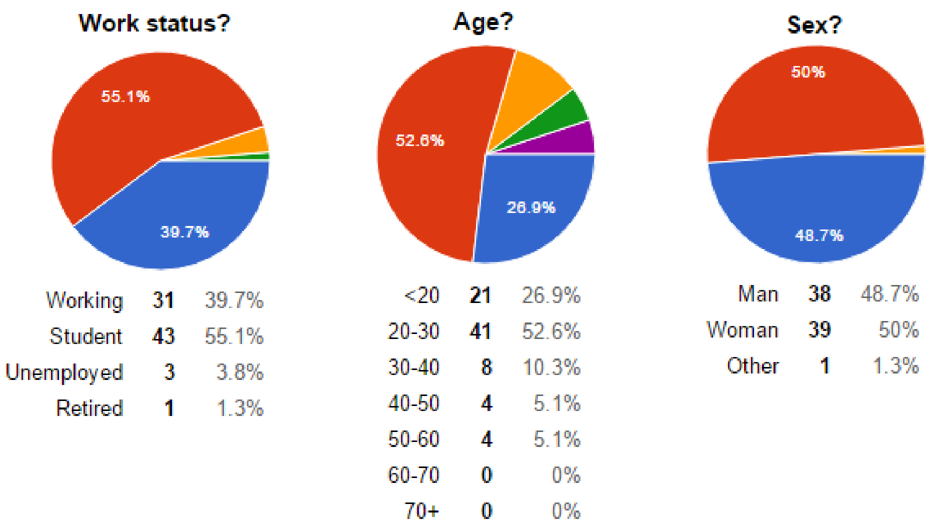
Table 5.1: Questions for the restaurant preferences survey.

Restaurant survey questions	
Questions	**Range**
Gender	[male, female]
Country	[list of countries]
Age	[20-29,...,60-70]
Work status	[working, student, unemployed, retired]
Who do you go to restaurants with?	[business associates, family, friends, partner]
How often do you go to restaurants?	[>once a week,..., <every 6 months or rarer]
How important are the following:	
Type of food	[1-5] (not important - very important)
Food taste	[1-5]
Price	[1-5]
Location	[1-5]
Rating/stars	[1-5]
Smoking	[1-5]
WiFi	[1-5]
Parking	[1-5]
Attire	[1-5]
Alcohol	[1-5]
Kids menu	[1-5]
Customized meals	[1-5]
Cleanliness	[1-5]
Menu variety	[1-5]
Availability of take-away	[1-5]
Food safety	[1-5]
Quality of staff	[1-5]
Service speed	[1-5]
Ambience	[1-5]

5.2.2 Data collection

With basis in what we learned from the survey, we start to look for data that can help us build *RestRec*. Many of the available restaurant datasets found on the Internet consist mostly of reviews and little factual data of the restaurant itself. However, in section 1.3 we mention that text analysis is not in the scope of this work. Thus, we focused our efforts on finding data with information such as what kind of food they serve, what their price range is, do they have WiFi, etc. In other words, information an establishment can easily provide themselves as there are no personal opinions involved.

The data we eventually found, describing a restaurant very comprehensively, is from

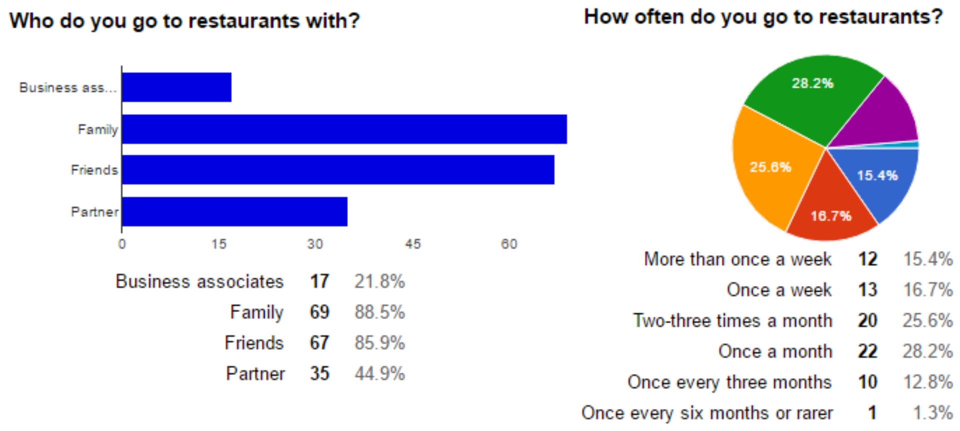


Working	31	39.7%
Student	43	55.1%
Unemployed	3	3.8%
Retired	1	1.3%

<20	21	26.9%
20-30	41	52.6%
30-40	8	10.3%
40-50	4	5.1%
50-60	4	5.1%
60-70	0	0%
70+	0	0%

Man	38	48.7%
Woman	39	50%
Other	1	1.3%

(a) Results from the survey showing age-, gender-, and work-distributions.



Business associates	17	21.8%
Family	69	88.5%
Friends	67	85.9%
Partner	35	44.9%

More than once a week	12	15.4%
Once a week	13	16.7%
Two-three times a month	20	25.6%
Once a month	22	28.2%
Once every three months	10	12.8%
Once every six months or rarer	1	1.3%

(b) Survey results.

Figure 5.1: Results from the restaurant diners' preferences survey.

Factual[2] data provider. By using their API we were able to quite easily retrieve information about thousands of restaurants worldwide. An overview of geographic locations and a description of the data are shown in Tables 5.3 and 5.4.

However, considering the scale of the work presented in this thesis, we decide it is not necessary to use the complete set of 2.3 million restaurants. The U.S. restaurant-data include extended information such as meal-type, alcohol, and ratings, which according to our survey is important enough to play a part when making recommendations. In addi-

[2]Factual: http://www.factual.com

Table 5.2: The distribution of responses from the survey regarding the importance of various attributes.

Not important	Somewhat important	Important
Smoking	Location	Type of food
WiFi	Rating/stars	Taste of food
Attire	Parking	Price
Kids menu	Alcohol	Cleanliness
Customized meals	Menu variety	Safety of food
	Take-away	Quality of staff
	Ambiance	
	Service speed	

Table 5.3: Distribution of where the Factual restaurants are located.

Country	Number of restaurants	Info
United States	>1.1 million	Includes core place data and extended restaurant attributes.
Great Britain	>300.000	
France	>400.000	
Germany	>400.000	
Australia	>100.000	

tion, as a large portion of the people who responded to the survey is American, it seems appropriate to use U.S. restaurants for the rest of our work. Consequently, we exclude restaurants that are not located in the U.S. due to not being as comprehensive, detailed and not resonating as well with our users. In total, the U.S. restaurants are described by a set of 64 different attributes which is listed in Appendix B.1.

For our CF method described in Section 4.3.3 to work, we are dependent on restaurants being rated by as many people as possible. As CF works best when the ratio of ratings vs items are reasonably high, a set of 1.1 million restaurants would consequently call for a user-base of substantial size. Therefore, we limit ourselves to handling restaurants in New York City only. Factual has data on over 20.000 restaurants in New York City, which is more than enough for the purpose of this thesis. To automate the data collection process, we wrote a python script which is listed in appendix A. This script retrieves restaurant data in JSON-format and stores it locally in the MongoDB instance we set up.

Furthermore, a dataset consisting of 20.000 restaurants is still too large-scale for our purposes. Thus, to increase our chances of obtaining good results, we must restrict the restaurant-set further. On the other hand, we still wish to have enough restaurants to ensure a representative selection of New York restaurants.

