MASTER'S THESIS | LUND UNIVERSITY 2015

Recommender System Validation Platform

Johan Ullén

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-25

Recommender System Validation Platform

(Comparing recommender models in test- and live-environments)

Johan Ullén

d04ju@student.lth.se
johanu@expertmaker.com

LUNDS UNIVERSITET
Lunds Tekniska Högskola

June 22, 2015

Master's thesis work carried out at Expertmaker AB.

Supervisor:
Jacek Malec, Jacek.Malec@cs.lth.se
Anders Berkeman, anders.berkeman@expertmaker.com
Examiner:

Jonas Skeppsted, Jonas.Skeppstedt@cs.lth.se

Abstract

With most applications where recommender systems are used, it is important that they produce a better result than a system with no recommender, or one with a previous recommender. Deploying an untested system, even to a smaller user sample can be very costly if the system produces negative results. It is often in a developer's interest to create several candidate systems. They need some way of comparing recommender systems before selecting one or a few to launch. While the methods of testing have been explored, and their statistical soundness motivated, in other work [1, 4, 7, 17], it is not obvious how to do it in practice. This report describes the implementation of a modular and configurable framework, and analyses this framework with two different cases. The experimentation shows the power of how such a framework can be utilized to reduce overhead work when approaching evaluation of a new recommender system.

Keywords: recommender system, generic framework, validation, evaluation

Acknowledgements

When I began this project in February 2014, I had no idea where it would take me. The problem formulation was intentionally vague to allow a flexible course to be set by myself. Both the data domain, the retailer problem that was at first provided, and the domain of recommender systems were new, and unknown, to me. I have learned through this year, and this project, what my strengths and interests, as well as my weaknesses, are.

The greatest obstacle for me in this project has been the vertical depth we had to go into to analyse the data, and motivate the results. I have never found it easy to approach an obscure, and unknown domain, and I too easily fall back on areas more comfortable to me. It is because of that most of the time spent with the project was spent on designing, and implementing, the framework we used. The idea was that the more modular we could build it, the easier it would be to modify it to suit our needs. I have, from working on this project, realised that my strength is in the design, and implementation, of structure. Creating a sturdy base in which I can fit the pieces I do not completely grasp, lets me stand more firmly when my balance is off.

I would like to thank examiner Jonas Skeppsted for examination and support, supervisor Jacek Malec for his help and feedback, Expertmaker and Lars Hård for the opportunity to do this project, Anders Berkeman, Johan Wensäter, and everyone else at Expertmaker for their technical support and guidance. I would also like to thank my two opponents, Christer Persson and Anders Rehn, for their insightful, and useful comments on the report. For proofreading of, and feedback on, the report I thank Ariel Lai, Mats Walhberg, Alexander Rappe, Henrik Backlund, Niklas Carnerup, Vilhelm Ullén, Zhang Xiao Ni, Markus Hegnelius, Sabina Ngo and Paul Stilley. Finally, special thanks to Hampus Sahlin for the cooperation between our two theses.

Contents

1 Introduction **9**
1.1 Domain . . . 9
1.1.1 Recommender Systems . . . 10
1.1.2 Evaluation of Recommender Systems . . . 10
1.2 Problem . . . 10
1.2.1 Why is it interesting? . . . 11
1.3 Work Distribution . . . 11

2 Background **13**
2.1 Recommender systems . . . 13
2.1.1 Basic Techniques . . . 13
2.1.2 Complications with Recommender System . . . 14
2.2 Evaluating recommender systems . . . 15
2.2.1 Experimental Settings . . . 15
2.2.2 Dimensions . . . 16
2.2.3 Comparison . . . 25
2.2.4 Complications with Evaluation . . . 25

3 Approach **27**
3.1 Philosophy . . . 27
3.1.1 Modularity . . . 27
3.1.2 Average . . . 28
3.2 Churn . . . 29
3.2.1 Data . . . 29
3.2.2 Recommender Purpose . . . 30
3.3 The Retailer . . . 30
3.3.1 Data . . . 30
3.3.2 Recommender Purpose . . . 31

