Technical University of Crete

School of Electrical and Computer Engineering

Markov Chain Monte Carlo for Effective Personalized Recommendations

By

Michail - Angelos Papilaris

A thesis submitted to the School of Electrical and Computer Engineering in partial fulfilment of the requirements for the Degree of Diploma in Engineering

Abstract

Personalized recommender systems aim to help users access and retrieve relevant information or items from large collections, by automatically finding and suggesting products or services of potential interest. User preferences are difficult to infer, and doing so often requires a tedious elicitation process relying on evidence of others' behavior. To overcome such limitations, we propose a Bayesian approach for finding personalized top recommendations, by capturing user preferences using a utility function which the system learns via a passive preference elicitation sampling-based framework. In brief, instead of asking the user to specify this function explicitly, which is unrealistic, we explicitly model the uncertainty over the utility function and learn it through feedback, in the form of clicks, provided by the user. The utility function is a linear combination of (weighted) features, and beliefs are maintained using a Markov Chain Monte Carlo algorithm. Additionally, we handle situations where not enough data about the user is available, by exploiting the information from clusters of (feature) weight vectors created by observing other users' behavior. Finally, in order to evaluate our system's performance, we applied it in the online hotel booking recommendations domain using a real-world dataset.

Title (Greek)

Ανάπτυξη ενός αποτελεσματικού συστήματος εξατομικευμένων συστάσεων με χρήση Markov Chain Monte Carlo.

Abstract (Greek)

Τα εξατομικευμένα συστήματα συστάσεων, αποσκοπούν να βοηθήσουν τους χρήστες να ανακτήσουν πληροφορίες από μεγάλες συλλογές, εντοπίζοντας και προτείνοντας προϊόντα ή υπηρεσίες δυνητικού ενδιαφέροντος. Συχνά η διαδικασία εξαγωγής προτιμήσεων των χρηστών είναι περίπλοκη και βασίζεται σε στοιχεία για την συμπεριφορά άλλων χρηστών. Για να ξεπεραστούν αυτοί οι περιορισμοί, προτείνουμε μια Μπαεσιανή προσέγγιση για την εξαγωγή εξατομικευμένων προτάσεων, καταγράφοντας παθητικά τις προτιμήσεις των χρηστών, χρησιμοποιώντας μια συνάρτηση χρησιμότητας (utility function) την οποία το σύστημα μαθαίνει. Πιο συγκεκριμένα, αντί να ζητάμε από το χρήστη να καθορίσει την συνάρτηση χρησιμότητας, το οποίο είναι μη ρεαλιστικό, μαθαίνουμε τη συνάρτηση χρησιμότητας παρατηρώντας έμμεσα τον χρήστη και τις επιλογές του (clicks), και συντηρώντας σχετικές (πιθανοτικές) πεποιθήσεις. Η συνάρτηση χρησιμότητας αποτελείται από έναν γραμμικό συνδυασμό (σταθμισμένων) χαρακτηριστικών και οι πεποιθήσεις ενημερώνονται χρησιμοποιώντας έναν αλγόριθμο Markov Chain Monte Carlo. Επιπρόσθετα, σε περιπτώσεις όπου δεν έχουμε συλλέξει αρκετά δεδομένα σχετικά με τον χρήστη, σχηματίσαμε ομάδες (clusters) που δημιουργήσαμε από τα δεδομένα που έχουμε συλλέξει από άλλους χρήστες. Τέλος, προκειμένου να αξιολογήσουμε την απόδοση του συστήματός μας, το εφαρμόσαμε στον τομέα των συστάσεων για ηλεκτρονικές κρατήσεις ξενοδοχείων χρησιμοποιώντας πραγματικά σύνολα δεδομένων (datasets).

Acknowledgements

I would like to express my appreciation to my thesis supervisor, Georgios Chalkiadakis, for his support and guidance towards the completion of my thesis. His guidance helped me immensely during the research and writing of this thesis. I really appreciate his valuable help and understanding, and I am deeply thankful due to the fact that he has always spared his time to discuss and solve problems that were presented during my implementation. Additionally, I would like to thank Michail G. Lagoudakis and Antonios Deligiannakis for their suggestions. Furthermore I would like to express my gratitude to my parents for their continuous support, endless encouragement, and confidence in me, especially during my studies. Many thanks to my friends, who have been very helpful in providing me with moral support throughout my studies and towards the acquisition of my diploma.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

The explosive growth in the amount of digital information available in the Web, along with the enormous numbers of its users have created a potential challenge of information overload which hinders the timely access to online items of interest. This has increased the demand for recommender systems more than ever before. Recommender systems ("RS" or "recommenders" for short) are information filtering systems that deal with the problem of information overload by filtering vital information fragment out of large amounts of dynamically generated information according to user's preferences, interests, or observed behavior about items. An RS has the ability to predict whether a particular user would prefer an item or not, based on the user's profile [1].

Recommender systems are beneficial to both service providers and users. They reduce the transaction costs of finding and selecting items in an online shopping environment. Systems like these have also proved to improve decision making process and quality. In e-commerce settings, recommender systems enhance revenues, and have proven themselves to be effective means of selling more products [2]. In scientific libraries, RS support users by allowing them to move beyond catalogue searches [3]. Therefore, the need to use efficient and accurate recommendation techniques within a system that will provide relevant and dependable recommendations for users cannot be over-emphasized.

Traditionally, RS have been useful in assisting users when they search in large information spaces such as collections of products (movies, books, music CDs, hotels, travelling destinations), documents (news article, medical texts, Wikipedia articles), or users for matchmaking (dating services, online game players/teams, consumer-to-consumer marketplaces). Recommender systems can save a user's efforts in searching for products or services in a variety of ways, for example, by (a) interacting with users in order to reduce uncertainty in their preferences, or (b) recommending a set of results to handle uncertainty from different sources such as user preferences, or ambiguous search queries [2].

Most website that manage large amounts of information such as booking.com or tripadvisor.com, use *hard constraints* (hard constraints are those which we definitely want to be satisfied) on some features, specified by the user, in order to produce suggestions for her.

For example, in a booking recommendation scenario, the user could specify the max price that is willing to pay or the number of the stars that a hotel wants to have and the system would retrieve results that meet those criteria. Unfortunately, this approach has the following practical limitations. First, users often only have a rough idea of what they want in a desirable item. For instance with respect to the cost of an item, they may only specify that “smaller is better”. Thus, hard constraints on a feature may result in either suboptimal recommendations when the budget is set too low, or in a huge number of candidate items when the budget is set too high. Second, the importance of each feature to the user is usually unknown. As an example, for some users the monetary budget may not be so important and they can afford to trade a “reasonable” amount of money for a hotel that provides more amenities or for a hotel that is located closer to the center; other users may be more sensitive to item cost. It is not realistic to expect a user to know, e.g., that they are 80% interested in the overall cost, and 20% interested in the number of the stars that a hotel has [4].

Against this background, in this thesis we intended to build a recommender system that: (a) does not rely on user-specified hard constraints; and (b) does not require an explicit user preferences elicitation process; rather, it learns a user model through passive observation of her actions. In particular, we follow a Bayesian approach that operates as follows. We model the user utility over items as a linear function of the item features, governed by a weight vector W. Each parameter w_i falls in the range of $[-1, 1]$, where a positive (negative) w_i means a larger (resp., smaller) value is preferred for the corresponding feature and our goal is to learn this vector in order to provide personalized recommendations to our user. We capture the uncertainty of the weight vector W, which parameterize the utility function, through a distribution P_w over the space of possible weight vectors. We model this distribution as a Multivariate Normal Distribution. This distribution is often used to describe, at least approximately, any set of (possibly) correlated real-valued random variables each of which clusters around a mean value.

Every time a user enters our system, we propose to her a number of items (hotels in our case) that are complying with the feedback we have received from her in previous interactions. The feedback is in the form of clicks, and each feedback we receive from the user can be translated in n-1 pairwise preferences[1]. After every feedback received, it should be leveraged and update the prior weight distribution through Bayes rule. However, as mentioned in [5], even for the simplest case of the uniform prior, the posterior could be very

[1] Those pairwise preferences could be mentioned as constraints in the next sections.

complex. Thus, we took a different approach. and instead of trying to refit the prior through an algorithm, such as *Expectation Maximization* (*EM*) [6, 7], a task that could prove to be inefficient and time consuming, we decided to keep the constraints gathered from user interactions and the prior distribution, and use a *Markov Chain Monte Carlo* algorithm [8, 9] in order to condition the prior and derive efficiently samples from the P_w posterior distribution. *Markov Chain Monte Carlo* (*MCMC*) techniques estimate by simulation the expectation of a statistic in a complex model. Successive random selections form a Markov chain, the stationary distribution of which is the target distribution. It is particularly useful for the estimation of posterior distributions in complex Bayesian models.

Now, having samples from the posterior distribution of the weight vector and the utility function, we wanted to pick items with the maximum utility and present them to our user as recommendations. Also, an important consideration was to present to the user items that could help us exploit and explore his preferences. A recommender naturally faces the dilemma of recommending items that better match its beliefs about the user, or items that may gradually improve user satisfaction and help to offer better future recommendations. Exploration was aided by the uncertainty inherent in our utility function estimate, and also by presenting random item recommendations and receiving feedback which might be “surprising” given our current knowledge about the user’s preferences. A challenge in picking the best items is that there is no universally accepted ranking semantics given the utility function. In our framework we used the expectation ranking (EXP Ranking) algorithm, which has been adopted widely for ranking items in the preference elicitation subfield [6, 10]. Items that have been ranked higher are presented to the user along with some random items as to explore and exploit his preferences. We capture user preferences by keeping track of clicks on presented items, indicating that the clicked items are more appealing to the user than the unclicked ones.

Finally, a major drawback of many RS that we want to address is handling situations where we have little or no knowledge about the user preferences. A simple and naïve solution to this problem is to show random items to a new user in order to get the initial feedback and update the prior (uninformative) distribution. However, this technique turns out to be very inefficient. We address this issue by constructing clusters from other users’ preferences and finding the most commonly accepted weight vectors. In addition to this, we can use those clusters, when we have no sufficient feedback from our user in order to provide her with recommendations of users with similar interests. For forming the clusters we used the *k*-

means algorithm [11, 12], which partitions n objects into k clusters in which each object belongs to the cluster with the nearest mean.

1.1 Contributions

In this work, we propose a complete model for learning the user preferences and use it to recommend items. It was very important for us to construct a system that will be lightweight and could thus be easily attached in any existing system; while also maintaining the ability to work as an independent, standalone system. We neither set questions to the user, nor use textual information regarding an item so as to elicit user preferences. Moreover, we do not rely on any user ratings of the recommended items. The proposed model, passively observes the user actions, and makes a recommendation based on them.

We propose a system which models user preferences using a linear utility function, governed by a weight vector W that can be described by a probability distribution P_w that captures the user's preferences over the items. Thereto, we use a non-intrusive Bayesian Preference Elicitation (PE) framework for eliciting user feedback on recommended items. Our method is completely personalized, that is, we produce recommendations based only on previous observations of a specific user. We do not use (nor do we have the need of) other users' recommendation data in order to make recommendations to a specific user. We simply observe the behavior of the user through her interaction with the system, and recommend to her items that she likes.

In addition, we take into consideration the problem of having conflicting constraints. This situation could be encountered when the user changes radically his preferences, or if we have gathered too many constraints and we could not find a weight vector that could satisfy all of them at the same time. We overcame this problem by using a linear programming algorithm that finds, effectively and efficiently, if there is, a convex region of all the possible weight vectors that satisfy all the user constraints.

Moreover, we overcame the problem of lacking prior knowledge regarding a user via clustering. Specifically, we used clustering in the user weight vectors gathered from previous user interactions with our system in order to find “popular” weight vectors and suggest items using those weight vectors. We will see in more detail in *Chapter 4* that clustering was instrumental to the reduction of the mean squared error even from the initials interactions, and therefore helped our learning task, without the need to gather too much feedback from the user.

Furthermore, the *MCMC* algorithm that we used for deriving samples of the posterior, needs to start from a point in the n-dimensional space that it has nonzero probability, otherwise the chain gets stuck and does not converge. In a multidimensional space like ours this is a very challenging task. In order to find efficiently and effectively a valid point in the space of feasible weight vectors, we employed *Simplex* [30], a well-known linear programming algorithm that takes into consideration the constraints that we have derived from the user and finds a weight vector that maximizes the utility in order to use it as an initial point in the Markov chain used by MCMC algorithm.

Finally, the real-world dataset that we used for conducting our experiments consists of 5000 hotels, each of which is being characterized by five main features. Many existing works either, have no proof of the models performance [14], or have tested the model's performance with a dataset that is much smaller than ours [15, 16, 17]. Therefore in order to have a reliable and valid way of testing the performance and the effectiveness of our model, we construct 100 users with different preferences derived from various realistic distributions, and the decisions that they made were based on the Euclidean distance between the user preferences and the features of the items presented. Additionally, in order to have a clear picture of our model's performance, for each simulated user we found and stored into our database the top 200 hotels based on the Euclidean distance between the user actual preferences and the hotels features. In this way, at the end of each experiment we were able to compare our model's suggestions with the hotels which she would prefer if he could search all of the hotels in our database (which, especially for a big dataset like ours, is an impractical and time inefficient task).

1.2 Thesis Layout

This thesis is organized into the following chapters: *Chapter 2* reviews related work. *Chapter 3* then provides, a detailed description of the components and algorithms that constitute our model. *Chapter 4* presents our experimental simulations and results. Finally, *Chapter 5* provides conclusions and outlines future work.

Chapter 2

Background and Related Work

Recommender systems have become increasingly popular in recent years, and are utilized in a variety of areas including movies, music, news, books, research articles, search queries, social tags etc. Many different works of recommender systems have been published over the years. In this chapter we provide a brief description of the work that we have mostly relied on, or has influenced our model. Additionally, we are going to give a short background of the most well-known techniques used in recommender systems until to date. We refer the interested reader to [2] for a comprehensive discussion of recommender systems and related techniques.

2.1 Inverse Reinforcement Learning

Our work was to a large extent inspired by work on *inverse reinforcement learning*. The (IRL) problem can be characterized informally as follows (Russell, 1998):
Given:

1. Measurements of an agent's behavior over time in a variety of circumstances.
2. If required measurements of the sensory inputs to the agent.
3. If available, a model of the environment.

The goal of IRL is to learn the reward function used by the user to optimize the long-term cumulative reward. IRL is motivated by situations where learning the reward function governing an MDP domain is a goal on its own.

An MDP, i.e., a *Markov Decision Process* [18], provides a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker. At each time step, the process is in some state *s*, and the decision maker may choose any action *a* that is available in state *s*. The process responds at the next time step by randomly moving into a new state *s′*, and giving the decision maker a corresponding reward $R_a(s, s')$. The probability that the process moves into its new state s′ is influenced by the chosen action. Specifically, it is given by the state transition function $P_a(s, s')$. Thus, the next state *s′* depends on the current state *s* and the decision maker's action *a*. But given *s* and *a*, it is conditionally independent of all previous states and actions; in other words, the state transitions of an MDP possess the Markov property.

Finally, a Markov decision process is a 5-tuple (*S*, *A*, *T*, *R*, *γ*) composed of:

i. A (finite) set *S* of states
ii. A (finite) set *A* of actions
iii. (Markov) transition function *T(s, a, s′) = Pr(s′* | *s, a)*, specifying the probability *Pr(s′* | *s, a)* of going to state *s′* after taking action *a* in state *s*
iv. Reward function *R(s)* (or *R_a(s,s')* if reward depends also on a), determining the immediate reward received after transition to state *s′* from state *s*.
v. *γ* ∈ [0, 1] is the discount factor, which represents the difference in importance between future rewards and present rewards

Against this background the usual *reinforcement learning (RL)* problem is concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward, while the goal of IRL is to learn the model's reward function. For instance, the first IRL approach [19], views the agent's decisions as a set of linear constraints on the space of possible utility (reward) functions. The Bayesian IRL approach of [5], on the other hand, assumed that a distribution over possible reword functions exists and has to be inferred. In our case, the goal is to learn the utility function that each item has for a specific user. We model this utility as being linearly additive, governed by a weight vector. Conceptually, therefore, our approach is thus similar to IRL, however, due to the fact that the problem we are facing is not a sequential decision problem, but rather one-step decisions we do not use an MDP to model it.

2.2 Related Work

One interesting work is that of Chajewska, Koller and Ormoneit on *Learning an Agent's Utility Function by Observing Behavior* [9]. In that work, they consider the task of predicting the future decisions of an agent based on her past decisions. To account for the uncertainty regarding the agent's utility function, they consider the utility to be a random quantity that is governed by a prior probability distribution. When a new agent is encountered, this estimate serves as a prior distribution over his utility function. As they observe his behavior, they use the constraints implied by his choice to condition this distribution, obtaining a posterior distribution through MCMC algorithm.