Due to the fact that the restaurant-data is incomplete (many fields do not have values), we utilize the results from the survey to filter out those restaurants which do not meet our requirements. We use the attributes *Location*, *Type of food*, *Price*, and *Rating* to remove

Table 5.4: A subset of the information Factual provides for each restaurant. See appendix B.1 for a complete list.

Attribute	Type	Description
Name	String	Entity name
Address	String	Address number and street name.
Locality	String	City, town or equivalent.
Neighborhood	String	The neighborhood(s) in which this entity is found
Region	String	State, province, territory, or equivalent.
Cuisine	String	The type of food served.
Price	Integer	A price metric between one and five.
Rating	Decimal	A rating between 1 and 5
Hours	String	JSON representation of hours of operation.
Attire	String	A single value from an enumerated list.
Attire_required	String	Gotta have this on to get in.
Attire_prohibited	String	Can't get in if you are sporting this.
Reservations	Boolean	Accepts reservations.
Smoking	Boolean	This place allows smoking somewhere.
Breakfast	Boolean	Serves breakfast.
Lunch	Boolean	Serves lunch.
Dinner	Boolean	Serves dinner.
Takeout	Boolean	Provides takeout/takeaway.
Cater	Boolean	Provides catering.
Alcohol	Boolean	Serves alcohol
Kids_goodfor	Boolean	Noted as being good for kids.
Kids_menu	Boolean	Has a kids menu.
Groups_goodfor	Boolean	Noted as being good for groups.
Seating_outdoor	Boolean	Outdoor seating is available.
WiFi	Boolean	WiFi is provided by the establishment.
Vegetarian	Boolean	Vegetarian options noted.
Vegan	Boolean	Vegan options noted.
Glutenfree	Boolean	Gluten free items noted.
Lowfat	Boolean	Lowfat options noted.
Organic	Boolean	Organic options noted.
Healthy	Boolean	Healthy dishes are explicitly available.

those where the available information is inadequate, and get down to 1256 remaining restaurants which now constitutes the final set of restaurants used in *RestRec*.

Due to having in total 63 different attributes per restaurant, we choose a subset of the most important and differentiating ones to present to the users. If we were to present all 63 attributes the user would most likely feel overwhelmed, and not put in the required time or effort, resulting in a bad user experience. The attributes that were deemed uninteresting by users are needless and therefore removed. Some of the attributes partly describe the same aspects (such as *kids_goodfor* and *kids_menu*) and are also redundant. Furthermore, we make the decision to focus only on dinner instead of including breakfast and lunch as we think this will not change the users' opinions of restaurants.

After processing the results from the survey we arrive at a final restaurant profile, shown in Table 5.5. There are 16 attributes in total, where 10 are boolean and 4 are multivalued. We expect this list of attributes will provide enough information for users to make a decision whether they would dine at a restaurant or not.

Table 5.5: The attributes shown to a user looking up restaurants.

Restaurant attributes			
Name	*Rating*	*Smoking*	*Kids_menu*
Locality	*Attire*	*Takeout*	*Seating_outdoor*
Cuisine	*Accessible_wheelchair*	*Parking*	*WiFi*
Price	*Reservations*	*Alcohol*	*Vegetarian*

However, even though we discarded restaurants with many undefined values, there are still some remaining and some of the restaurants have a bigger percentage of null-values than others. Figure 5.2 shows the distribution of null values per restaurant, and we can see that around 85 % of them has 3-5 undefined attributes.

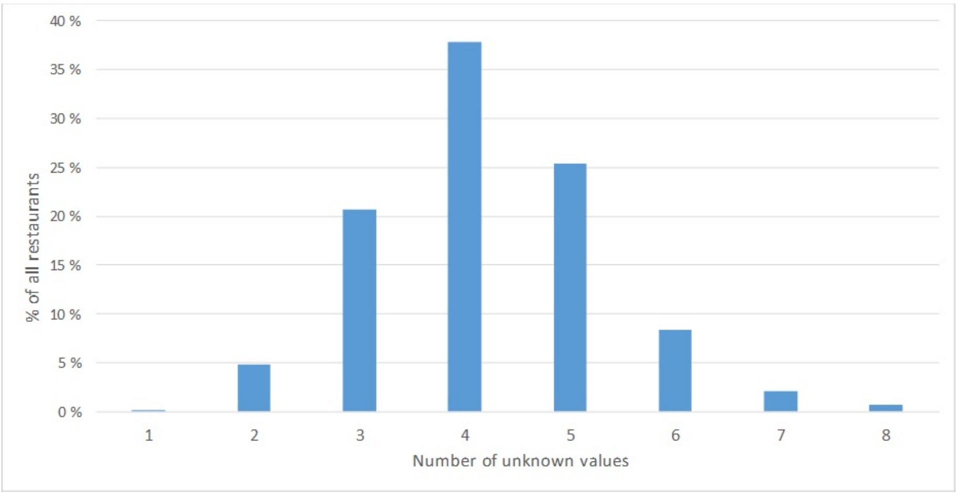


Figure 5.2: The distribution of null values per restaurant.

Similarly, the coverage varies from attribute to attribute. For the boolean values this is shown in Figure 5.3. We can see that several attributes are set for all restaurants, such as WiFi, reservations, and smoking. On the other hand we have attributes like *vegetarian options* and *kids menu* where there are a substantial amount of undefined values.

This is shown in Figure 5.3 for the boolean values. We can see that several attributes are set for all restaurants, such as WiFi, reservations and smoking. On the other hand we have attributes like vegatarion options and kids menu where there is a substantial amount of null values. In these cases we consider it likely the restaurant does not possess the quality in question, and we treat it accordingly (undefined equals False in *RestRec*). Our recommendations are based on this assumption, and as a consequence, restaurants with many blank attributes for qualitites they might still have will not be recommended as much as they could have been.

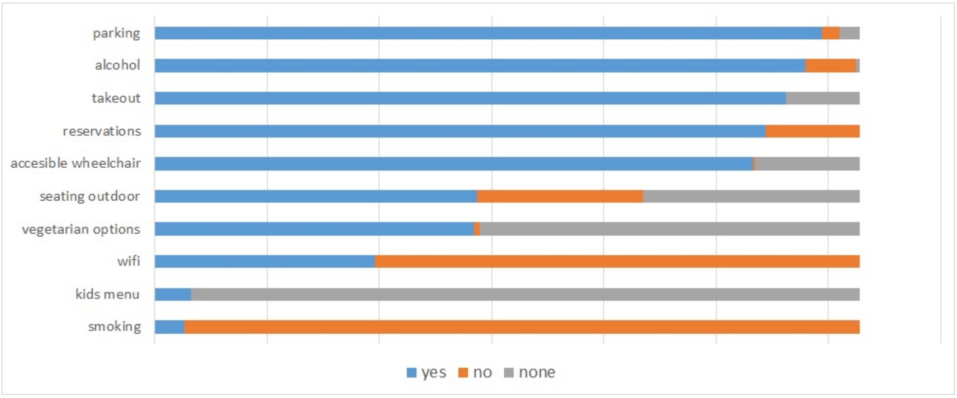


Figure 5.3: A distribution over the boolean values for the restaurants

The remaining attributes are defined for all restaurants, and in Figure 5.4 we see that the most popular cuisines are American, European, Cafe, and Italian. All in all, there are 121 different cuisines among the 1256 restaurants. Figure 5.5 shows that price and rating among the restaurants are quite evenly distributed, which should make for a good recommendation basis.

5.3 Experimental setup

RestRec was made public on http://www.restrec.no and spread to the users by use of social media and word of mouth. Due to the large item-space and relatively poor hardware we were not able to both update user-profiles and calculate recommendations in real time, and as a result we were forced to gather our data in two steps. First, we had the *profile-building* period where we asked users to rate as many restaurants as they wanted so that we could use the data to build accurate user-profiles. Second, we had the *recommendations* period were we tried to get the same set of people to rate the *quality* of their recommendations by use of ”Yes”, ”No”, or ”Neutral” for each recommended restaurant.