4 Implementation **33**
4.1 Development 33
4.2 Framework 34
4.2.1 Recommender System Evaluation 34
4.2.2 Sequence 35
4.2.3 Modularity 38
4.2.4 Configuration 39
4.2.5 Utilities 39
4.3 Churn 39
4.3.1 Data cleaning 39
4.3.2 Sampling 40
4.3.3 Simulation 41
4.3.4 Evaluation 41
4.4 The Retailer 41
4.4.1 Data cleaning 41
4.4.2 Sampling 41
4.4.3 Simulation 42
4.4.4 Evaluation 43

5 Results **45**
5.1 Churn 45
5.1.1 Single Evaluation 45
5.1.2 Multiple Permutations Evaluation 48
5.2 The Retailer 50

6 Discussion **53**
6.1 Results 53
6.2 Evaluation 53
6.3 Framework 53
6.4 Future 54

7 Conclusion **55**

Bibliography **57**

Appendix A Churn **63**

Appendix B Configuration File **67**

Preface

Since the early to mid 1990s when the first collaborative filtering techniques were first suggested [8, 18] evaluation of recommender systems have been discussed. While there is a large amount of information to be found on such evaluation [4, 7, 8, 17, 18, 25, 27], they mostly cover an academic example where there is a distinct separation between train- and test- data. A real world case is rarely as simple as that, as there are either dependencies between the train- and test- data or tendencies . This report focuses on exploring a general methodology to evaluate both the simple case, where data is easily separated, and more complex cases.

This project was conducted at Expertmaker [5], a local software developer with focus on business optimization powered by artificial intelligence. Expertmaker offers a rare opportunity to work hands on with a large amount of data, a chance to work with experienced people with vast knowledge of recommender systems, and the machine learning domain. Expertmaker also provided a unique opportunity to work on hardware that can handle the large amount of data provided. It is interesting to note that, although the data provided may be considered large, it is but a small fraction of the data handled by the company servers on a daily basis.

Two cases have been studied for this purpose: a mobile operator churning example (the churn case), explored in sections 3.2, 4.3 and 5.1, and a retailer loyalty example (the retailer case), explored in section 3.3, 4.4 and 5.2. The churn case, an example often used for exploring prediction problems, is a mobile phone operator who wishes to know what customers are most likely to cancel their plan. Expertmaker has been building a recommender system for a large supermarket chain in the United States. The purpose of the project is to recommend discount coupons to customers, to increase loyalty.

This report covers a project of how to build a generic evaluation system for recommender systems. In chapter 1 the reader will be introduced to recommender systems, their evaluation, and the problem at hand. A thorough exploration of recommender systems and their evaluation, as well as a brief summary of other literature on related topics, can be found in chapter 2. Chapter 3 explains the theoretical approach the author has taken to generalizing

evaluation of recommender systems. This chapter also introduces two cases: the churn case, and the retailer case. The implementation of a generic framework (the framework) as well as the specific implementations for the churn, and the retailer cases, can be found in chapter 4. Results from the execution of the implementation of the examples are displayed, and explained in chapter 5. Chapter 6 gives the author's point of view of the results and discusses further needs of implementation, and research, on the topic. Finally, the author's conclusion of this project can be found in chapter 7.

Chapter 1
Introduction

In today's fast moving global society the staggering amount of information, and products, to weed through has become almost insurmountable for consumers. A more and more commonly occurring way to cope with this is the use of recommender systems. Recommender systems seek to predict the preference users would give items, and by doing so make it possible to sort them.

This chapter introduces the domain of recommender systems and their evaluation. In section 1.1 the domain of recommender systems, and their evaluation is introduced. Section 1.2 introduces the reader to the problem with generalization of recommender systems. Finally section 1.3 describes how the workload was split between the author of this project and Hampus Sahlin, whose yet unfinished project shared parts of building and utilizing the framework.

1.1 Domain

Recommender systems belong to the domain of machine learning. By utilizing historical knowledge of users, a recommender system can try to predict users' preferences in the future. For an online movie streaming service, such as Netflix [13], a recommender system can use customers ratings of movies they have seen, to predict how well they will like other movies. A retailer supermarket customer can receive discount coupons of items they are likely to purchase, based on what they usually purchase. For the end consumer this would be beneficial if they can easier find cheaper products, or when they are introduced to items, products, or movies, they didn't know existed. For the service provider it is beneficial as good recommendations can build loyalty from the customer. On the other hand, the recommendations could be bad, e.g. a customer could have been recommended a product they either knew they didn't want, or one they were disappointed with after purchase or consumption. This could essentially be worse than no recommendation at all.