The work most relevant to ours however, is perhaps that of Xie et al [14] who propose an alternative approach for finding personalized top-k packages for users, by capturing users' preferences over packages using a linear utility function which the system learns. Instead of asking a user to specify this function explicitly, they model the uncertainty in the utility function and propose a preference elicitation-based framework for learning the utility function through feedback provided by the user. In their work they propose several sampling based methods which can capture the updated utility function, given user feedback. Also they develop an efficient algorithm for generating top-k packages using the learned utility function, where the rank ordering respects any of a variety of ranking semantics proposed in the literature.

2.2.1 Popular Techniques Used in RS

The most used techniques in recommendation systems are the collaborative filtering (CF), content-based, or hybrid methods that attempt to eliminate the weaknesses of the individual approaches. Additionally, some approaches [10, 14] use preference elicitation (PE) to model the user preferences using a utility function. As mention earlier, there has been a lot of research in this domain, thus in this section we are simply going to describe the most popular methods that have been used in recommender systems, and also briefly mention some typical works representative of each method category.

2.2.2 Collaborative Filtering

Collaborative filtering (CF) is the most widely implemented and most mature technology available in the market. Collaborative recommender systems aggregate ratings or recommendations of objects, recognize commonalities between the users on the basis of their ratings, and generate new recommendations based on inter-user comparisons [20, 21]. The greatest strength of collaborative techniques is that they are completely independent of any machine-readable representation of the objects being recommended and work well for complex objects where variations in taste are responsible for much of the variation in preferences. Collaborative filtering is based on the assumption that people who agreed in the past will agree in the future and that they will like similar kind of objects as they liked in the past.

Matrix Factorization (MF) plays an important role in a CF-based recommender system. MF has recently received greater exposure, mainly as an unsupervised learning method for latent variable decomposition and dimensionality reduction [22, 23]. Prediction of ratings and recommendations can be obtained by a wide range of algorithms, while neighborhood-based CF methods are simple and intuitive. The Matrix Factorization techniques are usually more effective because they allow the discovery of the latent features underlying the interactions between users and items.

2.2.3 Content based methods

Another common approach when designing recommender systems is content-based filtering. Content-based filtering methods are based on a description of the item and a profile of the user’s preference [1, 24, 25]. In a content-based recommender system, keywords are used to describe the items and a user profile is built to indicate the type of item this user likes. In other words, these algorithms try to recommend items that are similar to those that a user liked in the past (or is examining in the present). In particular, various candidate items are compared with items previously rated by the user and the best-matching items are recommended. Content-based methods arguably constitute an outgrowth and continuation of information filtering research. In such a system, the objects are mainly defined by their associated features. A content-based recommender learns a profile of the new user’s interests based on the features present, in objects the user has rated.

2.2.4 Preference elicitation

Preference elicitation (PE) has been studied extensively in the multi-agent systems (MAS) community [6, 10]. The general idea of PE is to model users' preferences using a utility function, and then learn the parameters of this utility function through user feedback. To do so, most PE techniques set queries to the user asking her to evaluate, order, or constrain potential system outcomes, while others try to translate a user's interaction with the system to preferences. The goal of a preference elicitation recommender system is to make suggestions that maximize the user’s utility function.

In the work of [10], Chajewska et al. propose an approach that uses a prior probability over the user's utility function, assuming she is influenced by similar users. After that, they select queries for the utility elicitation based on the VPI measure of each query, in order to maximize user utility. Providing gambling queries to the user in order to take feedback is a very common thing in Preference Elicitation (PE) technique, and this work constitutes a representative example of the PE domain.

In the work of [26], Lieberman has implemented a recommendation system that uses Preference Elicitation technique in the domain of rental real estates. More specifically, the users inputs the features of the apartment that they were looking for and the system suggest a number of apartments. The user is able to rate if and how much satisfied she is about each feature, so that the system can retrieve and prune the suggestions based on the user rating on each feature.

2.2.5 Hybrid Methods

Combining any of the two techniques in a manner that suits a particular industry is known as Hybrid Recommender system. This is the most sort after Recommender system that many companies look after, as it combines the strengths of more than two recommender system techniques and also eliminates any weakness which exist when only one techniques is used. There are several ways in which the systems can be combined. Recent research has demonstrated that a hybrid approach, combining collaborative filtering and content-based filtering could be more effective in some cases [15, 16, 27]. Hybrid approaches can be implemented in several ways: by making content-based and collaborative-based predictions separately and then combining them, by adding content-based capabilities to a collaborative-based approach (and vice versa), or by unifying the approaches into one model. Several studies empirically compare the performance of the hybrid with the pure collaborative and content-based methods and demonstrate that the hybrid methods can provide more accurate recommendations than pure approaches. These methods can also be used to overcome some of the common problems in recommender systems such as cold start and the sparsity problem.

A work that combines preference elicitation with collaborating filtering is the work of McNee et al. that compares three interface strategies for eliciting movie ratings from new users [17]. In the first strategy, the system asks the user to rate movies that were chosen based on entropy comparisons to obtain a maximally informative preference model. That is, the system decides which movies users should initially rate. In another strategy, users were allowed to freely propose movies they wanted to rate. In a mixed strategy, the user had both possibilities. A total of 225 new users participated in the experiment which found that the user-controlled strategy obtained the best recommendation accuracy compared to the other two strategies in spite of a lower number of ratings completed by each user.

In work of Babas, Chalkiadakis and Tripolitakis [28], essentially a content-based Bayesian approach that also employs user feedback in the form of ratings of recommender items. More specifically they model the types of both the user and the object under recommendation as multivariate Gaussian distributions. Then, they make use of Normal-inverse Wishart priors to model the recommendation agent beliefs about user types. By means of this process, the agent is able to recommend to a user items that best fit both her long-time preferences and current mood. To achieve its objective, the agent maintains item types and user types (corresponding to modeled user preferences).

Chapter 3

Our System

In this chapter we will provide a detailed description of our recommendation system - and the mathematics behind its operation. An overview of our model appears in Figure 1. There, we see that our system works as follows: When a user enters in our system, we retrieve the constraints derived from her previous interactions and we check if there are any conflicting constraints in order to handle them. Next we update our beliefs by conditioning the prior distribution with the aforementioned constrains. Subsequently we rank our items according to the posterior samples and present the top N items to the user. We record which of those items are being selected (clicked) by the user, and we assume that clicked items are more appealing to her that the un-clicked ones. We store in our database the new constraints derived from the feedback received, and we progress to the next interaction. Finally, we form clusters with the posterior samples from the registered users on our system and we use them with the new users in order to limit the initial uncertainty that we have for them.

Given this, the rest of this chapter is organized as follows. Section 3.1 provides the setting for our model, describing notation and model setup choices (such as decisions relating to dataset preprocessing), and, importantly, our assumptions regarding the form of a user's utility function. Then, the rest of the sections in this chapter describe our model components in turn: Section 3.2 discusses the derivation of constraints; Section 3.3 describes our approach to beliefs' updating; Section 3.4 describes our ranking algorithm; and finally section 3.5 describes our use of clustering in this work.

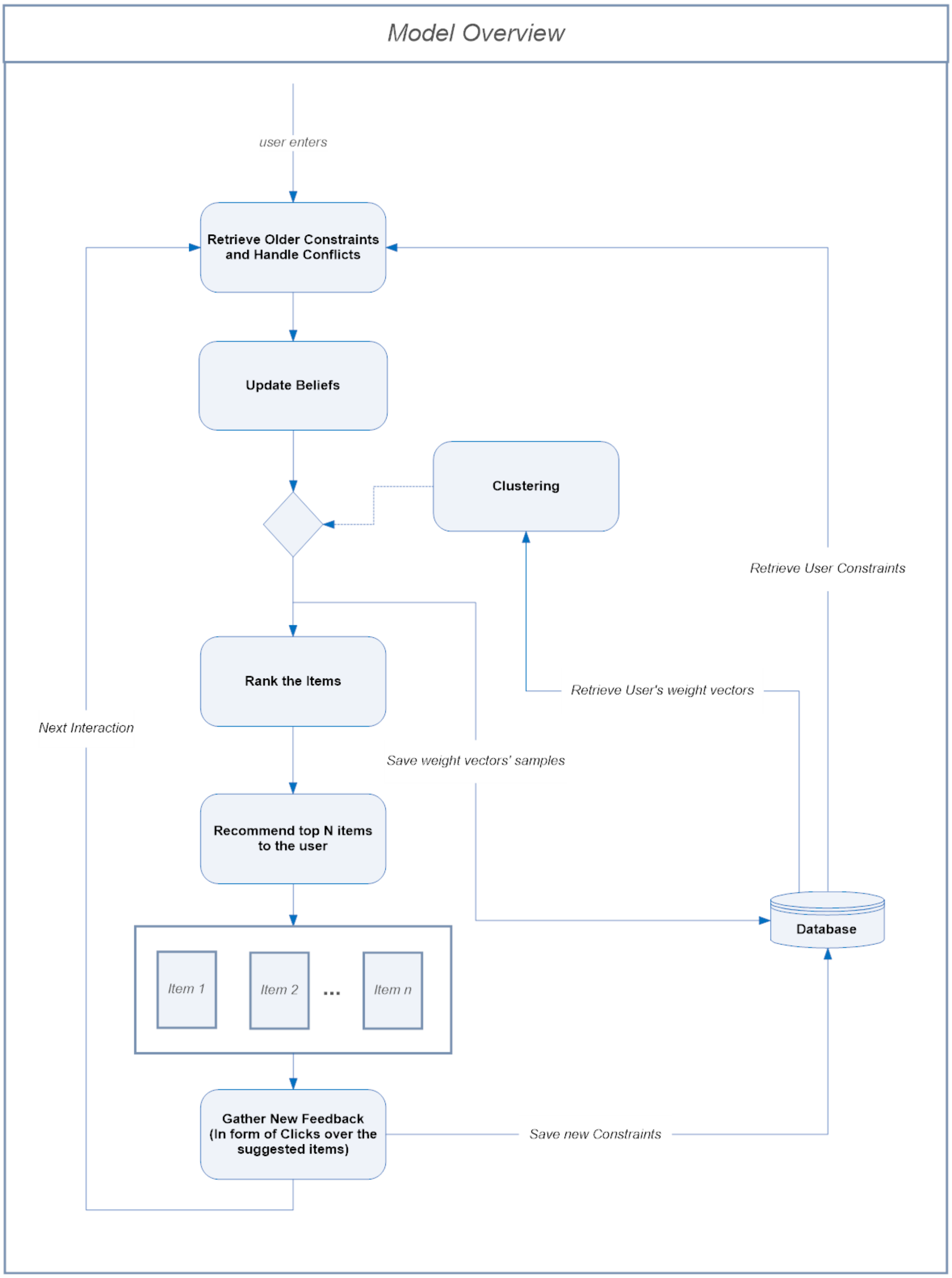

Model Overview
user enters
Retrieve Older Constraints and Handle Conflicts
Update Beliefs
Clustering
Retrieve User Constraints
Rank the Items
Retrieve User's weight vectors
Next Interaction
Save weight vectors' samples
Recommend top N items to the user
Database
Item 1
Item 2
...
Item n
Gather New Feedback (In form of Clicks over the suggested items)
Save new Constraints

Figure 1: *Model Overview*

3.1 Model Settings

In this section we are going to explain some of the notation and symbols that we will be using in the following sections. Also we are going to describe in detail the composition of the utility function and also how the utility function is being parameterized with a weight vector W that we want to learn.

3.1.1 Symbols and Notation

Item Set S_I: We assume that we are given a set of n items.

Feature Vector F: Each item is being described by m features (f_1, f_2, … , f_m). For example, a feature of an item could be the price, rating etc. For simplicity, when no ambiguity arises, we use vector F to denote both an item and its corresponding feature vector.

Value Vector V: Every item i ∈ I is being described by an m-dimensional vector V {v_1, v_2, …, v_m}, where v_j ∈ V corresponds to the value of the respective feature f_j. Without loss of generality, we assume all feature values are non-negative real numbers in [0, 1].

Weight Vector W: This is being described by an m-dimensional vector {w_1, w_2,… , w_m}, where w_i describes the degree of appreciation of the respective feature f_i by the user. This vectors parameterizes the utility function of each item, and each w_i is in the range of [-1, 1].

Utility Function U(i)**:** For an item i, U(i) is calculated as a linear combination of the V vector multiplied by the weight vector W. The utility function essentially defines a total order over all items.

Constraint Set $S_{C(k)}$: For every user (k) we have a *Constraint* set that is based on the feedback that we have received from her. Every constraint C_i has the following form: $i_k > i_t$ where i_k, i_t are items in the item set S_I.

3.1.2 Dataset Preprocessing

In order to produce targeted recommendation to a user, our proposed model needs all of our items $i \in S_I$ in our database to be normalized. To achieve that, we found the maximum and the minimum values of each item feature, and then for each item feature value $v_i \in V$ we used the following formula in order to normalize our values.

$$v_i' = \frac{V_i - \min(v)}{\max(v) - \min(v)} \quad (1)$$

where $v_i \in V$, $\min(v)$ and $\max(v)$ are the minimum and maximum values of the respective feature in the item set S_I. In our case the values that each hotel feature could take were nonnegative (price, stars, distance from the town center, ratings, amenities) before the normalization. For some item features, the range of values that could take was known in advance. For example, the number of the stars that a hotel could have was ranging from zero to five. On the other hand, for some other features such as the price per day that a hotel could have, we have to find the maximum and the minimum value in the collection in order to normalize it. After the normalization process all items' features in our collection lie in the [0, 1] interval.

3.1.3 Weight Vector

For every user the utility function of an item is parameterized by a weight vector W. The length of the weight vector W is equal to the respective number of the item's features. Our final goal is to learn this vector and suggest items to the user that have the maximum utility taking into consideration this weight vector. Each element w_i of the weight vector, could take values in the range of [-1, 1]. A negative weight (w_i) means that the respective value v_i of the feature is disliked by the user, a positive value means that the value which corresponds to the respective feature is preferred, and finally if the weight is equal to zero means that the user is indifferent about the specific feature. The weight vector W is not known in advance, but it is described by a probability distribution P_w. We model this distribution as a Multivariate Normal Distribution as in the work of (Babas, Chalkiadakis, Tripolitakis) [28]. This distribution is often used to describe, at least approximately, any set of (possibly) correlated real-valued random variables, each of which clusters around a mean value. The P_w can be learned by the feedback received from the user, as we describe in section 3.3 below.

3.1.4 Utility Function

The user-specific utility function U over items $i \in S_I$, that we wish to learn, directly depends on its feature vector. The space of all mappings between possible combinations of the feature values and utility values is uncountable, making this task challenging. Since we need to express the structure of an item concisely, we assume that the utility function is linearly additive and governed by a weight vector like the one described in the previous subsection. This is a structure for the utility function commonly assumed in practice [9, 29]. For each item, the utility function is being calculated by the following formula.

$$U(i) = \sum_{j=1}^{n} w_j \cdot f_j \quad (2)$$

where f_j is the *value* of the respective feature, and w_j is the weight value associated with this feature. The value for n depends on the number of features that each item has. In our experiments we used a dataset that consisted of hotels with five main features each, price, stars, ratings, distance from the city center and number of amenities.

Users often only have a rough idea of what they want in a desirable item, and also find such preference very difficult to quantify. For instance, w.r.t. the cost of an item, they may only specify that “lower is better”. For this reason, users are not able to specify or even know the exact values of the weight w_i that drive utility function U. We model this uncertainty in a Bayesian manner, assuming that W is not known in advance, but it can be described by a probability distribution P_w.

3.2 Constraints

Our aim is to create a lightweight model that could be easily integrated into any existing application. More specifically we want our model to be able to capture and update the user's preferences without any disruption to the user end. This was achieved by making our system a passive observer of the user's behavior. We wanted to avoid what most preference elicitation methods do, which is setting queries to the user asking her to evaluate, order, or constrain potential system outcomes [10]. In order for our model to be a passive observer and capture the user preferences, we record the clicks the user made to the items suggested to her. More specifically, our system suggests to the user a number of items, and we record which of them are being selected (clicked on) by the user. We make the assumption that clicked items are more appealing to the user than the un-clicked ones. Thus, the feedback in form of new clicks, produces a set of pairwise preferences in the form of i1 > i2, where i1 and i2 are the suggested Items.