The *profile-building* period lasted for 12 days, counting a total of 45 different users providing 1691 ratings for 425 different restaurants. On average there are close to 423 ratings per context. In the gathering of this data, users were given a set of 20 restaurants and the option of rating them in any context.

The *recommendations* period lasted 3 days and 26 users out of the 45 from the previous phase returned to provide their opinions on the recommendations. In total we received

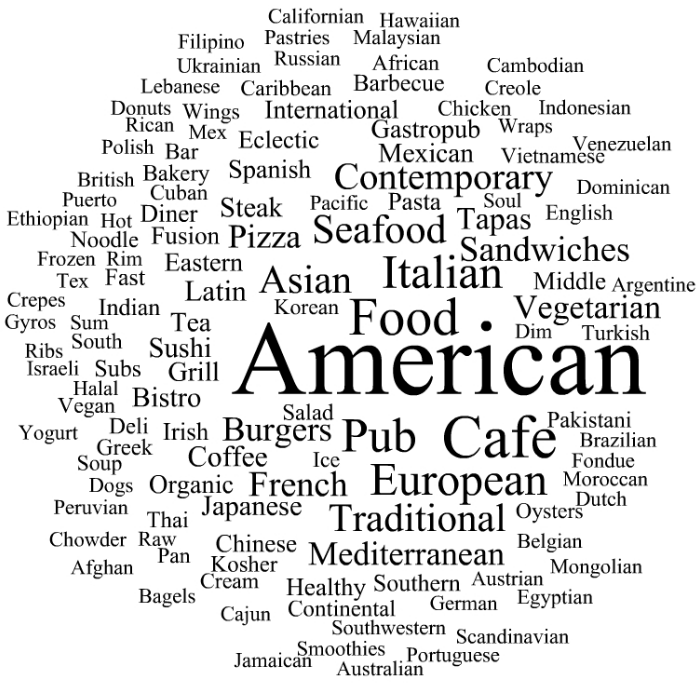
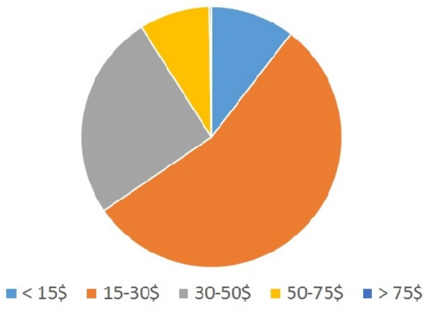
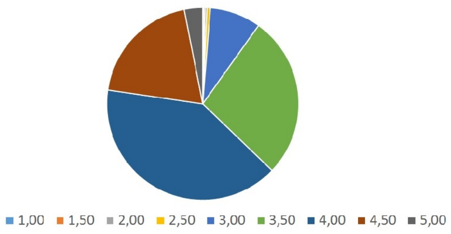


Figure 5.4: A word cloud representing the distribution of the different cuisines



(a) Price



(b) Rating

Figure 5.5: Distribution of the price and rating attribute on the restaurants

958 ratings, giving an average of 37 per user. When providing feedback on the recommendations in this phase users were presented with 10 restaurants (4 from CBF, 4 from CF, 2 random) in randomized order.

Our user-base is exclusively Norwegian, 75 % is aged 20-30, and evenly divided between male and female. The data presented here will form the basis on which we will perform our experiments and analyze *RestRec* in the next sections.

5.4 Experimental results

In this section we present various results with the goal of ascertaining the quality of *RestRec* and the recommendations made. We start by analyzing the users to determine whether they have been consistent in their ratings. This is important to establish as the subsequent results will be based on this data. After this has been done, we move on to the overall performance of the system where we aim to quantify the prediction accuracy of *RestRec*. The final two experiments are related to cold-start performance and how contextual information affect recommendations.

5.4.1 User analysis

As with all user-generated data, we have to consider the possibility that part of our data may be noisy, poor, or simply created with bad intentions. The ratings gathered in the *profile-building* period is what will drive the recommendations later, so we use these ratings to try to determine the users' consistency when rating. If users are more interested in rating for the purpose of providing data than actually making a conscious decision, the subsequent recommendations would reflect this and consequently bring down the average performance of the system. Thus, it is in our interest to exclude users that have not understood this when we perform our experiments.

The approach we took to determine this was to see if we could identify a difference between upvoted and downvoted restaurants per user. More specifically, we calculated the euclidean distance between the average upvoted restaurant and the average downvoted restaurant for every user in each context, and aggregated this into a single value for each user. Figure 5.6 shows the user-consistency graph. A low value is interpreted as a stronger indication of inconsistent rating than that of a high value. The goal of this experiment is to establish how users rated compared to a completely random ”dummy-user”, shown by a gray line in the figure (baseline). We can see that all users perform better than the baseline, but a large portion is very close to random.

To further establish the consistency of the ratings it is interesting to look at how much time passes between rating two different restaurants. This time-interval can give an indication of whether the user has read about the restaurant or not before rating, and also allows us to identify any trends. Hopefully, we will see that the time-interval is at a reasonable level and rather constant with respect to the amount of restaurants rated. Due to *RestRec* storing a timestamp for each rating, it is trivial to calculate a system-wide average for time passed between the rating of different restaurants. After filtering out large values that occurs between user-sessions, we get the results shown in Table 5.6. We can see that the time-interval does indeed stay constant independent of how many restaurants users rate.

5.4.2 Overall performance

To evaluate the overall performance of the system we decide to use the precision metric defined in Equation 2.13. Precision is an uncomplicated metric for calculating the success

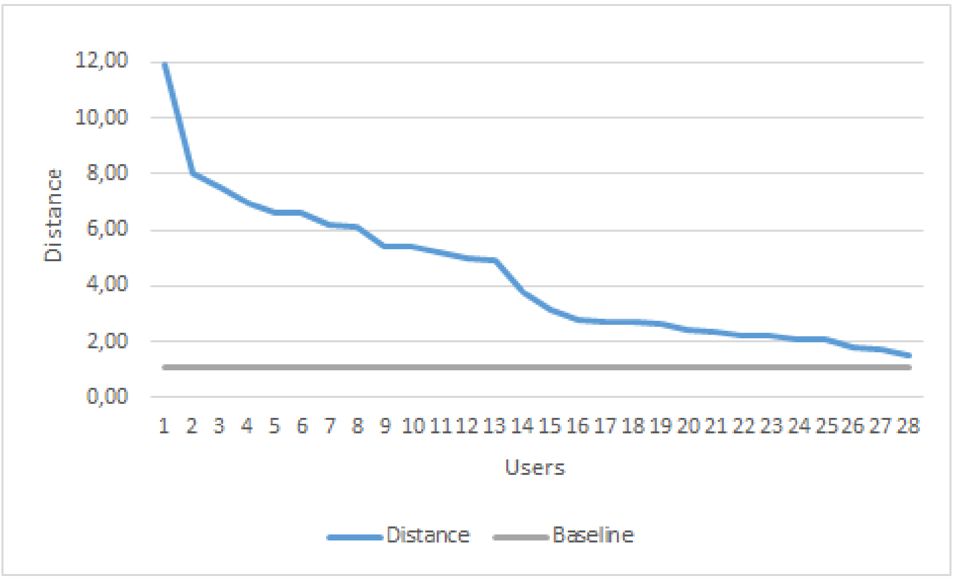


Figure 5.6: Distance between average upvoted and downvoted restaurant for each user, aggregated across each context. The baseline is that of a random ”dummy-user”.