1.1.1 Recommender Systems

Recommender systems typically produce a sorted list from some allocation of items. Each item is given a weight based on features that can be extracted from the items or the users. *Collaborative filtering*, and *content-based filtering* are two common ways of extracting features for an item. *Collaborative filtering* builds a model from users' historical data. Users may be recommended items they commonly use, for example a user who often buys vegetables may be given a discount coupon for vegetables, or a user on Amazon [2] will, when buying an item there, be given a list of recommendations of what other users also bought, when they bought that item. *Content-based filtering* utilize discrete characteristics of the items, when browsing an article on The New Yorker [21] a user will get links to related articles.

1.1.2 Evaluation of Recommender Systems

Evaluating recommender systems is usually done by accuracy metrics. Three different dimensions of accuracy: *prediction*, *classification* and *rank*, are most commonly used [4, 7, 17]. *Prediction accuracy* is a metric of how far off a system is in its ratings, e.g. in a scoring system such as the Swedish movie rating and recommendation website Filmtipset [6]. A system tries to predict what rating a user would give a movie. Once the user has given a movie a rating the system compares the actual rating with the predicted rating and a score is given.

Classification is the attempt to correctly label items and users. On Netflix [13] a recommender system will try to predict what movies a customer want to see. On Amazon [2] a recommender systems will try to predict what products a customer want to buy. These items are identified either as a customer want them, or as they don't want them. The *classification accuracy* is the comparison between the predicted class and the actual class. In a binary system, such as the ones described above, these are either *true positive*, *false positive*, *false negative*, or *true negative*. This becomes more complicated with multiple class labels.

Ranking accuracy is the evaluation of the order of the recommendations. The sooner an item is recommended correctly the higher score it will provide to the evaluation. This is based on the analysis of the recommendation as a whole rather than on item properties, like the *prediction* and *classification accuracies*.

1.2 Problem

Recommender system evaluation is not a new research topic. The first articles, according to Adomavicius and Tuzhilin [1], about *collaborative filtering*, and their evaluation, appeared the mid 1990s [8, 18]. Since then, and in particular in the 2000s, a vast amount of research reports, and books, have been written on the subject [1, 4, 7, 17, 22, 25, 27]. While these reports, and books, cover a lot of different aspects, both basic and advanced, there is little concrete information on how to evaluate a recommender system in practice.

1.2.1 Why is it interesting?

To systematically evaluate recommender systems it is necessary to know whether the systematic approach provides the same result as manual configuration. The setting up of such a framework has in itself been the most interesting, while generating results has become secondary to that.

While off-line evaluation has been thoroughly covered in the past [4, 7, 20], a generic and modular way of implementation, with user models and simulations, seem somewhat lacking. The modularity of a framework that allows it to evaluate any kind of recommender problem, with only minor additional implementation, could be a great asset to anyone working with recommender systems.

1.3 Work Distribution

Although this thesis is a solo project, the parallel work with Hampus Sahlin lead to a few joint efforts. The design, and implementation, was done in unison, and has some specifics implemented to handle the scenarios Sahlin's project needs. While these specifics will not be covered in this report, it is worth mentioning they exist, and provide an additional frame of modularity. The gist of the specifics are automatic rerunning of the framework with multiple configurations, using results from previous runs in the initialization of models, and returning not only the recommendations of a model, but also the weights, and other data used, to generate the results.

Design

In the early parts of the projects Sahlin and the author worked mostly separately with the same data, to get a basic understanding of its domain, and how to create recommendations. While some of the recommender models, for the retailer case, designed at this time survived until the latest iterations of the framework ,they are not included in the results. Based on the very similar work done manually it was decided on framework design, on which both could work in an easier way. The design of the framework was discussed, and planned, over a week, and the result was the base of what would become the framework presented in 4.2. This design was later remodelled as needed, but the general structure is still the same when this report is written.