In each iteration we present to the user n items. Assuming the user prefers the $k \in [1, n]$ item, then that preference is expressed with (n – 1) pairwise set of preferences: [2]

$$\textit{Item } k > \textit{Item } i, \ \forall\, i \in [0, k) \cup (k, n] \quad (3)$$

In order to make clear how the constraints work, we present an example. Assume that we present to the user three items, (i1, i2, i3), and each of the items has 3 features (f1, f2, f3). If a user selects the second item, then we derive the following pairwise preferences: i2 > i1 (4) and i2 > i3 (5). Then, Eq. (4) corresponds to a constraint that we use as follows.

$$i2 > i1 \quad \Rightarrow$$

$$U(i2) > U(i1) \quad \Rightarrow$$

$$\sum_{j=1}^{n} wj \cdot f_2 j > \sum_{j=1}^{n} wj \cdot f_1 j \quad \Rightarrow$$

$$w1 * f1_{Item2} + w2 * f2_{Item2} + w3 * f3_{Item2} > w1 * f1_{Item1} + w2 * f2_{Item1} + w3 * f3_{Item1} \quad (6)$$

[2] Also mentioned as Constraints

Working in the same way as above, for (5), we set:

$$i2 > i3 \quad \Rightarrow$$

$$U(i2) > U(i3) \quad \Rightarrow$$

$$\sum_{j=1}^{n} wj \cdot f_2j > \sum_{j=1}^{n} wj \cdot f_3j \Rightarrow$$

$$w1*f1_{Item2} + w2*f2_{Item2} + w3*f3_{Item2} > w1*f1_{Item3} + w2*f2_{Item3} + w3*f3_{Item3} \quad (7)$$

In equations (6), (7) the feature values that each item has are known in advance. We can use the two following lemmas, found in [14], which highlight some characteristics that govern the feedback received.

Every feedback received in terms of clicks, for example, when a user clicks on the first item of the two that were suggested to her, rules out all the weight vectors that do not satisfy the item1 > item2 constrain.

Lemma 1 [14]: Given Feedback $item_1 > item_2$: (8) if $item_1 > item_2$ does not hold under the weight vector w, $P_w(w \mid item_1 > item_2) = 0$; (9) for two weight vectors w_1, w_2, $w1 \neq w2$, if $item_1 > item_2$ holds under both w_1, w_2, and $P_w(w_1) > P_w(w_2)$, we have $P_w(w_1|item_1>item_2) > P_w(w_2|item_1>item_2)$.

Proof: *(1) Consider the likelihood $P(item_1 > item_2 \mid w)$ in equation (9). If $item_1 > item_2$ does not hold under w, clearly the likelihood of $item_1 > item_2$ under w is 0, so $P_w(w|item_1>item_2)=0$. (9) Consider $P_w(w_1|item_1>item_2)$ and $P_w(w_2|item_1>item_2)$. Since $item_1>item_2$ holds under both w_1, w_2, $P_w(item_1>item_2 \mid w_1) = P_w(item_1>item_2 \mid w_2) = 1$, so* $\frac{Pw(w1 \mid Item1>Item2)}{Pw(w2 \mid Item1>Item2)} = \frac{Pw(w1)}{Pw(w2)}$ 。

Lemma 2 [14]: The set of valid weight vectors which satisfy a set of preferences forms a convex set. So valid weight vectors form a continuous and convex region.

Proof: *By definition, for any w_1, w_2 which satisfy our set of constraints S_c, $\forall\, i := item1 > item2 \in S_c$, $w_1 * item_1 \geq w_1 * item2$ and $w_2 * item_1 \geq w_2 * item_2$. Then $\forall \alpha \in [0, 1]$, $\alpha \cdot w_1 \cdot item_1$*

$\geq \alpha \cdot w_1 \cdot item_2$ and $(1 - \alpha)w_2 \cdot item_1 \geq (1 - \alpha)w_2 \cdot item_2$. Combining these inequalities shows that any convex combination of w_1 and w_2 also forms a valid w. ∘

Therefore we search to find the convex region in the weight vector distribution P_w that could satisfy both constraints (6), (7). We will see that there are instances where this convex region is very small or even that does not exists. In the next section, we will describe how we handle situations like the aforementioned.

The feedback can be used in order to update our posterior probability distribution P_w through Bayes' rule as follows:

$$P_w\,(w|\;Item\;k > Item\;i\;) = \frac{P(Item\;k{>}Item\;i\;|w)\;Pw(w)}{\int_w P(Item\;k{>}Item\;i\;|w)\;Pw(w)\;dw}, \forall\; i \in [0,k) \cup (k,n] \quad (10)$$

Where $P(Item\;k > Item\;i\;|\;w)$ defines the likelihood of $Item\;k > Item\;i$ given w. However, the cost of refitting through an algorithm such as *Expectation Maximization* (*EM*) [6, 7] is extremely high, so we take a different approach: that of representing the posterior by maintaining both the prior distribution and the set of feedback preferences received. Even if P_w is a "nice" distribution with a compact closed form representation, the posterior can be quite complex [9]. The problem is that irregular polytopes (even convex ones) are computationally difficult to deal with. Thus, we use *Markov Chain Monte Carlo* techniques to generate a set of samples from the posterior distribution in an efficient fashion.

3.2.1 Handle Conflicting Constraints

As mentioned earlier, with every interaction with our system we derive a number of constraints that is in the following form:

$$Item\;k > Item\;i\;\;\forall\; i \in [0, k\;) \cup (\;k, n] \quad (11)$$

A major challenge that we encountered was that, after a few interactions with our system, some of the constraints that we have gathered may have conflicts with the new ones. This incident could happen mainly in two occasions: (a) when the user changed radically his preferences; or (b) if we have gathered too many constraints, and could not find a weight vector that could satisfy all of them at the same time. In order to deal with this problem we used the well-known linear programming algorithm, called *Simplex* [30].

It was the one of the first algorithms for solving linear programs, invented in the 1940's by George Dantzig [30]. The simplex method is very efficient in practice, has polynomial-time average-case complexity [31]. However, its worst-case complexity is exponential, as can be demonstrated with carefully constructed examples [13]. The Simplex algorithm can solve any kind of linear program, but it only accepts a specific form of the program as input. So first we have to do some manipulations.

The following algorithm outlines the basic steps of Simplex algorithm:

Algorithm 1: *Simplex algorithm basic steps [30]*

```
Step 1:    Write the linear programming problem in standard form

Step 2:    Write the coefficients of the problem into a simplex tableau

Step 3:    Gaussian elimination

Step 4:    Choose new basic variables

Step 5:    Read off the solution
```

We used the Simplex algorithm in order to find whether a feasible weight vector[3] is available. Due to the fact that the specific algorithm has the limitation that all the variables should be nonnegative, and each w_i in the weight vector could take values in the range of [-1, 1], we could not just run Simplex with an unrestricted variable, because it won't set w_i to a negative value. The solution to this problem was to replace each w_i in the linear equations that feed the simplex algorithm with two variables:

[3] We consider a weight vector to be "feasible" (or "valid") if it satisfies all the available constraints we have gathered from previous interactions.

$$w_i = w_i' - w_i'' \qquad w_i', w_i'' \geq 0 \quad (12)$$

In the case that Simplex returns that there is no feasible weight vector which could satisfy all constraints, we remove the oldest constraint we have gathered and we rerun the algorithm. We repeat the same procedure until we find a valid weight vector. We observed empirically that this method of removing the old constraints that were conflicting with the new ones led to a recommender that was more flexible and “responsive” to potential radical and sudden changes of a user’s preferences.

3.3 Beliefs’ updating

When performing Bayesian inference, we aim to compute and use the full posterior joint distribution over a set of random variables. Unfortunately, this often requires calculating intractable integrals. Even if the prior distribution is a “nice” distribution with a compact closed form representation, the posterior distribution can be quite complex [5, 9]. The problem is that irregular polytopes (even convex ones) are computationally difficult to deal with. Refitting the weight distribution P_w after each feedback that we receive from the user is a very inefficient and time consuming task for a real time recommendation system. For example, the cost of refitting through an algorithm such as *expectation maximization (EM)* [6, 7] is extremely high [32, 33], so we take a different approach. To circumvent this, we propose a sampling based framework which obviates the need for a posterior. Instead, items preferences resulting from user feedback can be translated into constraints on the samples drawn from the posterior P_w. In our approach we follow a technique that also is used in the work of [9] and [14]: we condition the weight prior distribution, which we model as *Multivariate Normal Distribution*, with the constraints that we have derived from each user in order to acquire samples from the posterior. We used *Markov Chain Monte Carlo* (MCMC) algorithm [8, 9] to generate a set of samples from the posterior distribution in an efficient fashion. MCMC is suitable for our model because it can easily handle cases with higher dimensionality. When using MCMC methods, we estimate the posterior distribution and the intractable integrals using simulated samples from the posterior distribution. In our case, in order to take samples from the W posterior distribution, we used the MCMC *Metropolis Hastings (MH)* algorithm [34, 35], with some additional optimizations.

3.3.1 Setting the ground for our MCMC algorithm

In case the amount of the feedback received is small, simple sampling techniques like the popular rejection sampling [36] could be effective. But as the feedback increases, those sampling methods prove to be inefficient. As stated in *Lemma 2*, the weight vectors that do not violate any of the constraints that we have gathered, form a convex region. Thus, as feedback increases, this convex region drastically shrinks. By using a simple rejection sampling technique, it is more likely the random sample we drew to be rejected because it is very likely to be outside of the convex region. As mentioned in section 3.2 section, in every interaction, we store (n – 1) constraints, where n is the number of suggestions made to the user in each interaction. Sampling Techniques like *Markov Chain Monte Carlo* are more suitable for cases like this, because they are "aware" of the feedback received, and can handle cases with higher dimensionality.

A major problem encountered in the sampling process was that in order for our MCMC algorithm to start collecting samples from the posterior distribution, it needs to start from a point that does not violate[4] the feedback received, and thus has non zero probability. Otherwise, if we start iterating from a point that has zero probability, the Markov chain remains stacked to the initial point. Additionally, in cases that we have received a lot of feedback from a user, searching randomly to find a weight vector inside the convex region in order to use it as a starting point in the MCMC proved to be an inefficient and time consuming task. Thus, we used *Simplex*. With *Simplex* we were able to find a starting point that was located inside the convex region and thus has non zero probability, which was necessary in order to iterate and start collecting samples from the weigh distribution posterior.

Before MCMC traverses the convex region based on the distribution P_w, we first construct a regular grid in the m-dimensional hypercube, where m is the number of the features that each item has. Each w_i of the weight vector W takes values in the range of [-1,

[4] I.e., one located inside the convex region that the valid weight vectors forms.

1]. So our hypercube is $[-1, 1]^m$. Our grid interval[5] is set to 0.02, so each dimension is divided into 100 intervals. After initiating the algorithm from the starting point, produced by *Simplex,* we carry out the sequence of *MCMC* steps, prescribed by the *MH* algorithm described below.

3.3.2 Metropolis Hastings Algorithm

The *MH* algorithm [35] simulates samples from a probability distribution by making use of the full joint density function and (independent) proposal distributions for each of the variables of interest. Algorithm 2 provides the details of a generic MH algorithm

Algorithm 2: *Metropolis Hastings algorithm [37]*

```
Initialize $x(0) \sim q(x)$
for iteration $i = 1, 2, \dots,$ number of samples do
    Propose: $x^{cand} \sim q(x^{(i)}|x^{(i-1)})$
    Acceptance Probability:
        $\alpha(x^{cand}|x^{(i-1)}) = \min\{ 1, \frac{q(x^{(i-1)}|x^{cand})\,\pi(x^{cand})}{q(x^{cand}|\,x^{(i-1)})\,\pi(x^{(i-1)})}\}$, where $x^{cand}$is the candidate
        sample and $x^{(i-1)}$is the previous sample
    $U \sim$ Uniform $(u; 0, 1)$
    if $(u < \alpha)$ then
        Accept the proposal: $x^{(i)} \leftarrow x^{cand}$ , where $x^{(i)}$ is the current sample and $x^{cand}$ is
        the candidate sample.
    else
        Reject the proposal: $x^{(i)} \leftarrow x^{(i-1)}$ , where $x^{(i)}$ is the current sample and $x^{(i-1)}$ is the
        previous sample.
    end if
end for
```

[5] Also known and as “Step of the Grid”.

The first step is to initialize the sample value for each random variable (this value is often sampled from the variable's prior distribution). The main loop of *Algorithm 2* consists of three components:

1. Generate a proposal (or a candidate) sample x^{cand}.

2. Compute the acceptance probability via the acceptance function based upon the candidates' distribution (usually mentioned as proposal distribution) and the full joint density $\pi(\cdot)$.

3. Accept the candidate sample with probability α, the acceptance probability, or reject it with probability $1 - \alpha$.

For step 1 above, the idea is to define a Markov chain over possible x values, in such a way that the stationary distribution of the Markov chain is in fact P(x). Thus, we use a Markov chain to generate a sequence of x values in such a way that as $n \rightarrow \infty$, we can guarantee that $x_n \sim P(x)$ [38]. The algorithm starts with simulating a "candidate" sample x^{cand} from the candidates' distribution (proposal distribution) $q(\cdot)$.

There are mainly two kinds of proposal distributions, symmetric and asymmetric. A proposal distribution is a symmetric distribution if:

$$q(x(i) \mid x(i-1)) = q(x(i-1) \mid x(i)) \quad (13)$$

Straightforward choices of symmetric proposals include Gaussian distributions or Uniform distributions centered at the current state of the chain. For example, if we have a Gaussian proposal, then we have $x^{cand} = x\ (i-1) + \text{Normal}\ (0, \sigma)$. Since the PDF for $\text{Normal}\ (x^{cand} - x(i-1); 0, \sigma) = \text{Normal}(x(i-1) - x^{cand}; 0, \sigma)$, this is a symmetric proposal. This proposal distribution randomly perturbs the current state of the chain, and then either accepts or rejects the perturbed value. Algorithms of this form are called "Random-walk Metropolis algorithm." Random-walk MH algorithms are the most common MH algorithms. For our needs, we choose a candidate as follows. With probability 0.5 we keep the old sample x(i-1), and with probability 1 - 0.5 = 0.5, x^{cand} is chosen uniformly from among x(i-1)'s 2m neighbors. Remember that we have constructed a regular grid in the m-dimensional hypercube, where m is the number of the features of the items.

Here we choose to work with a symmetric proposal distribution, because it makes the algorithm more straightforward, both conceptually and computationally. As shown in Algorithm 3, we perform a random walk. After choosing (uniformly) the neighbor which that we will visit, then move forward with 50% probability; and move backwards with 50% probability also. After we have choose the direction and if we are going to move forward or backward, we choose the number of steps that we are going to move in the grid. With 0.9 probability we move only one step (as we have mention the step interval is 0.02[6]). With probability 0.1 we choose to move with equal probability either 2*step, 3*step or 4*step. From our experiments we observe that making, with small probability, relative bigger "jumps" in the grid, helped in preventing the Markov chain from getting stuck in a particular part of the distribution. A good proposal distribution can make a huge difference. However, it is also important to say that even if the proposal distribution is "bad", one can always solve the problem with brute force (more interactions).

Algorithm 3: *Choosing a candidate Sample*

```
//we have already choose the direction (the neighbor) that we are going move.

u ~ Uniform (u; 0, 1)
if (u < 0.5) then  //with 50% prob we keep the old sample and with 50% we choose to move.
    u1 ~ Uniform (u; 0,1)
    if (u1 < 0.5) then //with 50% prob we move forward.
        u2 ~ Uniform (u; 0,1)
        if (u2 < 0.9) then //we choose to do one step.
            grid_step = 0.02;
        else //we choose to do more steps in the grid (jump).
            u3 ~ Uniform (u; 2,4)
            grid_step = u3*0.02;
```

[6] Thus, we add or subtract one step in the direction we chose (e.g. we add or subtract the step in the price value).

```
        end if
    else //with the remaining 50% prob we move backward.
        u2 ~ Uniform (u; 0,1)
        if (u2 < 0.9) then //we choose to do one step.
            grid_step = - 0.02;
        else //we choose to do more steps in the grid (jump).
            u3 ~ Uniform (u; 2,4)
            grid_step = - u3*0.02;
        end if
    end if
    u2 ~ Uniform (u; 0,m) (where m is the number of the features of the items)
    $x^{cand}$ = x(i-1).get_u2_feature + grid_step;
else //with 50% prob we keep the old sample.
    $x^{cand}$ = x(i-1);
end if
```

The next step after choosing the candidate sample x^{cand} , is to check if the x^{cand} satisfy all the constraints we have gathered from the feedback of the user. If this sample violates one or more constraints then we reject it and we keep the old sample, otherwise we decide whether to keep it based on the acceptance function.

Intuitively, the MH acceptance function is designed to strike a balance between the following two constraints [37]:

1. The sampler should tend to visit higher probability areas under the full joint density; this constraint is given by the ratio $\frac{\pi(Xcand)}{\pi(x(i-1))}$.
2. The sampler should explore the space and avoid getting stuck at one site (e.g., the sampler can reverse its previous move in the space; this constraint is given by the ratio: $\frac{q(x(i-1)\mid Xcand)}{q(Xcand\mid x(i-1))}$.

It is important that the MH acceptance function has this particular form, because that ensures the algorithm satisfies the condition of “detailed balance”, which guarantees that the stationary distribution of the MH algorithm is in fact the target posterior that we are interested in [38]. In cases such as our own, when we have a symmetric proposal, the acceptance function could be simplified as demonstrated below [38]:

$$\alpha(x(i)|x(i\text{-}1)) = min\{1, \frac{q(x(i-1)|x(i))\,\pi(x(i))}{q(x(i)|x(i-1))\,\pi(x(i-1))}\} = min\,\{1, \frac{\pi(x(i))}{\pi(x(i-1))}\} \quad (14)$$

Where π(·) is the full joint density. This result is intuitive: when the proposal distribution is symmetric (q(x(i) | x(i−1)) = q(x(i−1) | x(i))), the acceptance probability *a* becomes proportional to how likely each of the current state x(i−1) and the proposed state x(i) are under the full joint density. As stated, we accept a given proposal with acceptance probability α, which is the outcome of the acceptance function described above. The min operator in the acceptance function makes sure that *α* is never larger than 1. Operationally, we draw a random number uniformly between 0 and 1, and if this value is smaller than *α*, we accept the candidate; otherwise we reject it.