Table 5.6: Average time-interval between rating different restaurants.

	Time-interval between rating restaurants				
Restaurant-pair	1-2	2-3	3-4	4-5	5-6
Seconds	12.4	13.3	10.1	10.3	10.0

rate of our recommendations, and it can be used to quickly establish the quality of the system.

In the experiment of calculating the overall performance of the system, a user requesting recommendations is presented with a list of 10 restaurants and the option to rate each of them with respect to the quality of the recommendation. The rating can be either one of ”Yes”, ”No”, or ”Neutral”. The precision is calculated for every user, and then aggregated with the precisions of other users having a similar amount of ratings. This way we can establish how the number of ratings affect the precision. Average precision for both CF and CBF is shown in Figure 5.7 with random recommendations serving as a baseline. We see that both our algorithms perform better than random, but there is no indication of the number of ratings having any effect on the precision.

In theory, the precision of CF should increase along with the number of users and ratings, as restaurants are then more likely to be rated by several users. Thus, we decide to examine the set of overlapping ratings in our data as this directly impacts the quality of CF. Shown in Table 5.7 are statistics for how many different users a restaurant is rated by. For example in *business*, only one restaurant is rated by 5 different users. We see that the

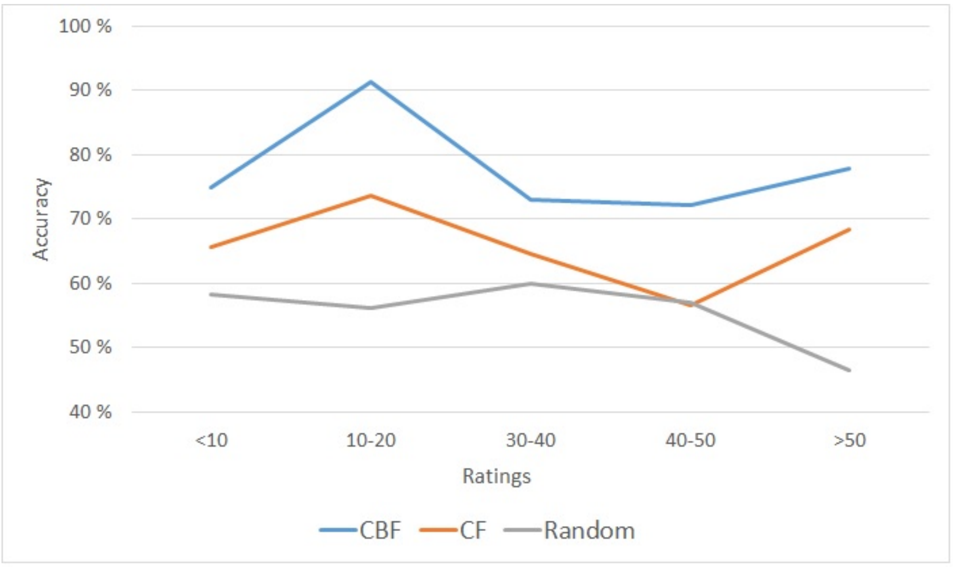


Figure 5.7: The precision of CF and CBF recommendations with regards to ratings, compared with a random baseline.

majority of restaurants are only rated by one user, but CF needs a restaurant to be rated by at least two users (the user requesting recommendations, and one other user) to be able to recommend it.

Table 5.7: This table shows how many unique users a restaurant is rated by, for each context. For example, within the business context there is only one restaurant rated by 5 unique users. 289 restaurants are rated by one user only.

Business		**Romance**		**Casual**		**Special**	
-	6	-	6	1	6	-	6
1	5	1	5	-	5	1	5
1	4	1	4	2	4	-	4
10	3	8	3	10	3	12	3
49	2	48	2	69	2	49	2
289	1	292	1	302	1	301	1

System-wide average precisions per context are presented in Table 5.8, showing that the system performs equally for all defined contexts.

5.4.3 Context

Our last experiment is related to context, and how the addition of this extra piece of information affects the recommendation process. Because of how we allow the users to rate a

Table 5.8: Average precision of *RestRec* for each context compared to a random generator.

	Our system	**Random**
Overall	71 %	53 %
Romance	70 %	54 %
Business	70 %	52 %
Special	72 %	53 %
Casual	73 %	53 %

restaurant for all contexts at the same time, most of the restaurants are rated for more than one context. By observing how users rate in one context compared to another, we can infer which attributes the users consider important to a context. To determine this, we calculate the average upvoted restaurant in each context and present a selected subset of attributes. The results are shown in Table 5.9.

Table 5.9: The average of the popular restaurants in the different contexts

	Special	**Romantic**	**Casual**	**Business**
Price	\$30-50	\$30-50	\$15-30	\$30-50
Avg. rating	4.5	4.0	3.5	4.0
Cuisine	Seafood, French, European	Italian European, American	Pub food, Cafe, American	Seafood, French, European
Serves alcohol	Yes	Yes	Yes	Yes
Smoking	No	No	No	No
Reservations	Yes	Yes	No	Yes
Takeout	No	No	Yes	No

We can see that the average user prefers cheaper restaurants and a simpler cuisine when in a casual setting. Romantic restaurants tend to be a little more expensive, and Italian food is popular choice. When looking for restaurants in the special context, users opt for the ones with high ratings. These are all reasonable results and a testament to the feasibility of employing contextual information in *RestRec*.

5.5 Discussion

Having implemented and evaluated *RestRec* quantitatively, it is time to discuss our findings. There are mainly three areas we wish to discuss in detail: *RestRec* and the technological restrictions imposed on our work, the collected data and the challenges related to this, and finally, the quality of recommendations made by *RestRec*.

5.5.1 *RestRec*

We mentioned in Section 5.3 that we gather data in two separate periods as a consequence of poor hardware and not having the ability to run the complete system in real time. However, there are benefits by doing it this way. For example, we would most likely have accumulated less data if we were to do it all at once. We could have spent much time to try and optimize the system with regards to speed, but ultimately it is the recommendations that are most important for our work. A complete *RestRec*-system would require users to switch between building their profiles and rate recommendations several times, and this would require much effort on the users' parts.

We also state that our user-base is exclusively Norwegian, but the survey we perform to learn more about restaurant diners' preferences are answered mostly by Americans. Logically, we should evaluate the system with an American user-base. However, such an approach would pose two problems: first, there are the possible consequences of posting *RestRec* on sites like Reddit (hacking, denial of service, etc), and second, it would be close to impossible to get the same set of users to come back a second time.

5.5.2 Data restrictions

When evaluating RSs one should preferably have access to a baseline which can be used for comparisons. In order to achieve this, work with the data must have been done at an earlier time. However, this has not been the case with *RestRec*, so naturally it is difficult for us to state whether we outperform other systems. To the best of our knowledge the restaurant data from Factual has not previously been used in a system such as ours, and the user-data is gathered specifically for our purposes. Ultimately, this leads us to evaluate the system against a random baseline which can only testify to the general intelligence of the recommendations. Given enough time and a large user-base it would be possible to perform *online evaluation* (Section 2.7.2) in order to find the optimal settings of the system, such as weights or how much we alter a user's profile based on feedback. Unfortunately, such an evaluation will have to wait due to time constraints.

Furthermore, there are the dangers of using crowdsourced data. First of all, the collected data can be discussed to be a bit narrow. Our users are exclusively Norwegian and rate restaurants located in New York City, U.S. Rating a restaurant in *RestRec* does not require the user to actually have eaten at the specific restaurant, or even been in the same country. In other words, the user does not need to spend much time, money, or effort in the rating of restaurants, which consequently can lead to more hasty and inaccurate ratings. Our user analysis in the previous section show that there are significant span in the consistency of ratings between users (Figure 5.6), but excluding too many users at this point would reduce our opportunities for further experiments.