Implementation

The basic structure of the framework's implementation was divided evenly between Sahlin and the author, this includes the sequence: data import, data sampling, and execution. The further development, most of the different modules, and naive models, were implemented by the author. The evaluation was done completely by the author.

Initially the framework was not particularly modular, only allowing different recommender models on the same data, and with one evaluation technique, and very slow besides that. To speed up the execution saving, and loading, partial results to disk, was

implemented. When more models with different data requirements were designed, a way to modularly build suitable data structures was implemented. A method to iterate over multiple configurations, to allow multiple executions, was also implemented.

Many additional propositions were suggested over time, e.g. parallel processing. But due to lack of time, and actual need, they were all scrapped.

Report

This report is in its entirety written by the author. In the early stages Sahlin and the author started writing a few paragraphs together, but as time went on, and Sahlin had other duties, those parts have been either rewritten, or scrapped. Not because of any fault on the writing, but due to the increased understanding, and needs, of the report.

Sahlin's Project

The parallel project is to design a general way of boosting a recommender model by combining results of other, more direct, models. By relying on the evaluation done in this project, and the weights used for sorting with different models, a better than, or equal, model can be generated. The link between this project and Sahlin's was the evaluation, and the framework it was set in. By first executing several naive models, and evaluating them, the framework allowed new models to execute those same models with new data, and let a booster model take the previous results, weigh them, and combine the results of the new execution. Given the evaluation this booster model generally perform better than its parts.

Chapter 2
Background

This chapter works as introduction of previous work related to the approach taken in this project. In section 2.1 common techniques for building recommender models are introduced. In section 2.2 different settings, and dimensions, commonly used when evaluating recommender systems are explored. Introducing the reader to the basics of recommender system evaluation, and some of the different settings possible, should help the reader follow the approach to the problem dealt with in this project.

2.1 Recommender systems

Multiple different techniques have been explored for building recommender systems [4, 7, 10, 11, 17]. Two of these, *collaborative-filtering*, and *content-filtering*, are discussed in section 2.1.1, while section 2.1.2 covers some known complications with recommender systems.

2.1.1 Basic Techniques

The two most common ways for building recommender systems are *collaborative-filtering* and *content-filtering* [4, 7, 10, 11, 17]. Other recommender systems described in [17] are *neighbourhood-based*, *constraint-based*, and *context-aware* systems. Combinations of systems, called *combinatorial filters*, can also be used. The latter four will not be described in this report, but it is good to be aware of their existence.

Collaborative-Filtering

Collaborative-filters are described by Koren, and Bell [10] as producing recommendations of items based on patterns of ratings, or usage wihtout need for exogenous information about either items or users. They makes recommendation based on ratings that users have assigned to items. Two different approaches exist: *user-based*, and *item-based* [4]. In

the approach of *user-based collaborative-filters*, users with items rated similarly, will be recommended the same, or similar items. *Item-based collaborative-filters* will recommend users items that are similarly rated to items they have rated.

On Filmtipset [6], a service that utilizes *user-based collaborative-filters*, users are recommended movies liked by other users who like the same movies they do. For example: person A has given Titanic, and The Rock, high ratings and user B has given Titanic, The Rock, and Terminator, high ratings. Then user A may be suggested to watch Terminator, based on the similar preference with user B. Amazon [2] services uses *item-based collaborative-filters*. When a user views a product on the Amazon website, they will get recommendations based of a few other products called "Frequently Bought Together". These items are recommended based on what other users have bought together with the item viewed.

Content-Filtering

Lops, de Gemmis, and Semeraro [11] describes content-filtering recommender systems as trying to find similarities between items and recommend items that have high similarity to what a user usually selects. De Wit [4] describes content-filtering as having an item domain with a predefined set of features. Those features will be used to determine similarity, then recommend to users based on a user profile, created explicitly, or learned, by user interaction with the system.

A movie, for example, could be characterised by cast, genre, theme, etc. Users may then be recommended movies with the same characteristics as movies they usually watch.