The amount of time it takes to converge to the chain's stationary distribution is called the "mixing time" or "burn in" time. Once the chain has mixed, it is "safe" to start collecting samples. For example, if we start from a "bad"[7] location in the distribution, the sampler spends the first n iterations to slowly move towards the main body of the distribution. That is why we have added a "burn in" phase to our algorithm. The "burn in" value that we have set is 3000 iterations. Thus, after we assign value to the first sample from the Simplex algorithm, we run 3000 iterations of the MCMC algorithm without collecting actual samples, in order to move closer to a point with higher probability in the distribution. After we start collecting samples from the posterior, to avoid collecting samples that are highly correlated, it is common to pick a subset of them. So we introduce a "lag" parameter, set to 100 iterations. This means that we keep one sample after every 100 iterations.

3.4 Ranking the Items

The ultimate goal of a recommender system is to suggest a short ranked list of items, namely top-n recommendations, supposed to be the most appealing for the end user. The system

[7] One with low probability.

focuses first on predicting the unknown ratings which are eventually used to generate a ranked recommendation list. Actually, the top-n recommendation task can be directly seen as a ranking problem, where the main goal is not to accurately predict ratings, but directly find the best ranked list of items to recommend. A framework based on utility function, like ours, essentially defines a total order over all items. We assume ties in utility score are resolved using a deterministic tie-breaker. The main question to be resolved is how to rank the items according to the samples from the posterior distribution P_w that we have derived with *MH* in order to present them to the user so as to receive new feedback. Moreover, recommender naturally faces the dilemma of recommending items that best match its current beliefs about the user, or items that could improve user satisfaction and help form more accurate beliefs. This corresponds to the typical exploration vs exploitation problem in learning environments [39]. Although several ranking methods exist, there is no universally accepted ranking semantics given the uncertainty in the utility function. In our work, we use a ranking method based on expectation. This is one of the most popular semantics, and has been used in several papers [6, 10] in the AI community.

3.4.1 Ranking based on Expectation (Exp) algorithm

The main question we must address is how to use the user samples that we have derived in order to rank the items and present them to the user. The MH algorithm returns us 10000 samples gathered from the weight distribution P_w. The more samples we use from the posterior, the better the suggestions are. On the other hand there is a trade-off here, because as we increase the number of the samples that we use in our ranking algorithm, its complexity also rises, and so does the system's response time. In the experiments we conducted, we observed that there is no need to use all the samples in the ranking algorithm. Therefore, we use only 50 samples which are randomly picked[8].

The expectation algorithm is defined as follows [14]. Given an item space I and probability distribution P_w over weight vectors w, find the set of top-n items I_n with respect to their expected utility value

[8] Those samples are picked randomly from the 10000 samples that the Markov Chain Monte Carlo algorithm returns us.

$$\forall\ i \in ST_I\ , \forall\ i' \in S_I \setminus ST_I\ , E_w(w\cdot i) \geq E_w(w\cdot i') \quad (15)$$

where *i* and *i'* is an item from the set of items S_I, ST_I is the set of top items, *w* is the weight vector and E_w is the expected utility value. As shown in Algorithm 4, first we construct a two dimensional array that we are going to populate with the utility of each item for every weight vector. The utility of an item is calculated by finding the sum of the item's features multiplied by the corresponding weights.

Let us suppose that our dataset consists of three items: *item1*, *item2*, *item3* and each of them has three features, as shown in Table 1 (a) below.

We provide an illustrative example:

items\features	f1	f2	f3
item1	0.3	0.7	0.5
item2	0.1	0.8	0.2
item3	0.7	0.2	0.5

(a)

weights\values	value 1	value 2	value 3
w1	-0.3	0.7	0.4
w2	0.5	0.8	-0.7

(b)

	Probability
w1	0.7
w2	0.3

(c)

utility	item1	item2	item3
w1	0.6	0.61	0.13
w2	0.36	0.55	0.16

(d)

	Utility (w1)	Utility (w2)	Expected Utility
item 1	0.6	0.36	0.528
item 2	0.61	0.55	0.592
item 3	0.13	0.16	0.139

(e)

***Table 1:** Exp example*

As shown in Table 1(b) we assume that we have two weight vectors according to which we want to sort our three items, and present to the user the item that is most appealing for her. In the Table 1(c) we have the probability for each of the two weight vectors. Subsequently we calculate the utility that an item has for each weight vector. For example, the utility of the item1 for the first weight vector is estimated as:

$$\text{f1} * \text{value1} + \text{f2} * \text{value2} + \text{f3} * \text{value3}$$
$$= 0.3 * (-0.3) + 0.7 * 0.7 + 0.5 * 0.4 = 0.6.$$

It is worth mentioning that because the items features are normalized and take values in the range of [0, 1] and the weight vector could take values in the range of [-1, 1], the range of values that the utility of an item with n features could take is in the range of [-n, n].

After filling the utility array, the expected utility value for each item can be calculated accordingly, using the probability of each weight vector. For Item1 in our example, the expected utility could be calculated as follows:

$$\text{utilityItem1}_{w1} * \text{probability}_{w1} + \text{utilityItem1}_{w2} * \text{probability}_{w2}$$
$$= 0.6 * 0.7 + 0.36 * 0.3 = 0.528.$$

Finally, we used the *Bubble Sort* algorithm (in Algorithm 5), in order to sort the dataset in ascending order according to the items expected utility, and present the top *n* ranked items to the user in order to receive feedback and repeat the whole process.

Algorithm 4: *Exp Ranking algorithm [14]*

```
Input:
        list weight_samples ~ each item of the list contains an array with the samples
                              values for each feature and a probability value.
        list dataset ~ each item contains an array with the normalized features values
                       of the item.

Procedure ExpAlgorithm( weight_samples,  dataset)
    double[][] utility_array = new double [weight_samples.size()][dataset.size()];
    initialize utility_array with zeros;

    for iteration i = 1, 2, …, weight_samples.size() do
        double[] sample = weight_samples.get(i).getSamples;
        for iteration j = 1, 2, …, dataset.size() do
            double[] features = dataset.get(i).getFeatures;
            utility_array[i][j] = sample[1]*features[1] + ...
                                  + sample[n]*features[n];
        end for
    end for

    double[] expected_utility =  new double [dataset.size()];
    initialize expected_utility with zeros;

    for iteration i = 1, 2, …, dataset.size() do
        for iteration j = 1, 2, …, weight_samples.size() do
            double sample_probability = weight_samples.get(j).getProbability;
            expected_utility[i] = expected_utility[i] + (utility_array[j][i] *
                                sample_probability) ;

        end for
    end for

    sort expected_utility;
    return the n items with highest expected_utility;
end procedure
```

Algorithm 5: Bubble Sort algorithm [40]

```
procedure bubbleSort( A : list of sortable items )
        int n = length(A)
        repeat
                boolean swapped = false

                 for iteration i = 1, 2, …, n-1 do
                        /* if this pair is out of order */
                        if A[i-1] > A[i] then
                                /* swap them and remember something changed */
                                swap( A[i-1], A[i] )
                                swapped = true
                        end if
                end for

        until not swapped
end procedure
```

3.4.2 Suggesting Items

A recommender makes suggestions to the user based on knowledge acquired through her user's previous interactions with the system. The main problem occurs when we have gathered little or no data on what the user prefers. In those situations, initial suggestions are not based on any previous knowledge of the user preferences, so could be almost "blind". Such a process would converge slowly, and especially the initial beliefs could be far from the user's true preferences. In the next section we propose a solution to address this problem.

Additionally, it is very important for our system to introduce an exploration component in the suggestions made. The user preferences in many cases are very complex and very difficult to map them all. We confront this challenge by presenting some random items combined with those that the exp algorithm returns. Those items serve the purpose of correcting the bias introduced from the initial distribution of P_w and combating mistakes and noise from user feedback. Moreover, the uncertainty inherit in the utility function aids exploration (in a true Bayesian manner).

In our experiments, we present to the user, in each interaction with our system, 5 items (hotels) that came from Exp Ranking algorithm described above, and also we introduce two random packages. After getting feedback from the user, we repeat the process of sampling through the posterior, using the new feedback, ranking the items according to the samples and present the top-n to the user.

There are two types of scenarios that our system could be used in. The most likely scenario is that our model to be used in conjunction with a "conventional" system that uses hard constraints defined by the user; or the user could even specify with her search input what she wants. In that case, our system could use those hard constraints and the input that the user gives, in order for our system to converge faster to the user's true preferences. On the other hand, when the system is used as standalone, the problem of learning the user's preferences is more pressing, because our system only relies on a very small number of suggestions that are made to the user in order to find the user preferences and thus to converge. We show in the next section that clustering can be helpful in tackling this problem.

3.5 Clustering

It is often the case in recommender systems that we have no knowledge of the preferences of a user, or that we have not gathered enough constraints in order to limit the uncertainty that the weight distribution P_w has. In such cases, recommenders can start proposing items only once they have been provided with hard constraints by the user, or once they have gathered enough information from user searches or choices. Another common approach is to randomly suggest items to the user in order to receive feedback and update the distribution. The last approach has the disadvantage that suggested items in the initial interactions with a user, might be far from her true preferences and this could result in producing suboptimal recommendations and therefore slow convergence of the weight distribution toward her true preferences.

In our work it was very important for our system to can be used both as a standalone, and also as a complementary system to a "conventional" one. Thus, we used clustering in the weight vectors we collect from our user's posteriors P_w. As we are going to show in the next chapter, we managed to almost reach minimum mean squared error that our system is able to achieve, even from the second interaction. Thus, instead of recommending random items in the initials interactions, we were able to use "clustered" weight vectors from users that have similar preferences with the current user. In our work, we used one of the most popular clustering algorithm, namely the k-means algorithm [11, 12], whose operation we describe below.

3.5.1 k-means

k-means is one of the simplest unsupervised learning algorithms that solve the well-known clustering problem. The problem is computationally difficult (NP-hard) [11]. However, there are efficient heuristic algorithms that are commonly employed and converge quickly to a local optimum. These are usually similar to the expectation-maximization algorithm for mixtures of Gaussian distributions via an iterative refinement approach employed by both algorithms. Additionally, they both use cluster centers to model the data. However, k-means clustering tends to find clusters of comparable spatial extent, while the expectation-maximization mechanism allows clusters to have different shapes [41].

The algorithm aims at minimizing an objective function known as squared error function given by:

$$J(v) = \sum_{i=1}^{c} \sum_{j=1}^{c_i} (\left\| x_i - v_j \right\|)^2 \quad (16)$$

where, $\left\| xi - vj \right\|$ is the Euclidean distance between x_i and $v_{j.}$,where v_j is a data point and x_i is a "centroid" (or, a cluster center); *'c_i'* is the number of data points in *i*th cluster; *'c'* is the number of clusters.

As shown in Algorithm 6 below, the key steps of the *k-means algorithm [42]* are the following:

1. Place k points into the space represented by the objects that are being clustered. These points represent initial group centroids.

2. Assign each object to the group that has the closest centroid.

3. When all objects have been assigned, recalculate the positions of the K centroids.

4. Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.

In more detail, *k-means* captures the insight that each point in a cluster should be near to the center of that cluster. First we choose *k*, the number of clusters we want to find in the data. In our case, we save after each user interaction the weight vectors samples derived from the posterior, and we use them as data in the k-means algorithm. Then, the centers of those k clusters, called centroids, are picked in some fashion (in our case, randomly). The algorithm then proceeds in two alternating steps: In the "reassign points" step, we assign every point in the data to the cluster whose centroid is nearest to it. In the "update centroids" step, we recalculate each centroid location as the mean (center) of all the points assigned to its cluster. We then iterate these steps until the centroids stop moving, or equivalently until the points stop switching clusters. Until k-means converges, after each “update centroids” step there will be data points belonging to a different cluster; these will be reassigned in the next “reassign points” step. When the algorithm converges, all the data points will actually be assigned to the corresponding centroid, so that further point reassignments and “update centroids” steps will have no effect.

It can be proved that the k-means eventually converges, as the sum of squared distances between each point and its centroid strictly decreases in both the “reassign points” and “update centroids” steps, and there are only finitely many cluster configurations. Despite some contrived examples in which k-means takes exponential time to converge, in practice it converges reasonably quickly. Since the “reassign points” and “update centroids” steps each take linear time, the practical run time of k-means is basically linear, and exactly so if you limit the number of iterations. Unfortunately, despite the fact that k-means is guaranteed to converge, it does not necessarily find the optimal configuration, i.e. the one corresponding to the global objective function minimum. The algorithm is also significantly sensitive to the initial selection of centroids. Running the algorithm multiple times reduces this effect.

Algorithm 6: *k-means algorithm [42]*

```
Input:
    $E = \{e_1, e_2, \ldots, e_n\}$ (set of entities to be clustered)
    $k$ (number of clusters)
    MaxIters (limit of interactions)
Output:
    $C = \{c_1, c_2, \ldots, c_k\}$ (set of cluster centroids)
    $L = \{l(e) \mid e = 1,2, \ldots, n\}$ (set of cluster labels of E)

foreach $c_i \in C$ do
    $c_i \leftarrow e_j \in E$ (e.g random selection)
end

foreach $e_i \in E$ do
    $l(e_i) \leftarrow argminDistance(e_i, c_j)\ j \in \{1 \ldots k\}$;
end

changed $\leftarrow$false;
iter $\leftarrow$0;

repeat
    foreach $c_i \in C$ do
```

```
                UpdateCluster($c_i$);
        end

        foreach $e_i \in E$ do
                minDist ←argminDistance($e_i$, $c_j$) $j \in \{1...k\}$;
                if minDist $\neq l(e_i)$ then
                        $l(e_i)$←minDist;
                        changed ←true;
                end
        end
        iter++;
until changed = true and iter ≤MaxIters;
```

3.5.2 Employing the Clusters

Every time we run MH for deriving the posterior samples for a user, we save those samples to our database. After having collected several samples of weight vectors of different users, we cluster them into k groups[9] using the k-mean algorithm. Such clusters serve two purposes.

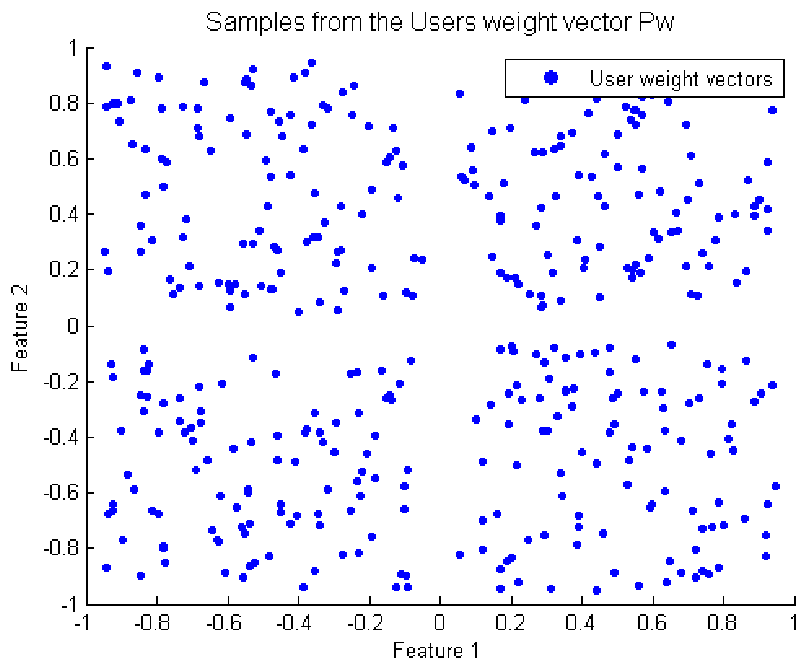
First, we use them to address the scenario of a new user entering the system. In that case all weight vectors are possible candidates, and we have the greatest uncertainty regarding user preferences. Thus, to limit this problem, we use the clusters centroids in the ranking algorithm, and suggest to the new user the most popular items from each group. In more detail, if our system suggests to a user n items in every interaction, we create n clusters that contain samples from the users' weight posterior distribution. Then the n centroids of those clusters are used in the ranking algorithm, to rank and present to new users the most highly ranked items based on the cluster centroids.

As shown in the experiments section, this process has a significant impact in reducing the initial uncertainty that we have for a new user. Figure 2 presents a simplified example, that shows how clustering helps in reducing the possible space of weight vector, assuming that we have n items in our dataset that have only two features (feature 1, feature 2) each. In Figure2 (a) depicts all the weight vectors that we have gathered from several users in our system. Let's say that we suggest 4 items per interaction in every user. Therefore, we construct 4 clusters, shown in Figure 2(b); Figure 2(c) also shows the centroid of each cluster.