Second, the size of the obtained data is too small to determine anything with reasonable certainty. Restaurant preferences collected from 45 users and the system evaluated by 26 users is not optimal, and can only serve as a starting point for further more conclusive experiments. Our results show promise (Figure 5.7) and we can see a certain trend in preferences, but we cannot claim to have established a ground truth for precision in restaurant recommendation.

And third, we calculate that the average user has only rated 9.4 restaurants per context.

This means that as per our definition of a cold user in Section 4.5.1, the average user of *RestRec* qualifies as a cold user. This is not ideal for the CBF algorithm, as we want users to have made at least 10 ratings to get a certain impression of the users' preferences.

5.5.3 Making recommendations

RestRec makes use of a combination of CBF and CF, with special *cold rules* to make recommendations. To what degree we are successful in overcoming the cold start problem is, once again, difficult to say due to the limited amount of available data. However, we know that the average user is a cold user and the results showed in Table 5.8 and Figure 5.7 clearly show that the system performs well above random. We should be able to obtain better results if we were to reduce the size of our restaurant data-set, but then again this would reduce the cold-start problem and that is not in our interest.

A total of 1256 restaurants leads to a very sparse user-restaurant rating matrix, making it hard to do CF successfully. Adding the fact that we consider each context in isolation, further increases the need for data. The restaurant domain is a domain where it takes a considerable amount of effort to provide proper feedback, thus it is important to make the most out of the feedback that is available. Having the users rate the same restaurant for every context is not a feasible approach in the long run. There should come a point where the system knows enough about a user to be able to infer the rating in a context given the rating in another context. This would drastically reduce the amount of needed data and pressure on the user to provide feedback.

Figure 5.7 shows how our CBF approach outperforms CF by nearly 10 % overall, and given the troubles of employing CF on our data, this does not come as a surprice. Considering the current state of our system, the relative gain in knowledge per rating is much greater for CBF than CF. A rating incurs a change of 3 % in the user's preference vector, but only further testing would reveal if this is a good value. In a larger system, it would perhaps be of interest to have a variable value instead of a constant. For example, the importance of a rating could be a function of how long ago the rating was made. This would allow for new ratings to have a larger impact than older ratings, and consequently make it easier for users to change their opinions about specific attributes.

5.6 Research questions revisited

> *How can we build an intelligent, personalized recommender system that works in the touristic domain by making use of established recommendation techniques?*

This has been the main research question and motivation for the work presented in this thesis, and at this point we have gained sufficient understanding and insight of the problem domain to attempt an answer. The question is threefold: first, there is the aspect of making intelligent and personalized recommendations. Second, there is the problem of defining the touristic domain. And finally, we wish to use established recommendation techniques.

By building *RestRec* we provide personal recommendations by making use of established techniques. In the previous section we showed that the recommendation accuracy is well above random, clearly indicating intelligence to a certain extent. And lastly, in Section 1.3 regarding the scope of our work, we substantiate our reasons for considering restaurant recommendation as a part of the touristic domain. This gives us a solid foundation to claim that we have indeed succeeded in our task, but there are always room for improvements.

Our research questions are listed below along with short evaluations of how they have been handled in this work.

5.6.1 RQ1

What challenges are there, and what methods have been developed to meet them?

In Chapter 2 we gave an introduction to the theory needed to implement an RS, and the challenges one is likely to encounter when doing so. We presented and discussed the problems of cold start and scalability, in addition to some emerging challenges like proactive recommendations and privacy. However, through our work with *RestRec* we have mainly been dealing with cold start and scalability, which are two of the most prominent problems regarding RSs.

The cold start problem pertains to the sparsity of information, and can affect both users and items. When a user is new to a system, the system does not know the user well enough to make accurate recommendations. For an item, the challenge is how to recommend an item that no one has rated yet. There are a variety of ways to handle a cold start, some of which are explained in Chapter 3. The most popular approach for dealing with a cold user is to have the user provide some information at the start, either through a questionnaire or by rating a set of example items.

The problem of scalability is the huge amount of operations involved in computing recommendations as the system grows larger. RSs very often handle data that is both high-dimensional and sparse, making calculations complex and time-consuming. Most users have very little patience when it comes to waiting for their recommendations, measures have to be taken to reduce the complexity of the data. In Section 2.6.2 we mentioned PCA and SVD as viable options for this task.

Nowadays, people are in a state of being recommended items at any time via their smart phones or on the websites they are currently surfing. This leads to the challenges of not only having to find out what to recommend, but also when to recommend it. The foundation of an RS is knowing as much as possible about its users and, as a result of this, privacy is becoming an increasingly important aspect to consider. It is very important that personal information is handled correctly so that sensitive information is not lost or stolen.

5.6.2 RQ2

What kind of systems already exist, and what are their strengths and shortcomings?

In Chapter 3 we presented a selected set of RSs for restaurants and tourism in general. The systems are called *R-Cube*, *I'm feeling LoCo*, *REJA*, *OpenTable*, and *TripBuilder*, and were selected to show some of the various approaches that are possible and to give a broad idea of existing solutions in both academia and industry.

Some of the systems make use of the most common methods like CF, CBF, or a hybridization of the two, whilst others make use of dialogue and knowledge-based methods. All of the systems deal with cold-start in different ways and are able to make acceptable recommendations in the tourism domain. However, only two of the systems incorporate contextual information into their algorithms and none of them have restaurant recommendation as their main focus.

Based on this analysis and the related research, there is much work to be done regarding context-aware RSs. The possible gain of employing more contextual information in RSs could be considerable.

5.6.3 RQ3

How can we find and use data to describe the items to be recommended?

Section 5.2 describes the process of finding data to use with *RestRec*. Before starting the search for appropriate data, we conducted a survey to learn more about how the users make their choices and what information they base them on. The results of the survey allowed us to restrict our search with regards to the type of data we needed. Many of the restaurant datasets we found on the Internet consisted mostly of reviews and not of information about the restaurants themselves. The data we eventually found, describing a restaurant very comprehensively, is from Factual data provider. Factual provides information on 2.3 million restaurants, each described by 64 different attributes. After modifying the data to fit our purposes we ended up with 1256 restaurants in New York which we utilized in *RestRec*.

5.6.4 RQ4

How can we identify the user's situation and use it to improve recommendations?

In Section 2.5 we introduced the concept of contextual recommendation. By incorporating available contextual information into the recommendation process as explicit additional categories of data, it is possible to make more accurate predictions. After researching the possibilities from other systems, we discovered that this type of recommendation can have a prominent impact in the restaurant domain. In Section 4.6 we presented how we wanted to incorporate context into *RestRec*, and we established four social situations within which the user could rate restaurants: *business*, *casual*, *romance*, and *special*. To learn more about how this extra piece of information impacts recommendations, we analyzed ratings with focus on the selected context. From the results described in Section 5.4.3, we establish that there is a connection between the social settings and certain restaurant-attributes.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis we have designed, implemented and evaluated an intelligent, personalized recommender system focusing on restaurant recommendation as a case. By conducting a review of recommender system literature and related work, we identified *hybrid* approaches as a possible solution to the cold-start user problem, and learned that contextual information has huge potential to improve this technology.

Our proposed system, *RestRec*, is implemented as a web-page where users can sign up, provide their opinions on a set of restaurants, and receive recommendations on where to eat. The recommendations are made by a hybridization of collaborative filtering and content-based filtering, with the addition of what we refer to as *cold rules* for handling cold starts. We performed a survey to learn more about users' social setting when visiting restaurants and what information they base their restaurant choices on. This knowledge is subsequently used to guide our search for restaurant data to be used in the evaluation phase. Users of *RestRec* can select a social setting when requesting recommendations.