2.1.2 Complications with Recommender System

Building a good recommender system often requires a large amount of data to make good predictions. Any kind of system that has such a requirement is by its nature both time, and memory, consuming. With modern hardware this is becoming less of a problem, but is still a great concern. In the first iteration of the Retailer project Expertmaker had to handle around three hundred thousand users, with an average of tens of transactions per day, each, and over a year of historical data with daily updates. In the full scale version of that project the amount of users is over twenty million. A vastly scaled down dataset with only around ten thousand users, and around a year of historical data was provided for the retailer case. This is still enough to stress a personal computer.

For recommendations domain expertise is very important. To extract features for *content-filtering* knowledge of both users, and items, can be used [4, 17]. With *collaborative-filtering* features are extracted from users behaviour, either by comparing users, or the items from a user point of view. To do this efficiently a good basic knowledge of both the user- and item- domains is required. Thorough analysis of the domains should thus be done before attempting to build the recommender system.

De Wit [4] identifies two main difficulties when building a recommender system: *cold-start* (new user or new item), and *over-specialization*. Lops, de Gemmis and Semeraro[11] also identify *over-specialization*, and *cold-start* (new user only), as well as *limited content*.

- *Cold-start* is when a recommender system cannot infer for users, or items, for which not enough information has been gathered. *Collaborative filtering* has problems handling both new users and new items while *content-based* systems only have problems with new users.
- *Over-specialization* happens when only items with high similarity scores are recommended. This effectively prevents serendipitous recommendations.
- *Limited-content* is the problem of distinguishing items users like from the ones they do not like due to lack of information of the items.

De Wit [4] also points out that recommender system results is dependent on the type of data, i.e., a recommender system that works well with a set of data that has many users, and few items, may do poorly on a set with few users, and many items. A general evaluation should be independent of the dataset.

2.2 Evaluating recommender systems

Evaluating recommender systems is generally done in three settings: *Off-line experiments* [4, 7, 20], *user studies* [20] and *on-line evaluation* [4, 7, 20]. These are explored in section 2.2.1. Each setting generate a list of relevancies. These are the relations between the recommendation and the test data. The relevancies are generally measured in dimensions [4, 7, 20]: *predictive accuracy*, *classification accuracy*, *rank accuracy* and *utility*. These are explored in section 2.2.2. In section 2.2.3 it is explored how to compare relevance lists, and in section 2.2.4 a discussion of known problems with recommender system evaluation is conducted.

2.2.1 Experimental Settings

The different approaches are generally used in different stages of the development of a recommender system [20]. Starting with *off-line experiments*, to evaluate many different recommender models at a low cost, following with a *user study* that can be conducted with a small sample of users, to further select a best model for the system, and finally a large scale *off-line evaluation* with unknowing users, to further optimize the model before full scale launch. It is advised by Shani and Gunawardana [20] that *off-line experiments* is used to select a few models for user studies, and finally conduct a full scale *on-line evaluation* using models that are validated with both *off-line experiments* and *user studies*.

Even after a model has been launched at full scale, continuous evaluation should be done to compensate for changes in the domain, and further optimize the model. This continuous evaluation is essentially the same as the *on-line evaluation* on a larger scale.

Off-line Experiments

Off-line experiments are fast, and cheap, but will generally only give a hint of a model's quantitative prediction power. Not taking into account the quality of the recommendations, nor evaluate anything of items users have not used in the historical data.

User Studies

User studies lets users knowingly try a system, and evaluate their experience formally. A user could for example be asked to rate recommendations in terms how interesting they are, how useful they are, how offending they are, etc. This is more expensive than *off-line experiments*, since users are involved, but allows deeper examination of the quality of a model, and subjective comparison between models, to be evaluated.

On-line Evaluation

On-line evaluation is a near full scale test where many users unknowingly use different models. This gives an objective evaluation of both quantity, and quality, but is much more expensive and risky than both *off-line experiments* and *user studies*.

2.2.2 Dimensions

Three accuracies are discussed by de Wit [4], *prediction accuracy*, *classification accuracy* and *rank accuracy*. Shan and Gunawardana [20] describes *accuracy of ratings*, *accuracy of predictions* and *accuracy of rankings*.

Besides accuracy there are a number of other dimensions that can be measured [4]. *Coverage* measures the percentage of items for which the recommender system can make predictions or recommendations. *Strength* and *confidence* measures how much a user will like an item, and how sure the recommender system is about that *strength*. *Diversity* measures the spread of recommendations, i.e. the opposite of limiting recommendations to only include items very similar to the ones a user has selected previously.