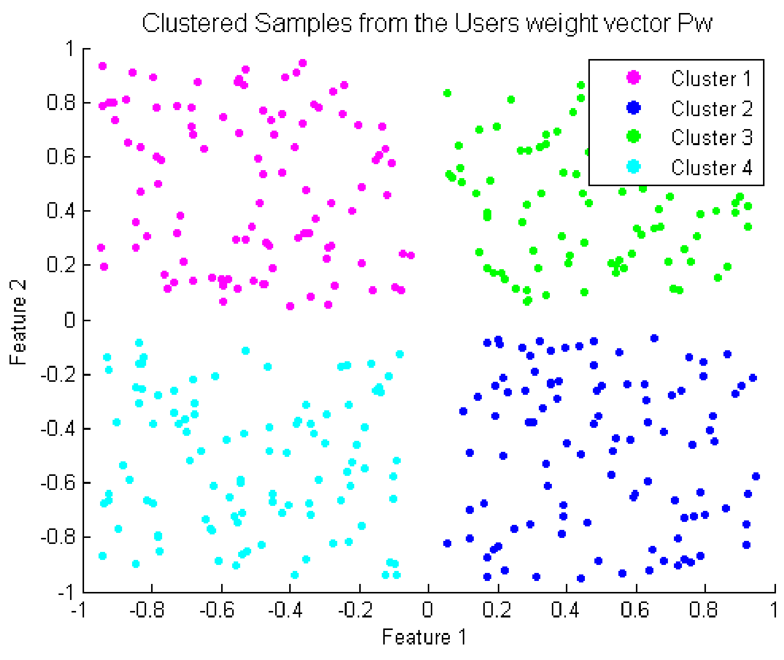
[9] We set it equal to the number of suggestion made to the user.

By suggesting items based on the cluster centroids, and receiving feedback from the user, we drastically decrease the uncertainty that we have about their preferences. (Recall that user feedback produces a set of pairwise preferences in the form of I1 > I2, where I1 and I2 are items). In other words, by using the cluster's centroids and leveraging the feedback received we "split" the weight vector space in k chunks.

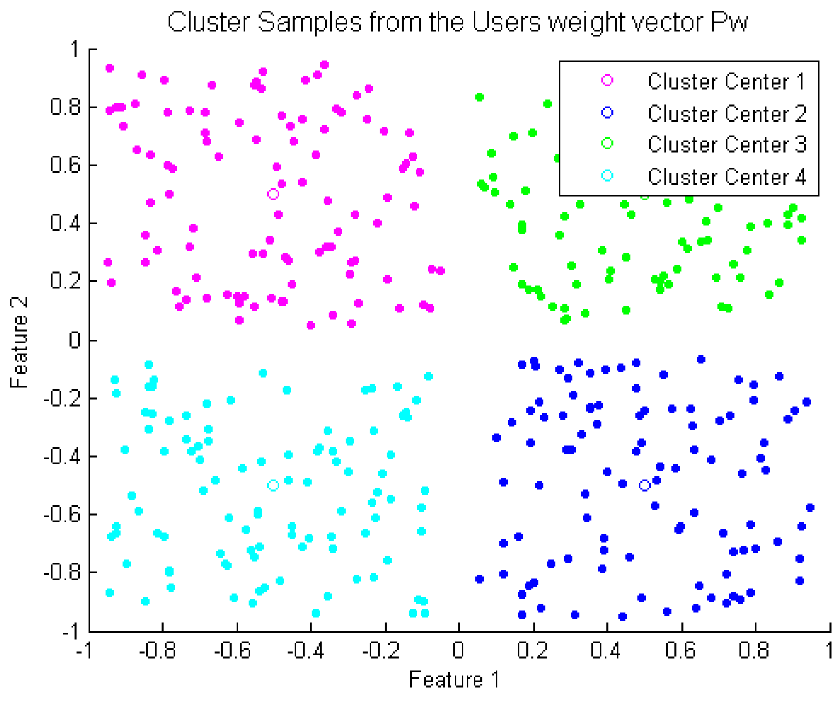
The second case we use the clusters is when we do not have gathered enough feedback from a user. This effect often occurs in the first three interactions, when we have gathered only a few constraints, and the user's posterior distribution still contains a lot of uncertainty. In cases like that, we use the clusters complementary to the samples from the algorithm in order to make the distribution converge faster. After deriving the weight vector samples from the user's posterior distribution, we find the cluster (constructed according to others' preferences, as explained above) that most of these weight samples belong in. Then we use (uniformly sampled) weight vectors from this cluster, in conjunction with the samples from the user's posterior, feed these in to the *Exp* algorithm and rank our items based on these samples. In this way, if the posterior has a lot of uncertainty we can reduce it by focusing on the most probable field in the space of possible weight vectors.



(a)



(b)



(c)

***Figure 2 :** Cluster example*

Chapter 4

System Evaluation

In this chapter, we study the performance of the proposed model. We run several sets of experiments and with different settings to validate our model's performance, with very encouraging results. For testing our system's performance, we built synthetic users using various distributions which we chose based on real data. Additionally, all the algorithms were implemented using Java8 (http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html) and for storing our data we used the PostgreSQL database (https://www.postgresql.org/). Also for dealing and managing with various distributions tasks, we used the Colt 1.2.0 library (https://dst.lbl.gov/ACSSoftware/colt/). Colt provides a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java. Also this library was developed in CERN. Finally for conducting the experiments and measuring the execution times, we used a desktop PC with the following specifications: CPU: i7 6700K 4.5Ghz, Ram: 16gb DDR4 3200Mhz, OS: Ubuntu 16.04.2 LTS.

The rest of this chapter is arranged as follows. In section 4.1 we discuss the dataset that we used. Subsequently, in section 4.2 we analyze how our simulated users were created. Then in section 4.3 we discuss the methodology used for conducting the experiments and the experiment setting in general. Finally, we present our various experimental settings, and evaluate our model's performance.

4.1 Dataset

In our work we used a real-world dataset that consists of 5000 hotels. The hotels are retrieved from TripAdvisor[10] in a *JSON*[11] format. In Figure 3 we present a sample of a hotel found in the *JSON* array of our dataset.

[10] https://www.tripadvisor.com
[11] https://www.w3schools.com/js/js_json_intro.asp

```
{
    "is_claimed": false,
    "distance": 1307.199253801,
    "mobile_url": "http:\/\/m.yelp.fr\/biz\/grand-h%C3%B4tel-des-gobelins-paris-2",
    "rating_img_url": "http:\/\/s3-
media1.ak.yelpcdn.com\/assets\/2\/www\/img\/5ef3eb3cb162\/ico\/stars\/v1\/stars_3_half.png",
    "review_count": 7,
    "name": "Grand H\u00f4tel des Gobelins",
    "snippet_image_url": "http:\/\/s3-media4.ak.yelpcdn.com\/photo\/NZ99qwK-
ZtCo4BniVtmF6Q\/ms.jpg",
    "rating": 3.5,
    "url": "http:\/\/www.yelp.fr\/biz\/grand-h%C3%B4tel-des-gobelins-paris-2",
    "location": {
      "city": "Paris",
      "display_address": [
        "57 boulevard St Marcel",
        "13\u00e8me",
        "75013 Paris",
        "France"
      ],
      "neighborhoods": [
        "13\u00e8me",
        "Port Royal\/Gobelins",
        "5\u00e8me"
      ],
      "postal_code": "75013",
      "country_code": "FR",
      "address": [
        "57 boulevard St Marcel"
      ],
      "state_code": "75"
    },
    "phone": "+33143317989",
    "snippet_text": "My fiance found this hotel through Hotels.com. It was nice, clean, and close to public
transportation (Metro and buses), restaurants, and a supermarket a...",
    "image_url": "http:\/\/s3-media3.ak.yelpcdn.com\/bphoto\/M7Ax-eKIaEb2ypCU__MEyw\/ms.jpg",
    "categories": [
      [
        "Hotels",
        "hotels"
      ]
    ],
    "display_phone": "+33 1 43 31 79 89",
    "price": "90",
    "currency": "euro",
    "rating_img_url_large": "http:\/\/s3-
media3.ak.yelpcdn.com\/assets\/2\/www\/img\/bd9b7a815d1b\/ico\/stars\/v1\/stars_large_3_half.png",
    "id": "grand-h\u00f4tel-des-gobelins-paris-2",
    "is_closed": false,
    "rating_img_url_small": "http:\/\/s3-
media1.ak.yelpcdn.com\/assets\/2\/www\/img\/2e909d5d3536\/ico\/stars\/v1\/stars_small_3_half.png"
}
```

Figure 3: *Hotel Sample from the JSON array*

We parsed the *JSON* array using Java, and then stored the hotels in our database. As we have mentioned in section 3.1.2, the item's (hotel's) features have to be normalized before they are ready for use. In order to compose a user's preferences, we focus on the following hotel features: the price/day for the hotel, the number of the hotel's "stars", the rating has got from past clients, the distance from the city center, and the amenities that it has (such as breakfast, pool, spa etc). The features we focused on, were chosen or were made to be quantifiable. For example, we could not use directly the actual hotel address as a metric, but we could retrieve from *Google Maps Distance Matrix API*[12] the distance in km from the city center that the hotel has. Thus, we were able to learn if the user prefers hotels near the center or in the countryside. Additionally we clarify that we did not have the intention to make recommendations to the user about the destination. During our experiments, we assume that the user has chosen her destination, and we try to learn his preferences in order to make personalized recommendations, so that she should not need to search manually through the thousands of hotels (5000 in our case) that a place may have. All the hotels from our dataset were located in Paris. Thus we assume that all of our users wanted to go to Paris and we try to make the best suggestion to them based on our current knowledge about their preferences.

4.2 Building the Synthetic Users

In this section we are going to describe how the synthetic users were created. We generated 100 simulated users that was used in order to evaluate our system's performance. Their preferences values were sampled from distributions. It was important that the simulated users represented as best as possible the preferences of the real users, in order to have a realistic assessment of the model performance. Thus as we are going to explain in more detail later in this section what distributions we chose, and why we chose them. Also by creating the user preferences by sampling distributions, we introduce an uncertainty necessary to account even for users with unconventional preferences.

[12] https://developers.google.com/maps/documentation/distance-matrix/start

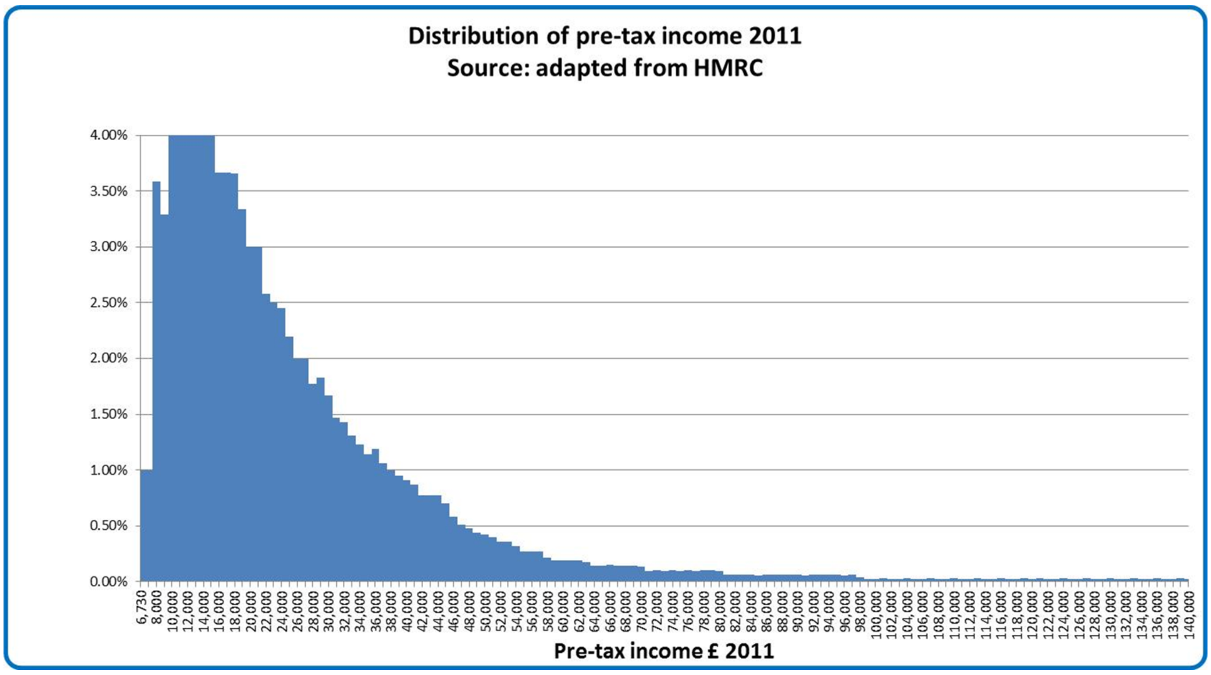


Figure 4: *2011 UK income distribution*

The preferences of each user consist of five values. The price that she is willing to pay for a hotel (per night), the number of the “stars” which he expects the hotel to have based on the price he pays, the previous guests ratings, the proximity to the city center, and finally the number of amenities she wants a hotel to have. After sampling the distributions we normalize those values. Thus all preferences values lie in the [0, 1] interval. In order to determine the price per day that a user is willing to pay for a hotel, we were inspired by the income distribution of the UK citizens for the year 2011 shown in Figure 4. Figure 5 depicts the distribution that we actually used and as it is clear, matches to the actual income distribution of Figure 4. Thus, for extracting the price preference, we used a Burr type XII distribution (a = 25007, c = 2.0, k = 2.0). The Burr distribution[13] can fit a wide range of empirical data. Different values of its parameters cover a broad set of skewness and kurtosis. Hence, it is used in various fields such as finance, hydrology, and reliability to model a variety of data types. Examples of data modeled by the Burr distribution are income, crop prices, insurance risk, travel time, flood levels, and failure data.

[13] https://en.wikipedia.org/wiki/Burr_distribution

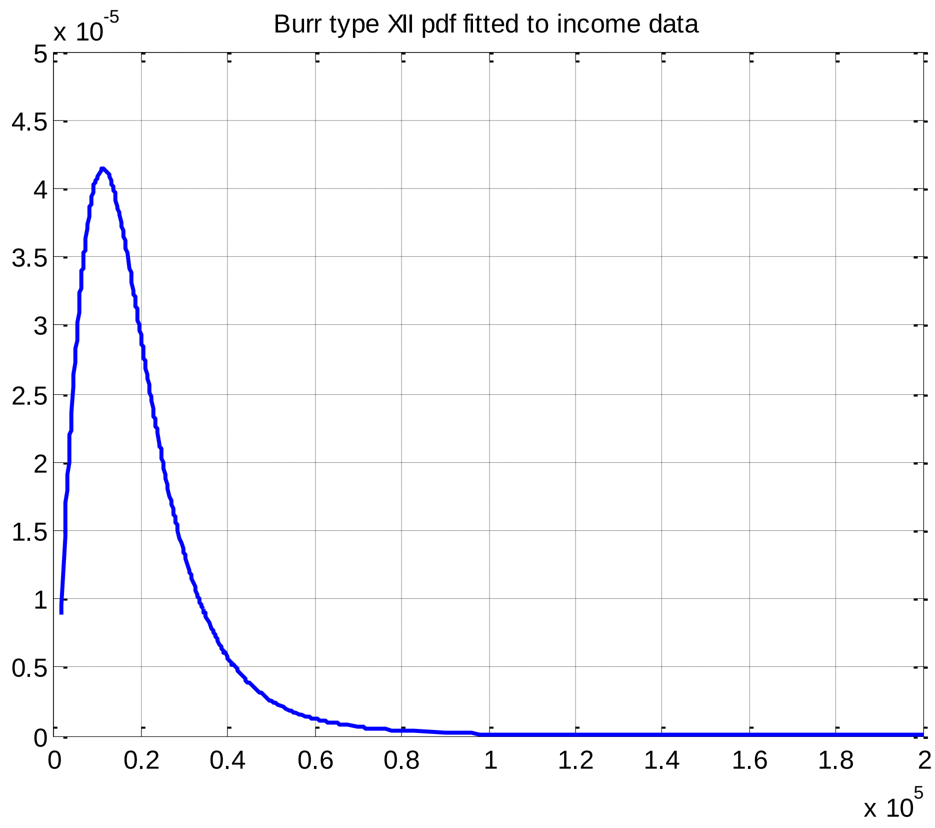


Figure 5: *Burr distribution that simulates the 2011 UK income distribution*

Afterwards, depending on the price sampled, we chose one of the three beta distributions shown in Figure 6, to derive the value for the preference for the number of the hotel stars. More specifically, we assume that if the "can pay" price is high, then it is expected that a hotel with a high number of stars (4-5) is preferred, so we sample a Beta distribution with the following parameters a = 8, b = 2 [14]. Respectively, we use a Beta distribution with parameters a = 2, b = 8 [15] in the case that the price value is low. Otherwise, we take sample from a Beta distribution with a = 8, b = 8[16]. Here, we choose a Beta distribution, as we want the number of stars to have mean values between zero and one. Additionally, such distributions are more appropriate for modelling uncertainties in the real world, as they can concentrate probability in a desired range [43, 44]. Of course, despite these facts, other distributions can also be used.