Due to a lack of comparable systems, the prediction accuracy is calculated with a random generator as baseline. The results show that our approach outperform the baseline by almost 20 %, and we are able to ascertain the benefit of using contextual information for making recommendations. After studying and analyzing existing work in the area, we have concluded that our approach is feasible, but there are more challenges that still are remaining to be solved.

6.2 Future work

We discovered many useful techniques and features when researching for and designing *RestRec*, but we had to prioritize the methods that would help us reach our research goals. In other words, there are many ways to expand on our system, and following are some suggestions on what future work can be focused on.

6.2.1 Using reviews for text analysis

Textual reviews are a rich source of information. Given enough reviews about a restaurant it is possible to, by use of text analysis software, to extract information. It is for example possible to learn what the strengths of a certain restaurant are, or what could be improved. In the future, it would be interesting to look at the possibility of incorporating textual reviews in *RestRec* and to complement the Factual[1] data.

As explained in the survey of the various available data, we could not find any data sets containing both user and restaurant information. We thought of the possibility to use some of the large restaurant review datasets, but although they contain much information about user preferences, they have very little information about the restaurants themselves.

We want restaurant data in order for CBF to be able to recommend cold items. However, if we get enough restaurant information from the reviews, we can use the user data and hence recommend restaurants with CF right away. Additionally, with this user data we could do a more quantitative evaluation by making test sets and trying to predict how well users like certain restaurants before comparing the prediction against the ground truth. The solution could be to employ text analysis techniques on reviews to extract information about the restaurants.

6.2.2 Factual data

Factual is in possession of data about many different entities in the touristic domain. In addition to the restaurant data we have made use of, Factual can give our system a whole new dimension by recommending hotels as well. They also have several other Point of Interests such as museums, cinemas, theatres, amusement parks and landmarks. An RS which makes use of all these entities could very well be the ultimate tourism recommender.

6.2.3 Implicit feedback

In Section 2.4.2 we introduced the concept of implicit feedback which refers to getting feedback without requiring any active involvement from the user. In our system we decided to not make use of this at the moment. First, because implicit feedback such as dwell time is very noisy. Second, the benefit of implicit feedback is low compared to explicit feedback. However, this is something that can be incorporated into the system over time as a part of future work. With help from the time the user has spent inspecting a restaurant profile or the average time used to rate restaurants the system can get additional information to base its recommendations on. For example, if a user never clicks on a certain type of restaurant presented, the system can learn after a time that the user does not favor that particular type of restaurant.

6.2.4 Additional contextual information

We have dedicated a significant amount of our work in this thesis to do context-aware recommendation, but as described in Section 4.6 there are several other contexts to consider. For example, when providing recommendations it can be important to recommend

[1]Factual: http://www.factual.com

restaurants nearby, taking into account the location context. This can be incorporated in our system if it was developed into a mobile application, or let the user specify their current exact location explicitly. The weather may also have an impact on people's choice of restaurants. If it is hot outside, the user may prefer restaurants with air conditioning or outdoor seating. What time of the day it is can be important if the user wants to find a restaurant which is open at the moment. In other words, location, weather, and time, make the restaurant domain very relevant for further context-aware recommendations.

Bibliography

Adomavicius, G. & Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, *17*(6), 734–749.

Anand, S. S. & Mobasher, B. (2007). Contextual Recommendation. In *From web to social web: discovering and deploying user and content profiles* (pp. 142–160). doi:10.1007/978-3-540-74951-6_8

Bakhshi, S., Kanuparthy, P., & Gilbert, E. (2014). Demographics, weather and online reviews: A Study of Restaurant Recommendations. In *Proceedings of the 23rd international conference on world wide web - www '14* (pp. 443–454). New York, New York, USA: ACM Press. doi:10.1145/2566486.2568021

Balabanović, M. & Shoham, Y. (1997, March). Fab: content-based, collaborative recommendation. *Commun. ACM*, *40*(3), 66–72. doi:10.1145/245108.245124

Barnes, T. J. (2013). Big data, little history. *Dialogues in Human Geography*, *3*(3), 297–302. doi:10.1177/2043820613514323

Brilhante, I., Macedo, J. A., Nardini, F. M., Perego, R., & Renso, C. (2013). Where Shall We Go Today ? Planning Touristic Tours with TripBuilder. *Cikm'13*, 757–762. doi:10.1145/2505515.2505643

Burke, R. (2000). Knowledge-based recommender systems. *Encyclopedia of library and information systems*, *69*(Supplement 32), 175–186. doi:10.2991/iske.2007.110

Burke, R. (2002). Hybrid recommender systems: survey and experiments. *User Modeling and User-Adapted Interaction*, *12*(4), 331–370. doi:10.1023/A:1021240730564

Cortes, C. & Vapnik, V. (1995). Support-Vector Networks. *Machine Learning*, *20*(3), 273–297. doi:10.1023/A:1022627411411. arXiv: arXiv:1011.1669v3

Cover, T. & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, *13*(1), 21–27. doi:10.1109/TIT.1967.1053964

Das, S. (2015). Making Meaningful Restaurant Recommendations At OpenTable. In *Proceedings of the 9th acm conference on recommender systems - recsys '15* (pp. 235–235). New York, New York, USA: ACM Press. doi:10.1145/2792838.2799501

Duda, R. O. & Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. doi:10.2307/1573081

Felfernig, A., Gordea, S., Jannach, D., Teppan, E., & Zanker, M. (2006). A short survey of recommendation technologies in travel and tourism. In *Ogai journal (oesterreichische gesellschaft fuer artificial intelligence)* (Vol. 25, *4*, pp. 17–22).

Gesellschaft für Konsumforschung. (2014). Travel booking statistics. [Statistics]. Retrieved from http://www.gfk.com/insights/press-release/around-90-percent-of-travel-bookings-today-involves-going-online-compared-to-only-50-percent-in-2006-gfk/

Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, *35*(12), 61–70. doi:10.1145/138859.138867. arXiv: 39

Golub, G. H. & Reinsch, C. (1970). Singular value decomposition and least squares solutions. *Numerische mathematik*, *14*(5), 403–420.

Hearst, M. A., Dumais, S. T., Osman, E., Platt, J., & Scholkopf, B. (1998). Support vector machines. *IEEE Intelligent Systems*, *13*, 18–28. doi:10.1109/5254.708428

Herlocker, J. L., Konstan, J. A., Borchers, A., & Riedl, J. (1999). An algorithmic framework for performing collaborative filtering. SIGIR '99, 230–237. doi:10.1145/312624.312682

Herlocker, J. L., Konstan, J. A., Terveen, L. G., & Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, *22*(1), 5–53. doi:10.1145/963770.963772. arXiv: 50

IEEE Standards Association. (2011). IEEE specification 29148:2011. [Specification standard]. Retrieved from https://standards.ieee.org/findstds/standard/29148-2011.html

International Telecommunication Union. (2015). Internet user statistics. [Statistics]. Retrieved from http://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx

Jain, a. K., Murty, M. N., & Flynn, P. J. (1999). Data clustering: a review. *ACM Computing Surveys*, *31*(3), 264–323. doi:10.1145/331499.331504. arXiv: arXiv:1101.1881v2

Joachims, T. (1998). Text categorization with support vector machines: Learning with many relevant features. *Machine Learning: ECML-98*, *1398*, 137–142. doi:10.1007/BFb0026683

Jolliffe, I. T. (2002). Principal Component Analysis, Second Edition. *Encyclopedia of Statistics in Behavioral Science*, *30*(3), 487. doi:10.2307/1270093