Prediction Accuracy

Prediction, in a recommender system context, is the predicted rating of an item. This mainly applies to systems that has, and predicts, grades of items. `www.filmtipset.se` [6] has a rating system where users grade any movie they have seen on a five point scale. Based on those ratings the website provides a guess for any movie the user has not yet rated. *Prediction accuracy* is the metric that describes the distance between the actual ratings, after the ratings have de facto been given, and the predicted ratings. Most *prediction accuracy* metrics are measures of error.

There are several different metrics commonly used for *predictions accuracy* [4, 20]. One of the most widely used is *mean absolute error* (*MAE*). *MAE* is the sum of the difference between a user's ratings ($r_i(b_k)$) and the predicted ratings ($p_i(b_k)$), and divides the result by the number of items considered (B_i).

$$MAE = \frac{1}{B_i} \sum_{b_k \in B_i} |r_i(b_k) - p_i(b_k)| \tag{2.1}$$

Mean squared error (*MSE*), and the more commonly used *root mean squared error* (*RMSE*) are the mean of the squares of all errors. *MSE*, and *RMSE*, will therefore penalize large errors more than *MAE* does.

$$MSE = \frac{1}{B_i} \sum_{b_k \in B_i} (r_i(b_k) - p_i(b_k))^2 \tag{2.2}$$

$$RMSE = \sqrt{MSE} \tag{2.3}$$

Both *MAE*, and *RMSE*, are commonly normalized by the range of the ratings. Normalized *MAE* (*NMAE*), and Normalized *RMSE* (*NRMSE*), are simply scaled down versions of their non-normalized parts, and will thus show the same result.

$$NMAE = \frac{1}{r_{max} - r_{min}} MAE \tag{2.4}$$

$$NRMSE = \frac{1}{r_{max} - r_{min}} RMSE \tag{2.5}$$

Classification Accuracy

Classification prediction is a prediction of if an item is a member of a certain class. The simplest *classification prediction* is with a binary class. With more classes the complexity is higher.

In the churn case a customer can be predicted to either be a churner or not. In the retailer case a coupon can be predicted to be either clipped or not, and a product can be predicted to be either bought or not.

A famous classification problem is the Iris flower classification [3], where the species of an Iris flower can be classified based on the physical lengths and widths of its sepals, and petals. There are three species of Iris flowers: Iris Setosa, Iris Versicolor and Iris Virginica. By training a model with the parameters, and known species, a model can somewhat accurately guess the species of new flower using only those parameters.

The accuracy in classification is the rate of true and false predictions. A confusion matrix, *cm*, is a $m * m$ matrix where rows, i, are *actual class* and columns, j, are *predicted class*. Positions in the matrix where $i = j$ is the correct classifications, $Actual_i$. All other classifications are incorrect.

As seen in table 2.1, *True Positive* (*TP*) is the number of correctly predictions of the true class. *False Positive* (*FP*) is the number of incorrectly predictions of the true class. *False Negative* (*FN*) is the number of incorrectly predictions of the false class. *True Negative* (*TN*) is the number of correct predictions of the false class.

		Prediction outcome		
		p	**n**	**total**
Actual value	**p′**	*True Positive*	*False Negative*	**P′**
	n′	*False Positive*	*True Negative*	**N′**
	total	**P**	**N**	

Table 2.1: Confusion matrix with positive-negative classification

Multiple classes can be computed similarly, table 2.2, and all metrics are calculated generally from this [4, 20]: *Precision*, *Recall* and *Specificity*. An additional classification accuracy metric that can be calculated is *Accuracy*.