[14] Shown in Figure 6 with the red line.
[15] Shown in Figure 6 with the green dotted line.
[16] Shown in Figure 6 with the blue dashed line.

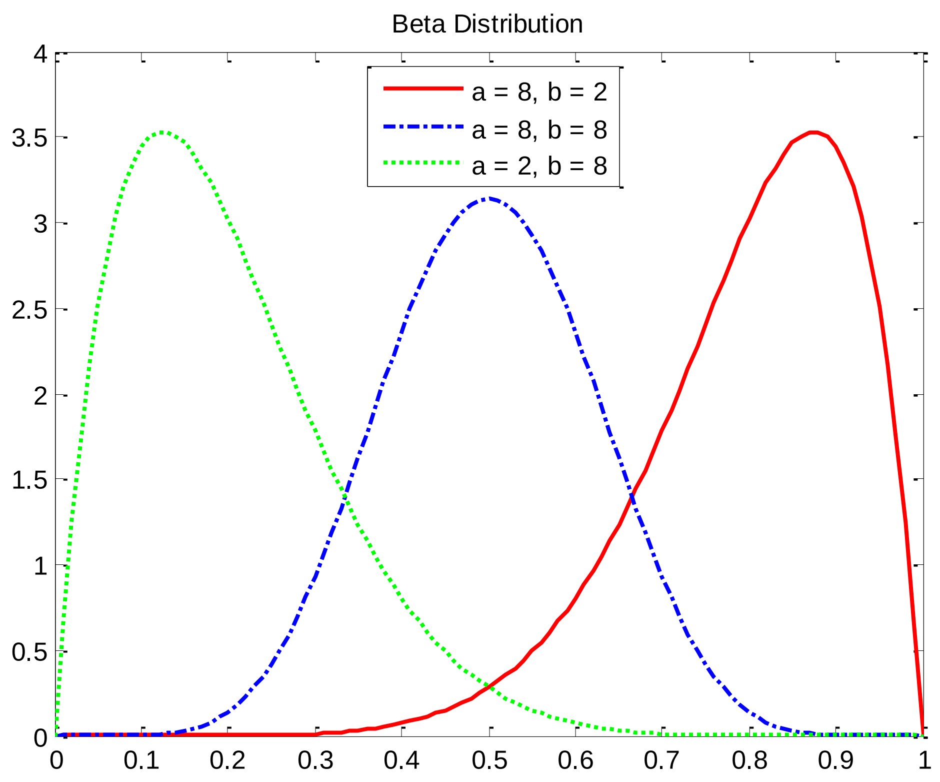


Figure 6: *Beta distribution.*

Subsequently we choose to sample a Beta distribution with parameters a = 1, b = 3, shown in Figure 7, in order to derive the proximity to the city center preference. Using this distribution we want to simulate the tendency, that the greatest percentage of people prefer a hotel near to the city center than in the countryside. A sampled value closer to zero corresponds to a preference for a hotel close to the center, while value closer to 1 means the opposite.

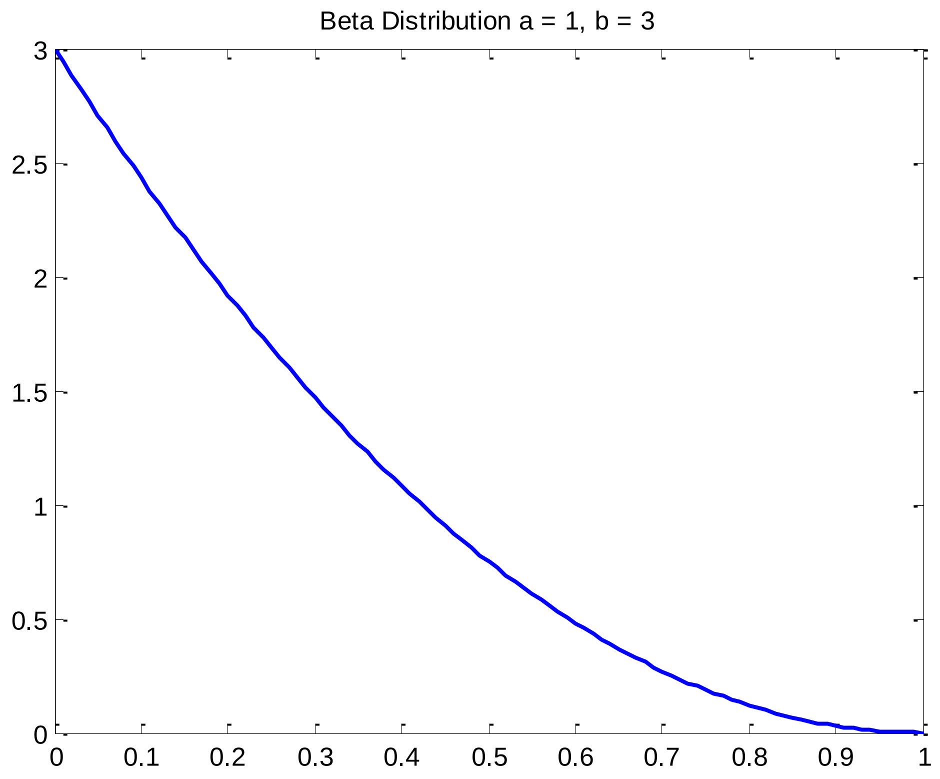


Figure 7: *Beta distribution a=1, b=3*

In the next step in order to specify the hotel "ratings by previous guests" value a user requires, we used a Beta distribution with parameters a = 6 and b = 2, shown in Figure 8. We choose this distribution in order to simulate the tendency that most users prefer hotels with high ratings. Finally the last feature that characterizes the preferences of a user was the required hotel amenities. The amenities include things like whether the hotel has pool, restaurant, spa, activities, etc. In order to derive the amenities preference, we used a uniform distribution in the range of [0, 1], where a higher number meant that a hotel with more amenities is preferred, and a value closer to zero implies it is not important for the user that the hotel has many amenities. We clarify that we made the assumption that all amenities have the same and equal impact to a user.

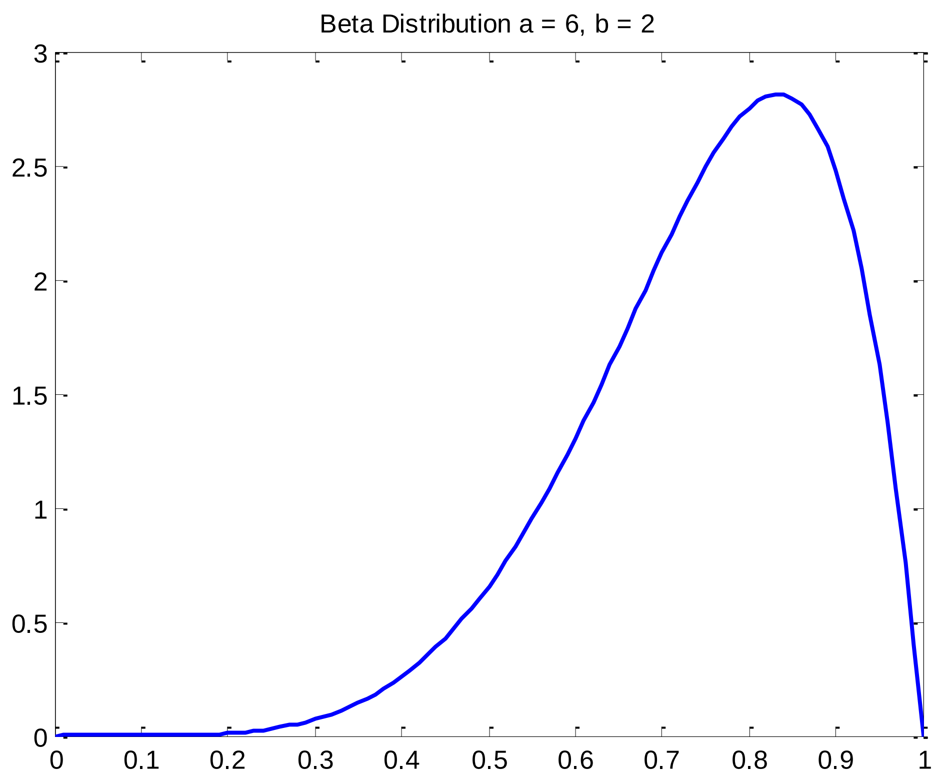


Figure 8: *Beta Distribution a=6, b=2*

4.3 Experiments and Results

For our experiments we used 100 simulated users, each of which had 20 interactions with our system. Employing synthetic users enables one to work with exact and quantifiable preferences, a challenging and error prone task where real users are involved. This help us measure the efficiency and effectiveness of our model.

In every interaction, we save to our database the new feedback that we derive, and we use the feedback implied by her choice to condition the prior distribution, obtaining samples from the posterior distribution using MH to rank our hotels, and thus to make new suggestions. In each interaction we propose to the user seven hotels, and she chooses one of them. Five of those are the top ranked hotels according to our current beliefs regarding user preferences, and the other two are picked completely randomly from our dataset. The choice of the simulated user was based on the Euclidean distance:

$$d(h, u) = \sqrt{(h_{F1} - u_{F1})^2 + (h_{F2} - u_{F2})^2 + \cdots + (h_{Fn} - u_{Fn})^2}$$

where h_{F1}, h_{F2}, … , h_{Fn} the values of the corresponding features of the hotel, and u_{F1}, u_{F2}, …, u_{Fn} are the actual user preferences. Thus the user will always choose the hotel with the minimum Euclidean distance from her preferences.

$$User_{\,choice} = min\ (d(h_1, u), d(h_2, u), \ldots, d(h_m, u)),$$

where m is the number of suggestions made to the user[17].

Additionally in order to have a clear picture of our model's performance, for each simulated user we found and stored to our database the top 200 hotels[18] based on the Euclidean distance between the actual user preferences and the hotels' features. Therefore, at the end of each experiment we were able to compare our model's suggestions with the hotels which she would prefer if she could search all of the hotels in our database (especially for a big dataset like ours, an impracticable and time inefficient task).

In our experiments, we calculate the (average) mean square error (MSE) for each user interaction, and thus we were able to observe the accuracy of the recommendation made as the interactions increases. The measure of mean squared error requires a target of prediction or estimation, along with a predictor or estimator which is said to be the function of the given data. MSE is defined as the average of squares of the "errors".

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (Y'_i - Y_i)^2 ,$$

where Y' is a vector of n predictions, and Y is the vector of observed values corresponding to the inputs to the function which generated the predictions. The n in our case, is equal to 20 and corresponds to the 20 user interactions.

In order to calculate the MSE [45], we used as the prediction values the real values that compose the preferences of the simulated users, and as estimated value we used the hotel that the user chose in the corresponding interaction. In more detail, for computing the "prediction" value, we sum the true preferences values of a user as shown below.

[17] In our case the suggestions in each interaction were seven.

[18] As we mentioned earlier, the total number of the hotels in our dataset is 5000, so the top 200 correspond only to the 4% of the whole dataset a small fraction of the whole dataset.

$$Y' = \sum_{i=1}^{n} u_{Fi} \ ,$$

where u_{F1}, u_{F2}, …, u_{Fn} are the actual user preferences over hotel features, and n is the number of preferences (five in our case). Additionally for computing the observed values, we sum the selected hotel feature values.

$$Y = \sum_{i=1}^{n} h_{F1} \ ,$$

where h_{F1}, h_{F2}, … , h_{Fn} the values of the corresponding features of the selected hotel, and n is the number of the hotel features. As we see in Figure 9, the MSE drops drastically in the first interaction and after the fifth interaction we observe that it starts to stabilize in a low value. Although we have not made use of the clusters yet, to reduce the uncertainty in the initials interactions, we observe from the result that the MSE is quite low even from these interactions.

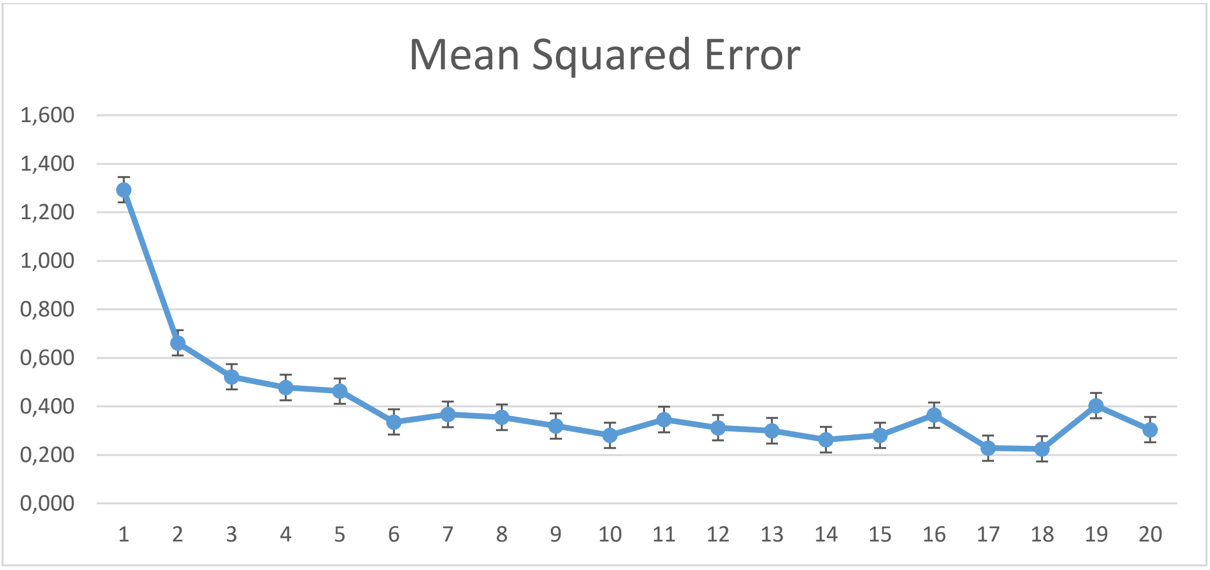


Figure 9: *Mean Square Error without using the Clusters*

The use of Mean Squared Error, calculated according to the user's true value function and her "ideal" (but "virtual") choices, could give us an insight of the overall method performance, we came up with a metric that is even more intuitive and indicative of the method's strength. Thus, for each simulated user we found and saved the "top 200" ranked hotels for the 5000 hotels of our dataset, based on the minimum Euclidean distance from the user’s true preferences. Then, we were able to compare the suggestions that our model made with the top 200 ranked hotels for each user.

As shown in Figure 10 and Figure 11 after only five interactions, our system was able to suggest hotels with almost 40% of them being among the best 200 hotels of each user. As mentioned before, we have stored the top 200 hotels for each user, and after the user interactions we can calculate how many of the recommendation made where inside in the user’s “top 200”. Then in Figure 10 we depict the average percentage of the user recommendations that belonged in the “top 200”. The “top 200” constitutes only 4% of our dataset of 5000 hotels. Similarly, we observe that about 20% belongs in the “top 50” (1% of our dataset), and about 9% belongs in the “top 10” (0.2% of our dataset). Furthermore, we can verify the results of Figure 9 by comparing it with the Figure 11 that shows the big difference in the models preference prediction, that we observe between the first five and the last five interactions (out of the twenty interactions per user made in total).