Kim, S. & Banchs, R. E. (2014, December). R-cube: a dialogue agent for restaurant recommendation and reservation. In *Signal and information processing association annual summit and conference (apsipa), 2014 asia-pacific* (pp. 1–6). doi:10.1109/APSIPA.2014.7041732

Koren, Y. (2008). Factorization meets the neighborhood: a multifaceted collaborative filtering model. KDD '08, 426–434. doi:10.1145/1401890.1401944

Krulwich, B. (1997). LIFESTYLE FINDER: Intelligent User Profiling Using Large-Scale Demographic Data. *AI Magazine*, *18*(2), 37. doi:10.1609/aimag.v18i2.1292

Lam, S. K., Frankowski, D., & Riedl, J. (2006). Do you trust your recommendations? An exploration of security and privacy issues in recommender systems. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 3995 LNCS, pp. 14–29). doi:10.1007/11766155_2

Lieberman, H. & Selker, T. (2000). Out of context: Computer systems that adapt to, and learn from, context. *IBM Systems Journal*, *39*(3.4), 617–632. doi:10.1147/sj.393.0617

Lika, B., Kolomvatsos, K., & Hadjiefthymiades, S. (2014). Facing the cold start problem in recommender systems. *Expert Systems with Applications*, *41*(4 PART 2), 2065–2073. doi:10.1016/j.eswa.2013.09.005

Linden, G., Smith, B., & York, J. (2003). Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, *7*(1), 76–80. doi:10.1109/MIC.2003.1167344

Lops, P., de Gemmis, M., & Semeraro, G. (2011). Recommender systems handbook. doi:10.1007/978-0-387-85820-3_3

Mander, J. (2015). Gwi social.

Martinez, L., Rodriguez, R. M., & Espinilla, M. (2009). REJA: A Georeferenced Hybrid Recommender System for Restaurants. In *2009 ieee/wic/acm international joint conference on web intelligence and intelligent agent technology* (pp. 187–190). IEEE. doi:10.1109/WI-IAT.2009.259

McKinsey & Company. (2011). Big data: The next frontier for innovation, competition, and productivity. *McKinsey Global Institute*, (June), 156. doi:10.1080/01443610903114527

Mervis, J. (2012). Agencies rally to tackle big data. *Science*, *336*(6077), 22–22. doi:10.1126/science.336.6077.22. eprint: http://science.sciencemag.org/content/336/6077/22.full.pdf

Michie, E. D., Spiegelhalter, D. J., & Taylor, C. C. (1994). Machine Learning , Neural and Statistical Classification. *Technometrics*, *37*(4), 459. doi:10.2307/1269742

Netflix. (2015). Netflix statistics. [Statistics]. Retrieved from http://files.shareholder.com/downloads/NFLX/860811406x0x854558/9B28F30F-BF2F-4C5D-AAFF-AA9AA8F4779D/FINALQ315LettertoShareholdersWithTables.pdf

Illustration of KNN. (n.d.). [Image]. Retrieved from http://cgm.cs.mcgill.ca/~godfried/teaching/projects.pr.98/sergei/figure/figure2.gif

Illustration of SVM. (n.d.). [Image]. Retrieved from http://38.media.tumblr.com/0e459c9df3dc85c301ae41db5e058cb8/tumblr_inline_n9xq5hiRsC1rmpjcz.jpg

PCA applied to a Gaussian distribution. (n.d.). [Image]. Retrieved from https://upload.wikimedia.org/wikipedia/commons/thumb/f/f5/GaussianScatterPCA.svg

The result of a cluster analysis. (n.d.). [Image]. Retrieved from http://dic.academic.ru/pictures/wiki/files/67/Cluster-2.svg

Osuna, E., Freund, R., & Girosit, F. (1997). Training support vector machines: an application to face detection. *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 130–136. doi:10.1109/CVPR.1997.609310

Rennie, J. D., Shih, L., Teevan, J., Karger, D. R., et al. (2003). Tackling the poor assumptions of naive bayes text classifiers. In *Icml* (Vol. 3, pp. 616–623). Washington DC).

Resnick, P. & Varian, H. R. (1997). Recommender systems. *Communications of the ACM*, *40*(3), 56–58.

Saiph Savage, N., Baranski, M., Elva Chavez, N., & Höllerer, T. (2012). I'm feeling LoCo: A Location Based Context Aware Recommendation System. *Advances in Location-Based Services*, 37–54. doi:10.1007/978-3-642-24198-7_3

Stern, D., Herbrich, R., & Graepel, T. (2009). Matchbox: large scale bayesian recommendations.

Wang, C. & Blei, D. M. (2011). Collaborative topic modeling for recommending scientific articles. KDD '11, 448–456. doi:10.1145/2020408.2020480

Watterson, B. (2005). *The complete calvin and hobbes*. Andrews McMeel Publishing.

Yeung, K. F. & Yang, Y. (2010). A proactive personalized mobile news recommendation system. In *Proceedings - 3rd international conference on developments in esystems engineering, dese 2010* (pp. 207–212). doi:10.1109/DeSE.2010.40

Appendix A

The script for scraping Factual

```
from factual import Factual
from pymongo import MongoClient
import time
import random

# Factual
o_auth_key = "lorem"
o_auth_secret = "ipsum"
schema = "restaurants-us"

factual = Factual(o_auth_key, o_auth_secret)
restaurants = factual.table('restaurants-us')
limit = 50
# Mongo
client = MongoClient()
client.master_db.authenticate("lorem", "ipsum")
db = client.master_db.restaurants

ratings = [1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]

#this is a list with strings that describes the neighbourhoods in new york
neighborhoods = []

done_with = []

too_big = []


def insert_restaurants(data):
    count = 0
    nof_restaurants = len(data)

    for restaurant in data:
        restaurant["_id"] = restaurant.pop("factual_id")
        try:
            db.insert(restaurant)
```

```python
            count += 1
        except Exception as e:
            pass
    print("{}/{} restaurants inserted.".format(count, nof_restaurants))


def get_data_and_insert(data, query):
    offset = 0
    insert_restaurants(data)

    while len(data) == limit:
        offset += 50
        time.sleep(random.randint(5, 10))
        data = restaurants.filters(query).limit(limit).offset(offset).data()
        insert_restaurants(data)

for hood in neighborhoods:
    if hood in done_with or hood in [x for x,_ in too_big]:
        continue

    get_by_rating = False
    offset = 0
    print("Currently working with neighborhood: {0}".format(hood))

    query = {'$and':[{'region':{'$eq':'NY'}},{'neighborhood':{'$blank':False}},
    {'locality':{'$eq':'NEW YORK'}},{'neighborhood':{'$in':[hood]}}]}
    result = restaurants.filters(query).limit(limit).include_count(True)
    tot_count = result.total_row_count()

    db_count = db.find({"neighborhood":hood}).count()
    print("There are {} restaurants in {}. We currently have {} of them in our database.".format(tot_count, hood, db_count))

    if db_count >= tot_count:
        print("Since we have them all, we move on.")
        print()
        continue

    if tot_count > 500:
        print("Getting restaurants by rating.")
        get_by_rating = True

    if get_by_rating:
        for rating in ratings:

            query = {'$and':[{'region':{'$eq':'NY'}},{'neighborhood':{'$blank':False}},
            {'locality':{'$eq':'NEW YORK'}},{'rating':{'$eq':rating}},{'neighborhood':{'$in':[hood]}}]}
            result = restaurants.filters(query).limit(limit).include_count(True)
            count = result.total_row_count()
            db_count = db.find({"neighborhood":hood, "rating":rating}).count()
            print("We have {}/{} of the restaurants with rating {}.".format(db_count, count, rating))

            if db_count >= count:
                continue

            if count > 500:
                too_big.append((hood, rating))
                print("Too big: {}".format(too_big))
                continue

            get_data_and_insert(result.data(), query)

        # Try to get the last ones without a rating
        query = {'$and':[{'region':{'$eq':'NY'}},{'neighborhood':{'$blank':False}},
        {'locality':{'$eq':'NEW YORK'}},{'rating':{'$blank':True}},{'neighborhood':{'$in':[hood]}}]}
        result = restaurants.filters(query).limit(limit).include_count(True)
        count = result.total_row_count()
        db_count = db.find({"neighborhood":hood, "rating":{"$exists":False}}).count()
        print("We have {}/{} of the restaurants with rating {}.".format(db_count, count, "blank"))
        if count > 500:
            too_big.append((hood, "blank"))
            print("Too big: {}".format(too_big))
        else:
            get_data_and_insert(result.data(), query)

    else:
        get_data_and_insert(result.data())

    db_count = db.find({"neighborhood":hood}).count()
    print("Done with {}. There are now {}/{} restaurants located in {} in the database. Moving on to the next hood.".format(hood, db_count, tot_count, hood))

    time.sleep(random.randint(5, 10))
    print()
```