Precision, the correct actual rate for a class:

$$Precision = \frac{TP}{TP + FP} \tag{2.6}$$

$$Precision_i = \frac{Actual_i}{CLASS_i} \tag{2.7}$$

Recall, the correct hit rate for a class:

$$Recall = \frac{TP}{TP + FNAnd} \tag{2.8}$$

$$Recall_i = \frac{Actual_i}{CLASS'_i} \tag{2.9}$$

$1-$*Specificity*, the ratio of false predictions.

$$Specificity = 1 - \frac{FP}{FP + TN} \tag{2.10}$$

$$Specificity_i = 1 - \frac{FP}{FP + TN} \tag{2.11}$$

Accuracy, the ratio of correct predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FNAnd} \tag{2.12}$$

$$Accuracy = \frac{\sum_{i=1}^{k} Actual_i}{\sum_{i=1,j=1}^{m,m} cm_{i,j}} \tag{2.13}$$

		Prediction outcome			
		$Class_1$	$Class_2$	$Class_3$	**total**
Actual value	$Class'_1$	$Class'_1$ $Class_1$	$Class'_1$ $Class_2$	$Class'_1$ $Class_3$	$CLASS'_1$
	$Class'_2$	$Class'_2$ $Class_1$	$Class'_2$ $Class_2$	$Class'_2$ $Class_3$	$CLASS'_2$
	$Class'_3$	$Class'_3$ $Class_1$	$Class'_3$ $Class_2$	$Class'_3$ $Class_3$	$CLASS'_3$
	total	$CLASS_1$	$CLASS_2$	$CLASS_3$	

Table 2.2: Confusion matrix with three classes

Shani and Gunawardana [20] describe a trade-off between these quantities. A longer recommendation list would typically improve recall, but it is likely to reduce the precision.

Using *classification accuracy* with recommender systems it is often useful to calculate the *classification accuracies* at a certain position, N, in a recommendation [20]. This implies a binary classification where each element before, and including, N are predicted as positive, and all after N are predicted as negative. This only works in applications where the number of recommendations to the users is preordained. In other applications it is preferable to evaluate algorithms over a range of recommendation list lengths. In this way it is possible to compute curves comparing precision to recall, Receiver Operating Characteristics (*ROC*) curves. More details on these curves, their computations, and other related metrics can be found in [4, 20].

Rank Accuracy

Sometimes the order of recommendations is more interesting than the complete recommendation list [20], i.e., when items are presented in order, and may be more numerous than any user can take in. Netflix [13], for example, will recommend movies in list that is too long for any user to get through, the most interesting movies should appear as early as possible in that list. In these cases *ranking accuracy* is a useful metric.

Shani and Gunawardana [20] describes a multitude of *reference ranking*, and *utility based* accuracy metrics . Among them are Normalized Distance-based Perforamnce Measure (*NDPM*), *Kendall's tau correlation*, *Spearman's p correlation*, *half-life utility*, *R-Score*, and *Normalized Discounted Cumulative Gain* (*nDCG*). While only *nDCG* will be

described in this report it may be useful to know of the others existence.

To understand *nDCG* it is important to understand all its parts. Ranking evaluation is based on a list of values (v) with the length $N + 1$ where p denotes the position. Position 0 is the first position, and N is the last position. Position 0 is the zero item, which is necessary to calculate cumulative gain, and discounted cumulative gain, properly. With U users, each user, denoted U_i will get a individual relevance list, *IndRel*:

$$IndRel(p) = value, \text{where value is set by the simulation} \tag{2.14}$$

An average list, *Avr*, is also useful to score and visualize a baseline:

$$Avr(v) = 1/U * \sum_{i=1}^{U} v(i) \tag{2.15}$$

Optimal sorting, *Opt*, is a sorting based on test results, i.e., the test will run first then sort the results according to the test score:

$$Opt(p) = \sum_{i=1}^{U} sorted(IndRel_i(p)), \forall p = \{1, 2, ..., N\} \tag{2.16}$$

This can be used for comparison and normalization.

If relevance had been easy to interpret no other metrics would have been necessary. All metrics mentioned above are essentially projections of the same data on different sets of coordinate systems. Relevance and gain are the same data but relevance displays an absolute value and gain a range between 0 and 1, normalized according to 2.19. Cumulative gain is, as described in 2.20, the sum of all gain values up to each index. This is the first metric that is in any way readable, but when it comes to scalable comparisons it may be inadequate.