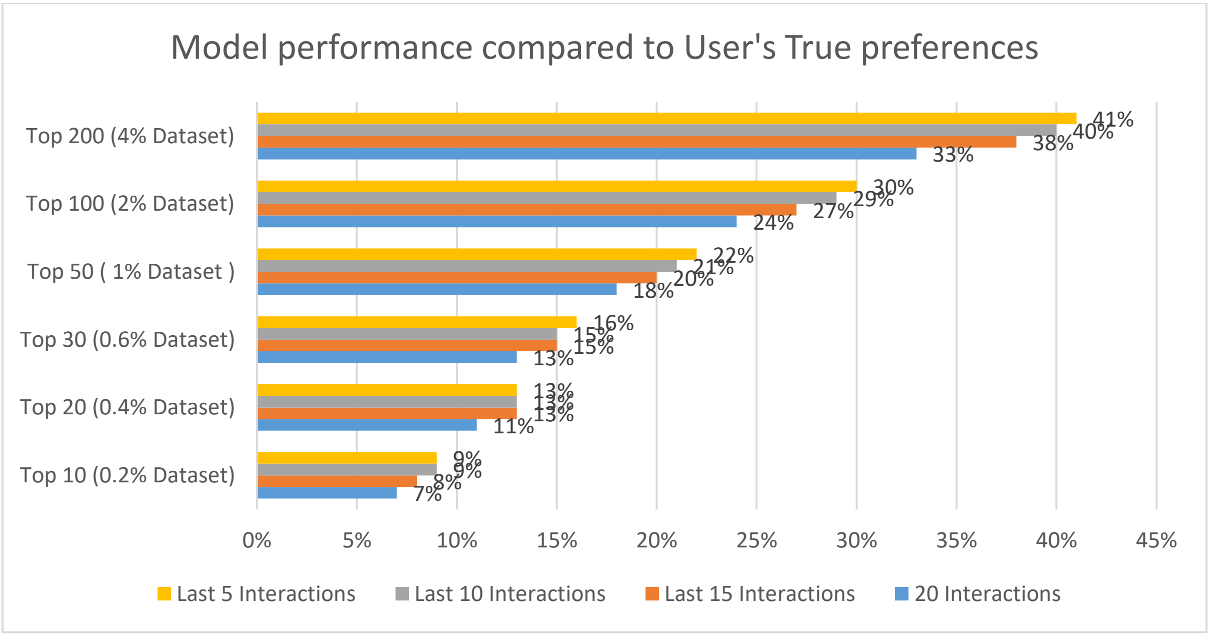


Figure 10: *Model performance compared to User's True preferences without the clusters*

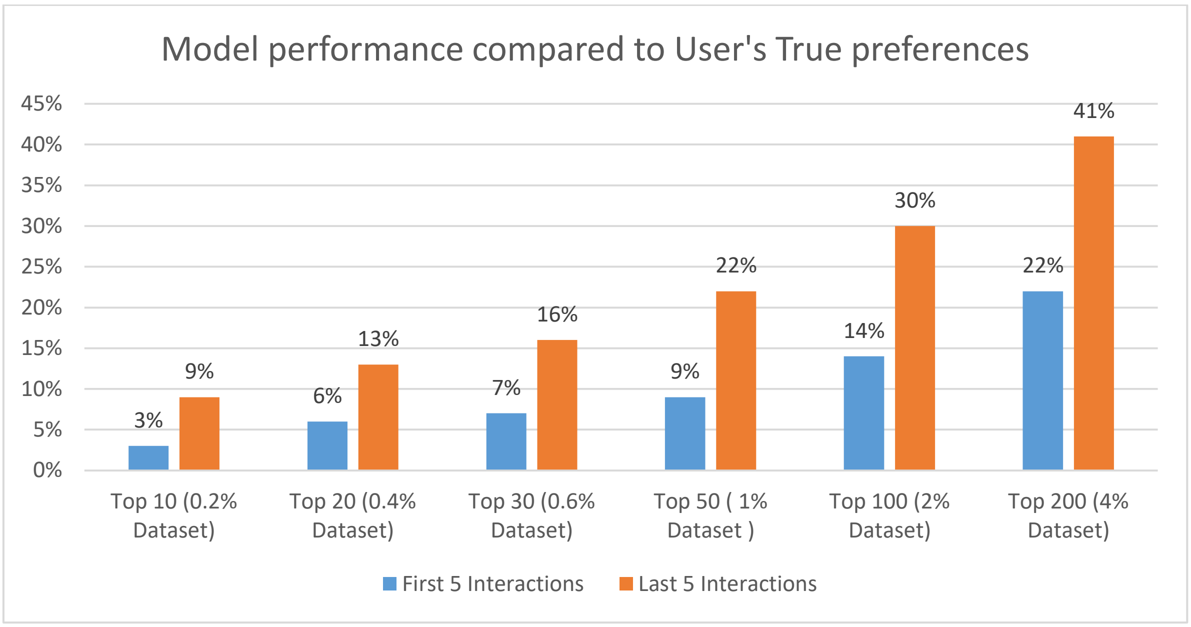


Figure 11: *Model performance between the first five and last five interactions (without using the clusters)*

4.3.1 Using different number of samples in the ranking algorithm

Figure 12 below shows the effect we observe on the MSE when we used a different number of weight vectors. We notice that if we use only one weight vector in order to rank our items, then we might draw a sample that has a low probability compared to the others. We observed that the more samples we use from the user's weight distribution posterior, the better the suggestion are, and this translates to a lower MSE. On the other hand, there is a trade-off here, because as shown in Table 2 below, the increase of the number of the samples used in the Exp algorithm to rank the hotels, increases also execution time. As shown in Figure 12, using less than 50 samples, the MSE increased and with more than 50 samples we get marginal reductions, so in all our experiment we used 50 samples in the ranking algorithm, as a good number of samples that balances the performance with fast system response time. This is essential for a real time recommender system. As shown in Table 2, by using 50 weight vector samples we manage to keep the average execution time under 2 seconds per interaction, which we believe is an acceptable waiting time for a real time RS.

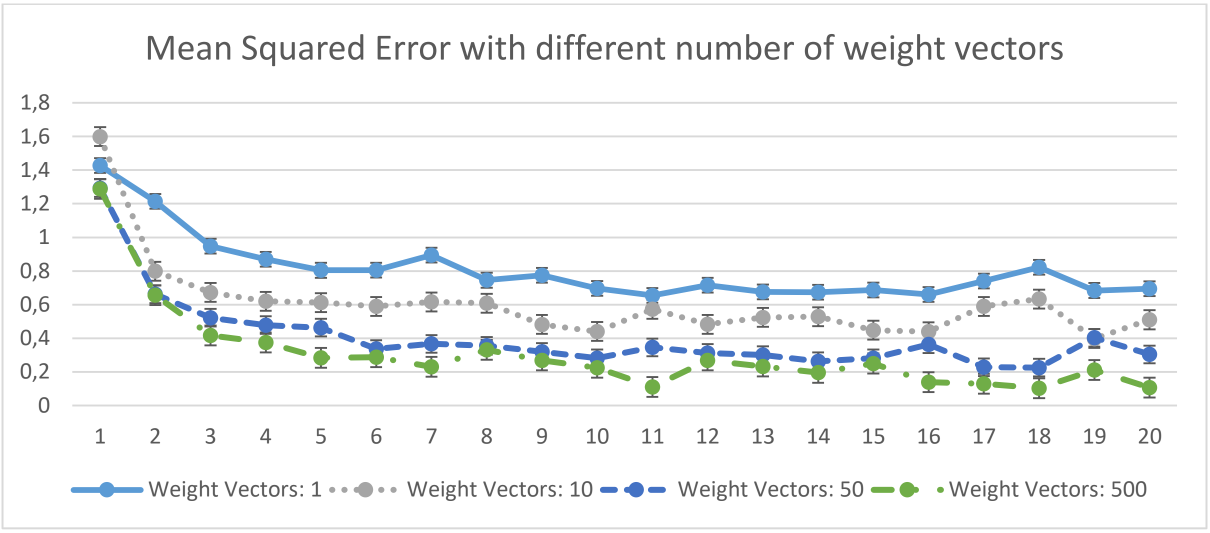


Figure 12: *Mean Squared Error with different number of weight vectors*

Numb. of weight vectors used	Execution Time (ms)
1	1620
10	1778
50	1975
500	2447

Table 2: *Average execution time (in ms) per interaction for using different number of weight vectors*

4.3.2 Using the clusters

In this section we compare the results and the performance that we observed when using clustering to reduce the initial uncertainty that we have for the user. As we see in Figure 13, we observe a faster reduction of the mean squared error when the clusters were used. However, in the long run we observe that we get about the same MSE. Subsequently in Figures 14 and 15 we evaluate the method's "quality", by observing how many of the suggestions made to the user belongs to his top rated 200 hotels. In Figure 15 we want to highlight the difference in performance that we see when we use the clusters, especially in the initial interactions that we have the greatest uncertainty regarding the user.

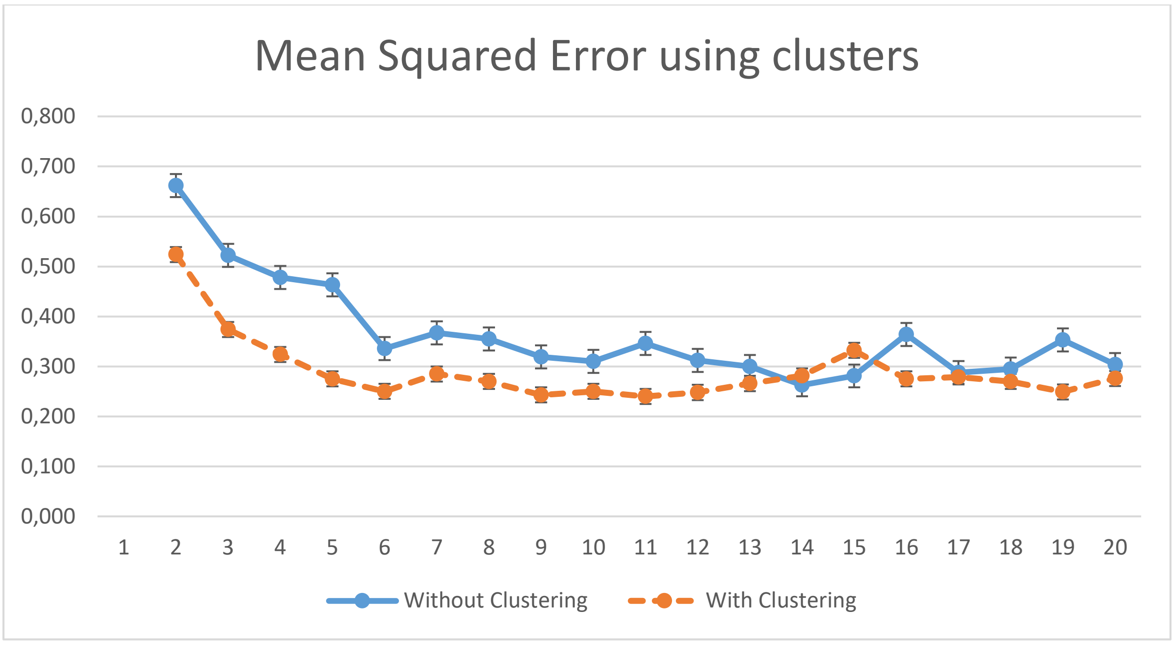


Figure 13: *Mean Squared Error using the clusters*

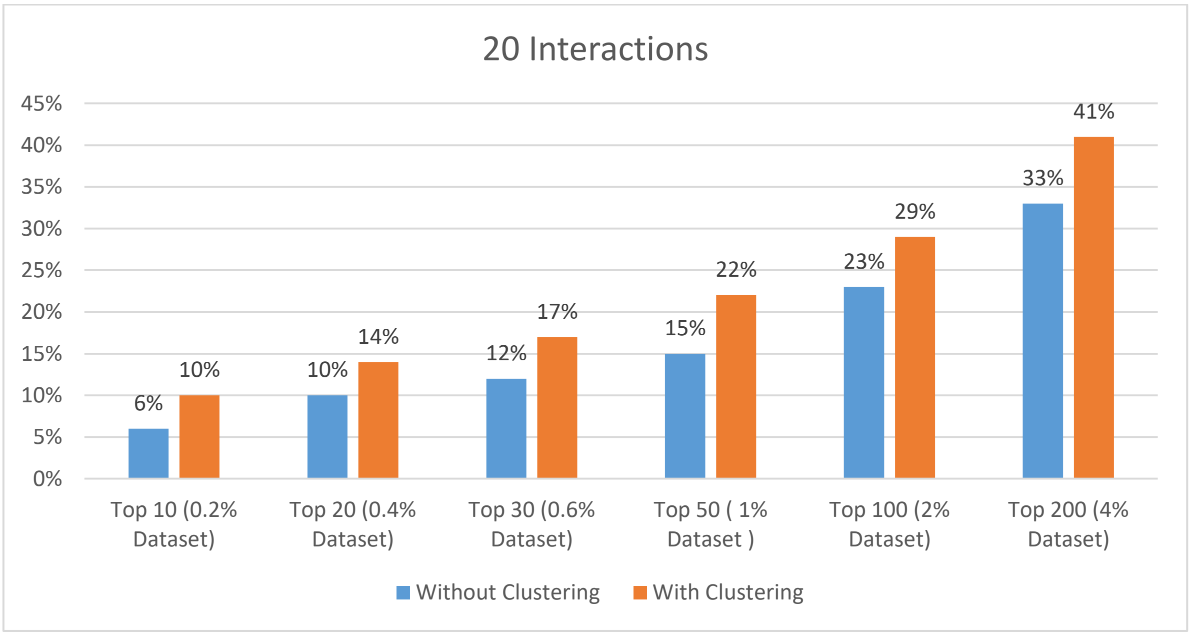


Figure 14: *Comparison in the overall system performance with and without clustering.*

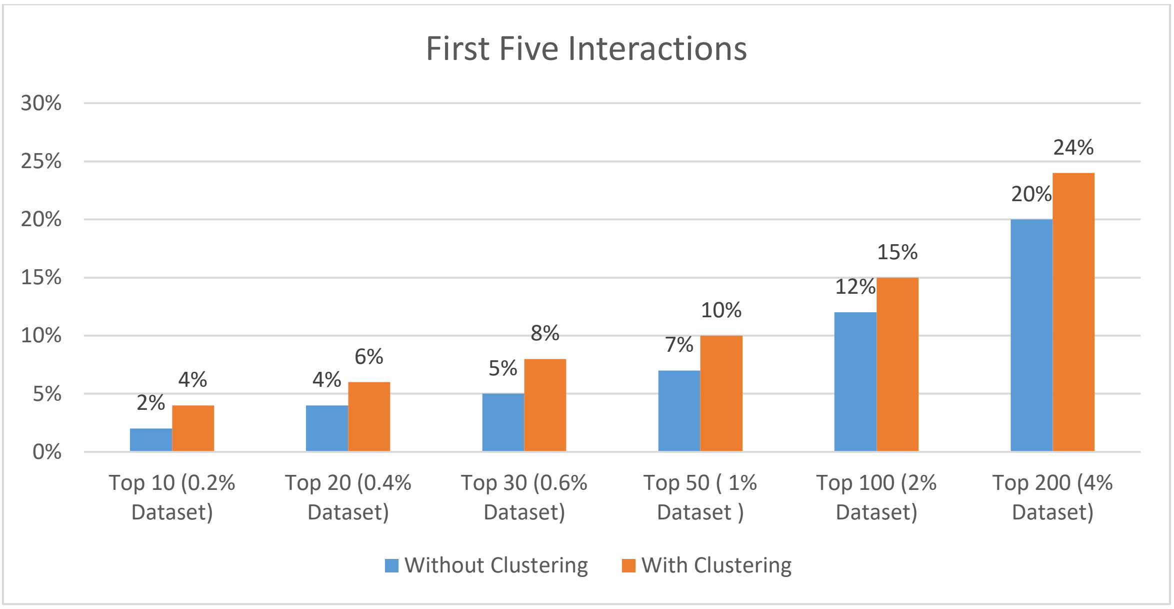


Figure 15: *Comparison in the overall system's performance between the first five interactions with and without clustering.*

When we compare the results in Figure 16 with those of Figure 10, we observe a significant increase of the percentages of "successful" recommendations. In more detail, for each user we found how many of the recommendations made belonged to his "top 200", and in Figure 10 and Figure 16 we depict the average percentage between the 100 users in total. As we can observe in Figure 10, after conducting 20 interactions per user, 33% of the suggestions belonged to the "top 200". However, when we used clustering, the affirmation percentage increased to 41% Additionally, we observed that after the first 10 user interactions (so this corresponds to the remaining 10 interactions, "last ten interaction" in Figure 16), an average of 48% of the total suggestions belong to the users' "top 200"; while an average of 30% of the total recommendations belongs to the user's "top 100" (as shown in Figure 16). Also, we can notice that among the 20 interactions, over 10% of the suggestions made, are located in the top 10 list of the user's most preferred hotels in the whole dataset.