Listing A.1: Code for scraping Factual.

Appendix B

Factual

B.1 Factual data

Table B.1: A complete overview of the data provided by Factual.

Attribute	Type	Description
Name	String	Entity name.
Address	String	Address number and street name.
Address_extended	String	Additional address, incl. suite numbers.
Po_box	String	PO Box.
Locality	String	City, town or equivalent.
Neighborhood	String	The neighborhood(s) in which this entity is found.
Region	String	State, province, territory, or equivalent.
Postcode	String	Postcode or equivalent (zipcode in US).
Country	String	
Latitude	Decimal	Latitude in decimal degrees (WGS84 datum).
Longitude	Decimal	Longitude in decimal degrees (WGS84 datum).
Tel	String	Telephone number with local formatting.
Fax	String	Fax number in local formatting.
Website	String	Authority page (official website).
Email	String	Primary contact email address of organization.
Owner	String	Owner name(s).
Cuisine	String	The type of food served.
Price	Integer	A price metric between one and five.
Rating	Decimal	A rating between 1 and 5.
Chain_id	String	Indicates which chain (brand or franchise) this entity is a member of.

Continuation of Table B.1

Attribute	Type	Description
Chain_name	String	Label indicating which chain (brand or franchise) this entity is a member of.
Category_ids	Integer	Category IDs that classify this entity.
Category_labels	String	Category labels that describe the category branch or 'breadcrumb'.
Hours	String	JSON representation of hours of operation.
Hours_display	String	Structured JSON representation of opening hours.
Founded	String	Year founded.
Attire	String	A single value from an enumerated list.
Attire_required	String	Gotta have this on to get in.
Attire_prohibited	String	Can't get in if you are sporting this.
Admin_region	String	Additional sub-division.
Payment_cashonly	Boolean	Only accepts cash
Reservations	Boolean	Accepts reservations.
Open_24hrs	Boolean	Open 24x7
Parking	Boolean	Some kind of parking is advertised; this will be true when any other parking attributes are true
Parking_valet	Boolean	Valet parking is available.
Parking_garage	Boolean	Garage parking is available.
Parking_street	Boolean	Parking on-street.
Parking_lot	Boolean	Parking lot adjacent, not necessarily dedicated to this place.
Parking_validated	Boolean	Validated parking is available.
Parking_free	Boolean	Free parking is available. This is common in most civilized places, but unknown in LA.
Smoking	Boolean	This place allows smoking somewhere.
Breakfast	Boolean	Serves breakfast.
Lunch	Boolean	Serves lunch.
Dinner	Boolean	Serves dinner.
Deliver	Boolean	Delivers.
Takeout	Boolean	Provides takeout/takeaway.
Cater	Boolean	Provides catering.
Alcohol	Boolean	Alcohol is served or can be consumed on the premesis; this will be true when any other alcohol attributes are true.
Alcohol_bar	Boolean	Has a full bar.
Alcohol_beer_wine	Boolean	Serves beer and wine only.
Alcohol_byob	Boolean	Bring Your Own Bottle.
Kids_goodfor	Boolean	Noted as being good for kids.
Kids_menu	Boolean	Has a kids menu.
Groups_goodfor	Boolean	Noted as being good for groups.
Accessible_wheelchair	Boolean	Premesis are noted explictly as being accessible by wheelchair

Continuation of Table B.1		
Attribute	**Type**	**Description**
Seating_outdoor	Boolean	Outdoor seating is available.
Wifi	Boolean	Wifi is provided by the establishment.
Room_private	Boolean	Private dining room is available.
Vegetarian	Boolean	Vegetarian options noted.
Vegan	Boolean	Vegan options noted.
Glutenfree	Boolean	Gluten free items noted.
Lowfat	Boolean	Lowfat options noted.
Organic	Boolean	Organic options noted.
Healthy	Boolean	Healthy dishes are explicitly available.

B.2 Restaurant example

```
{
        "_id" : "1d6ba261-6e02-4fbf-847a-64ed78aca896",
        "wifi" : false,
        "locality" : "New York",
        "meal_deliver" : true,
        "cuisine" : [
                "Italian",
                "Cafe",
                "Pasta",
                "Pizza",
                "American"
        ],
        "meal_lunch" : true,
        "country" : "us",
        "tel" : "(212) 317-2908",
        "region" : "NY",
        "neighborhood" : [
                "Midtown South",
                "Flatiron",
                "Kips Bay"
        ],
        "hours_display" : "Mon-Thu 7:00 AM-11:00 PM;
                           Fri-Sat 7:00 AM-11:30 PM;
                           Sun 7:00 AM-10:00 PM",
        "meal_takeout" : true,
        "latitude" : 40.743903,
        "longitude" : -73.983961,
        "parking_street" : true,
        "parking_lot" : true,
        "meal_dinner" : true,
        "meal_breakfast" : true,
        "alcohol_beer_wine" : true,
        "smoking" : false,
        "alcohol" : true,
        "address" : "420 Park Ave S",
        "price" : 3,
        "options_glutenfree" : true,
        "parking" : true,
        "accessible_wheelchair" : true,
        "attire" : "smart casual",
        "seating_outdoor" : true,
        "hours" : {
                "wednesday" : [
                        [
                                "7:00",
                                "23:00"
                        ]
                ],
                "sunday" : [
                        [
                                "7:00",
                                "22:00"
                        ]
                ],
                "monday" : [
                        [
                                "7:00",
                                "23:00"
                        ]
                ],
                "tuesday" : [
                        [
                                "7:00",
                                "23:00"
                        ]
                ],
                "thursday" : [
                        [
                                "7:00",
                                "23:00"
                        ]
                ],
                "saturday" : [
                        [
                                "7:00",
                                "23:30"
                        ]
                ],
                "friday" : [
                        [
                                "7:00",
                                "23:30"
                        ]
                ]
        },
        "category_labels" : [
                [
                        "Social",
                        "Food and Dining",
                        "Restaurants",
                        "Italian"
                ],
                [
                        "Social",
                        "Food and Dining",
                        "Restaurants",
                        "International"
                ]
        ],
        "postcode" : "10016",
        "category_ids" : [
                358,
                464
        ],
        "room_private" : true,
        "email" : "info@theonerestaurants.com",
        "rating" : 4,
        "name" : "Asellina",
        "open_24hrs" : false,
        "alcohol_bar" : true,
        "groups_goodfor" : true,
        "reservations" : true,
        "payment_cashonly" : false,
        "website" : "http://togrp.com/asellina/"
}
```