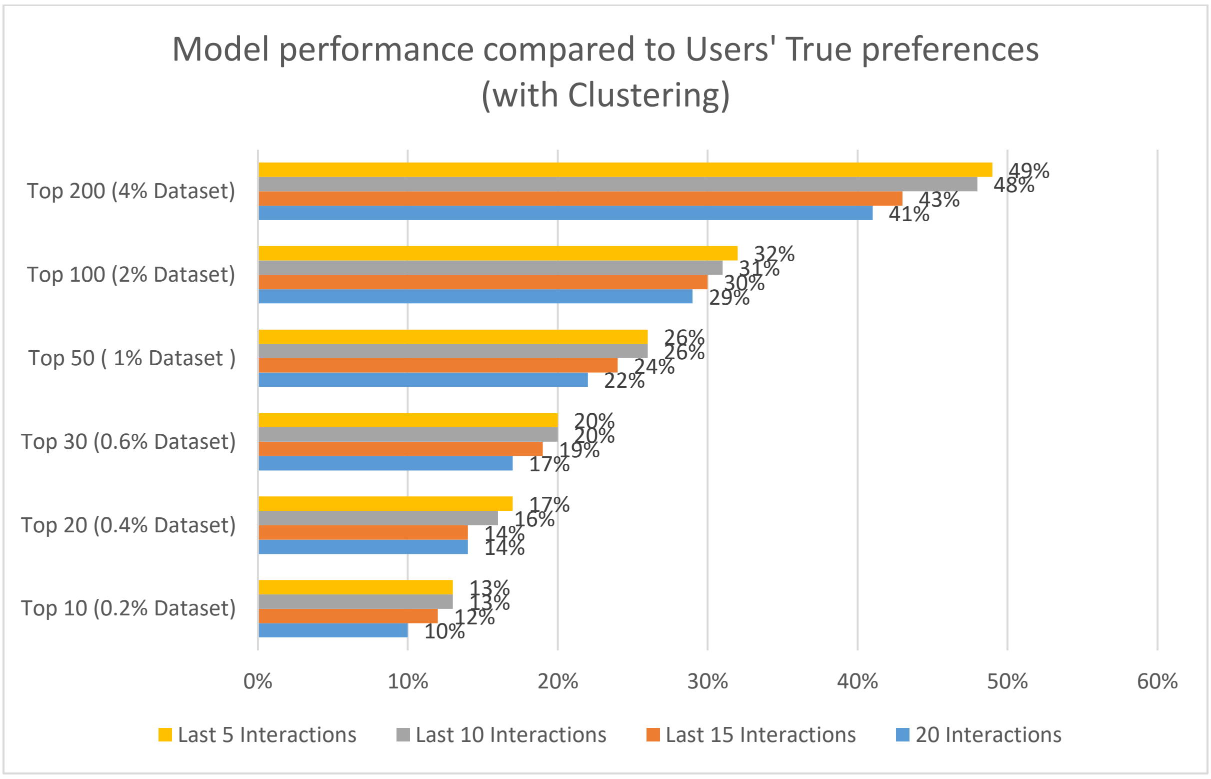


Figure 16: *Model performance compared to Users' True preferences using the clusters*

4.3.3 Used as a supplementary system

Finally, in this section we test the system's performance in a more realistic setting. In most websites such as Amazon[19], a recommendation system is being used in conjunction with a "conventional" system. When we refer to a "conventional" system we mean a system that uses hard constraints or the search input in order to retrieve hotels, PC parts, books, movies and in general items that meet the criteria defined by the user. In a setting like that, our recommendation system could benefit and use that hard constraints or the clicks that a user does and thus converge faster. In order to simulate a case like that, we randomly pick one of the top 200 hotels that each user has, and inject it as a suggestion to the first interaction that he had with our system. This case simulates the situation that the user searches manually in a site and finds what he wants through hard constrains or the search input. As shown in the Figure 17 above, our system was able to suggest hotels with low MSE, after only two interactions, and thus very close to her actual preferences.

[19] https://www.amazon.com/

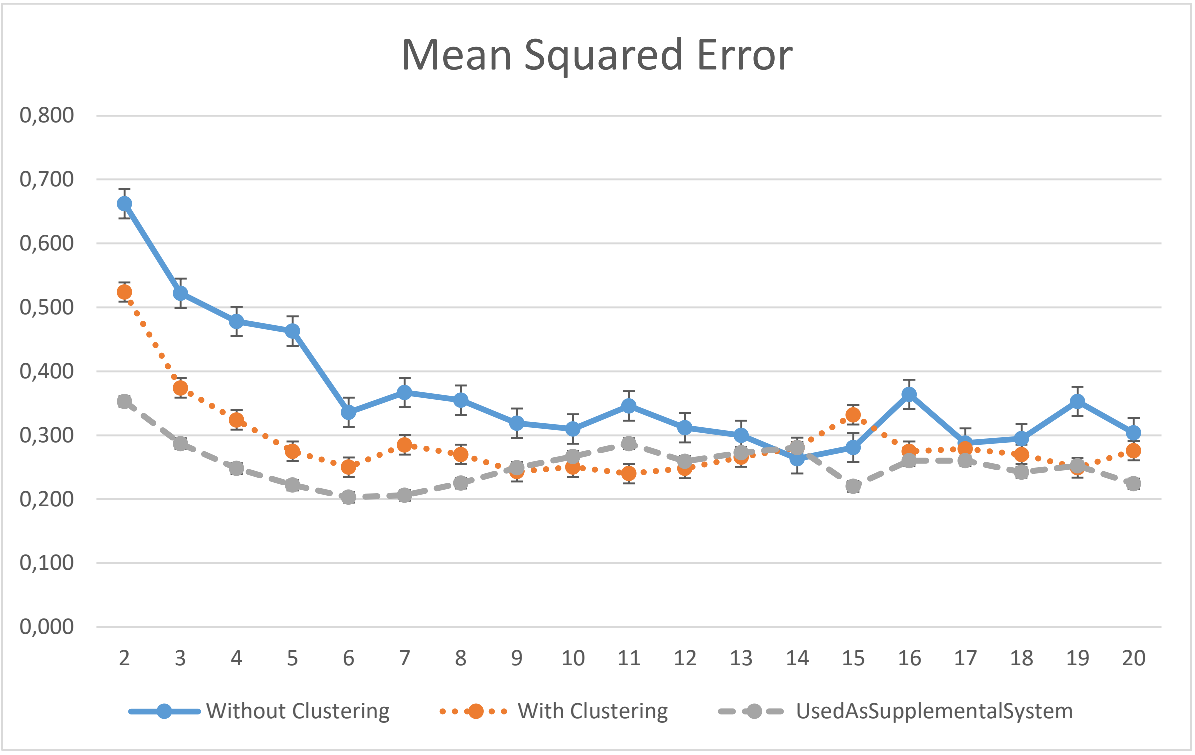


Figure 17: *Mean Squared Error comparison in a case that the recommendation system assumed to be used in conjunction with a conventional system.*

Chapter 5

Conclusion and Future work

In this work, we presented a lightweight recommendation system which uses a Bayesian preference elicitation framework, and applied it in the online hotel booking recommendations domain. Our approach make personalized recommendations, is quite simple and fast, but still generic and effective. It was very important for us to make a recommendation system that is not intrusive or asks the user to rate items. Thus we manage to build a passive observer of the user’s actions that makes suggestions based only on the selections (clicked items) that the user made in our system.

The core of the proposed system is a linear utility function which captures a user's personalized preferences over items. Unlike works which assume that this utility function is given, we learn weight parameters of this function through implicit user feedback in the form of clicks on suggested items. We then use these constraints to transform a given prior distribution over utilities into a more informative posterior. Due to the fact that updating the prior distribution in a multidimensional space every time that a user provides feedback is a very demanding and time consuming task, we used a sampling based method using an *MCMC* algorithm which treats user feedback as constraints, and avoids calculating posterior distributions of the weight parameters explicitly. Subsequently, we can use those samples from the weight vector posterior in order to rank our items according to them, and present the top-n ranked items to the user in order to receive new feedback and update the system's knowledge of the corresponding utility function.

As we observed in our experiments, our system stabilizes in a low MSE value after just a few interactions. In order to reach this low MSE value even earlier, we make use clustering which is an optional component of our system. More specifically, we used the weight vectors derived from past users interactions, and cluster them in k groups. Thus, we were able to use them to drastically reduce the initial uncertainty that we had for a new user. Moreover we compared the suggestions made to a user with the list of his “top 200” items. As we observe in Chapter 4, typically over 40% of the suggestions made were located in the “top 200” list of each user (a list that corresponds to the 4% of the dataset).

There are many interesting possible extensions to this work. One interesting extension of our model would be to provide package recommendations. We could use an aggregation profile like the one used in [14]. More specifically, aggregations can be used over feature values of items in a package. E.g., the sum of the costs of items defines the overall cost of a package, while the average of the ratings of items indicates the overall quality of a package. Package recommendations could be useful in many domains. For example, instead of recommending hotels, our system could recommend complete holiday packages including the airplane tickets, the hotel, the place and activities that can be done by the user based on his preferences.

Finally, it is very common that a user is affected, by unexpected factors or unquantifiable features such as the presentation of the item or the available pictures. For example, in a booking recommendation scenario, like ours, the user may not choose based solely on features, on the contrary she may influenced by the beautiful photos that a hotel may have. There exist numerous works in the marketing and advertising domain that describe how the presentation of an item could strongly influence the choices of a user. Additionally, a user may accidentally click on a suggested item, or may change her mind after clicking. Thus, in order to take into account cases like these we could incorporate a noise model in our system that could simulate unexpected and unquantifiable factors which affect the user's choice in order to construct a more realistic model of the user preferences.

Bibliography

[1] Lops, P., M. Gemmis, and G. Semeraro (2011). *Content-based recommender systems: State of the art and trends*. In F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor (Eds.), Recommender Systems Handbook, pp. 73-105. Springer US.

[2] Ricci, F., Rokach, L., Shapira, B., Kantor, P.B *Recommender Systems Handbook*, 2015 - Springer

[3] Bollacker, K.D., Giles, C.L.: CiteSeer: *An Autonomous Web Agent for Automatic Retrieval and Identification of Interesting Publications*. In: K. Sycara, M. Wooldridge (eds.) Proceedings of the Second International Conference on Autonomous Agents, pp. 116–123. ACM Press (1998)

[4] Thomas Whalen and Geoffrey Churchill, *Decisions under Uncertainty*, Robinson College of Business Georgia State University Atlanta, Georgia 30303-3083 USA

[5] Deepak Ramachandran, Eyal Amir, *Bayesian Inverse Reinforcement Learning*, In Learning (J Shawe-Taylor, R S Zemel, P Bartlett, F C N Pereira, K Q Weinberger, eds.), Morgan Kaufmann Publishers Inc., volume 51, 2007.

[6] C. Boutilier. *A pomdp formulation of preference elicitation problems*. In Association for the Advancement of Artificial Intelligence (AAAI), pages 239–246, 2002.

[7] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[8] Applegate, D. & Kannan, R. (1991). *Sampling and integration of near log-concave functions*. STOC '91 Proceedings of the twenty-third annual ACM symposium on Theory of computing. Pages 156-163.

[9] Urszula Chajewska, Daphne Koller, Dirk Ormoneit, *Learning an Agent's Utility Function by Observing Behavior*. International Conference on Machine Learning (ICML), 2001 - ai.stanford.edu.

[10] U. Chajewska, D. Koller, and R. Parr. *Making rational decisions using adaptive utility elicitation*. In Association for the Advancement of Artificial Intelligence (AAAI), pages 363–369, 2000.

[11] MacQueen, J. B. (1967). *Some Methods for classification and Analysis of Multivariate Observations*. Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability.

[12] Lloyd, S. P.. *Least square quantization in PCM*. Bell Telephone Laboratories Paper 1957.

[13] V.Klee and G.J. Minty. *How good is the simplex algorithm*, Inequalities III (Academic Press, New York, New York, 1972).

[14] Min Xie, Laks V.S. Lakshmanan, Peter T. Wood, *Generating Top-k Packages via Preference Elicitation*. Proceedings of the VLDB Endowment Volume 7 Issue 14, October 2014 Pages 1941-1952

[15] Melville, P., R. J. Mooney, and R. Nagarajan (2002). *Content-boosted collaborative filtering for improved recommendations*. In Proceedings of the 18th AAAI Conference, Menlo Park, CA, USA, pp. 187-192.

[16] Debnath, S., N. Ganguly, and P. Mitra (2008). *Feature weighting in content based recommendation system using social network analysis*. In Proceedings of the 17th International Conference on World Wide Web (WWW 08, New York), NY, USA, pp. 1041-1042. ACM.

[17] McNee S.M., Lam, S.K., Konstan, J., Riedl, J., *Interfaces for eliciting new user preferences in recommender systems*. Proceedings of User Modeling Conference, Springer, 2003, 178–187.

[18] R. Bellman. *A Markovian decision process*. In Proc. of Journal of Mathematics and Mechanics, 1957

[19] Andrew Y. Ng, Stuart J. Russell, *Algorithms for Inverse Reinforcement Learning*, ICML '00 Proceedings of the Seventeenth International Conference on Machine Learning, Pages 663-670

[20] J.L. Herlocker, J.A. Konstan, L.G. Terveen, J.T. Riedl, *Evaluating collaborative filtering recommender systems*, ACM Trans Inform Syst, 22 (1) (2004)

[21] Ekstrand, M. D., J. T. Riedl, and J. A. Konstan (2011, February). *Collaborative Filtering recommender systems*. Found. Trends Hum.-Comput. Interact. 4 (2), 81-173.

[22] Zhou, Y., D. Wilkinson, R. Schreiber, and R. Pan (2008). *Large-scale parallel collaborative filtering for the netflix prize*. In Proc. 4th Int. Conf. on Algor. Aspects in Information and Management, LNCS 5034, pp. 337-348. Springer.

[23] Yehuda Koren, *Matrix Factorization Techniques for Recommender Systems*, Published by the IEEE Computer Society, IEEE 0018-9162/09, pp. 42- 49, ©IEEE, August 2009

[24] Peter Brusilovsky (2007). *The Adaptive Web*. p. 325. ISBN 978-3-540-72078-2.

[25] Pazzani, M. and D. Billsus (2007). *Content-based recommendation systems*. In P. Brusilovsky, A. Kobsa, and W. Nejdl (Eds.), The Adaptive Web, Volume 4321 of Lecture Notes in Computer Science, pp. 325-341. Springer Berlin Heidelberg.

[26] Shearin, S. and H. Lieberman (2001). *Intelligent profiling by example*. In Proceedings of the 6th International Conference on Intelligent User Interfaces, IUI '01, New York, NY, USA, pp. 145-151. ACM.

[27] Basu, C., H. Hirsh, and W. Cohen (1998). *Recommendation as classification: Using social and content-based information in recommendation*. In In Proceedings of the Fifteenth National Conference on Artificial Intelligence, pp. 714-720. AAAI Press.

[28] Konstantinos Babas, Georgios Chalkiadakis, and Evangelos Tripolitakis: *You Are What You Consume: A Bayesian Method For Personalized Recommendations*. In Proc. of the 7th ACM Conference on Recommender Systems (ACM RecSys 2013), Hong Kong, China, October 2013.

[29] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Decisions with Preferences and Value Tradeoffs*. Cambridge University Press, 2003.

[30] George B. Dantzig, Alex Orden, Philip Wolfe. *The generalized simplex method for minimizing a linear form under linear inequality restraints.* Pacific Journal of Mathematics, 1955 - msp.org

[31] Jorge Nocedal Stephen J. Wright, *Numerical Optimization*. Springer Science, 1999.

[32] G McLachlan, T Krishnan, *The EM algorithm and extensions,* Wiley, 1997

[33] A. P. Dempster, N. M. Laird and D. B. Rubin, *Maximum likelihood from incomplete data via the EM algorithm*, Wiley for the Royal Statistical Society. Series B (Methodological) Vol. 39, No. 1 (1977), pp. 1-38.

[34] Hastings, W.K. (1970). "Monte Carlo Sampling Methods Using Markov Chains and their Applications. Biometrika, 1970 - Biometrika Trust.

[35] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. *Equation of State Calculations by Fast Computing Machines*. The Journal of Chemical Physics 21, 1087 (1953).

[36] W. R. Gilks and P. Wild, *Adaptive Rejection Sampling for Gibbs Sampling*, Journal of the Royal Statistical Society. Series C (Applied Statistics) Vol. 41, No. 2 (1992), pp. 337-348

[37] Ilker Yildirim, Bayesian Inference: *Metropolis-Hastings Sampling*, Department of Brain and Cognitive Sciences University of Rochester Rochester, NY 14627 2012

[38] Siddhartha Chib, Edward Greenberg, *Understanding the Metropolis-Hastings algorithm*, Pages 327-335 | Received 01 Apr 1994, Published online: 27 Feb 2012

[39] Russell, S. and P. Norvig (2009). *Artificial Intelligence: A Modern Approach* (3rd ed.). Upper Saddle River, NJ, USA: Prentice Hall Press.

[40] Wang Min, *Analysis on Bubble Sort Algorithm Optimization*, Information Technology and Applications (IFITA), 2010 International Forum.

[41] Ashok Kumar Panda, Satchidananda Dehuri, Isha Padhy, *Index Page Synthesis Using Genetic Algorithm*, Proceedings of the International Conference on Information Systems Design and Intelligent Applications 2012 (INDIA 2012) held in Visakhapatnam, India, January 2012 pp 489-496

[42] K. A. Abdul Nazeer, M. P. Sebastian, *Improving the Accuracy and Efficiency of the k-means Clustering Algorithm*, Proceedings of the World Congress on Engineering 2009 Vol I WCE 2009, July 1 - 3, 2009, London, U.K.

[43] Kota, Ramachandra, Chalkiadakis, Georgios, Robu, Valentin, Rogers, Alex and Jennings, Nicholas R. (2012) *Cooperatives for demand side management*, At the Seventh Conference on Prestigious Applications of Intelligent Systems (PAIS @ ECAI), France. 29 - 30 Aug 2012.

[44] C Akasiadis, G Chalkiadakis, *Cooperative electricity consumption shifting*, Sustainable Energy, Grids and Networks, 2017

[45] David M. Allen, *Mean Square Error of Prediction as a Criterion for Selecting Variables*, Pages 469-475 | Published online: 09 Apr 2012