Relevance (*Rel*) Each position in a recommended list is given a score based on the test data, seen in figure 2.1. This score can be binary (1 if the item recommended exists in the test data, 0 otherwise), binary discounting (1 if the item recommended exists in the test data, -1 otherwise), counted (number of occurrences in the test data), or something else (e.g. a time period from test date to occurrences in test data). When evaluating a model over several instances of data their relevance measurements can be averaged over all instances 2.1.

$$Rel(p) = \sum_{i=1}^{U} IndRel_i(p), \forall p = \{1, 2, ..., N\} \tag{2.17}$$

Also useful for normalization is sorted version of the relevance, *Idl*:

$$Idl = sorted(Rel) \tag{2.18}$$

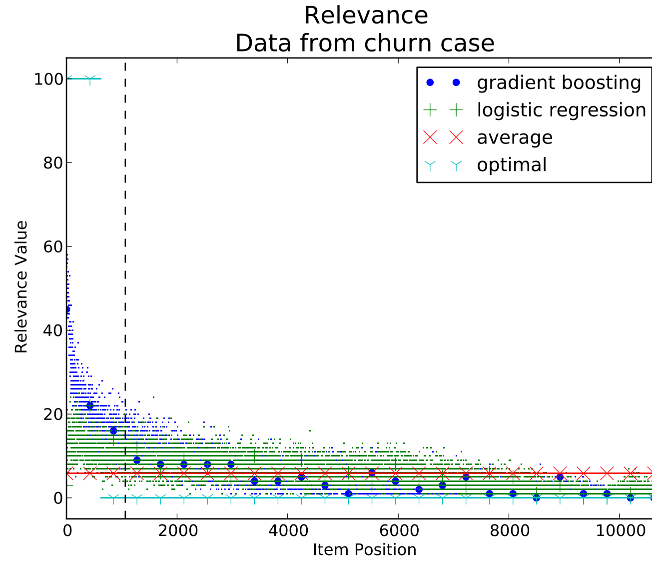


Figure 2.1: $Rel(p), \forall p = \{1, 2, \ldots, N\}$

Gain is normalized relevance, seen in figure 2.2. In the normal case, only running one instance of a recommender model, gain will be equal to the relevance. However, to evaluate models, rather than the result of a model, it is often useful to run it over several instances with different permutations of the data (e.g. separation of train and test data), and thus getting a statistically more significant result. To normalize this the relevance vector is divided by the number of instances run. With a binary, or discounted binary, relevance score this will yield a value between 0 and 1, or -1 and 1, respectively. With other counted relevance scores the gain can be any natural number.

$$Gain(v, p) = v(p)/U \tag{2.19}$$

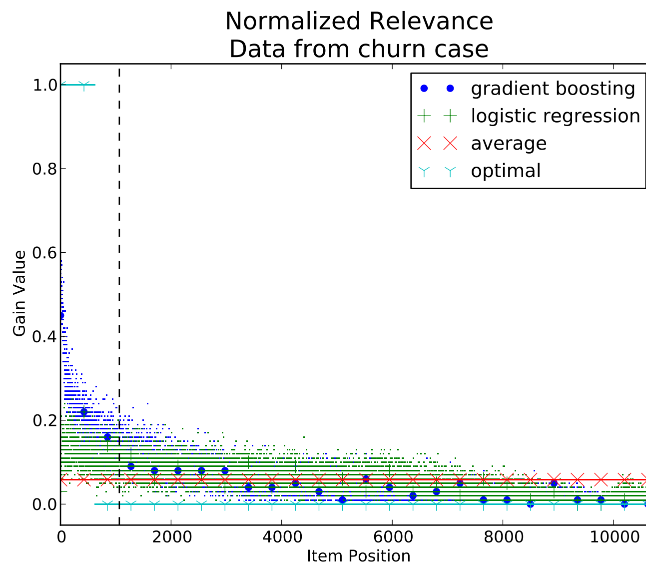


Figure 2.2: $Gain(p), \forall p = \{1, 2, \ldots, N\}$

Cumulative gain (CG) is the sum of all subsequent values of gain 2.20, seen in figure 2.3. *CG* is useful as it is becomes a fairly continuous, never decreasing curve. All *CG* computations of the same length should have the same value at the *N-th* position. A good recommendation will generally be steep in the lower positions, and curve closer to the end.

$$CG(v,p) = \sum_{i=1}^{p} v(i), \forall p = \{1, 2, \ldots, N\} \tag{2.20}$$

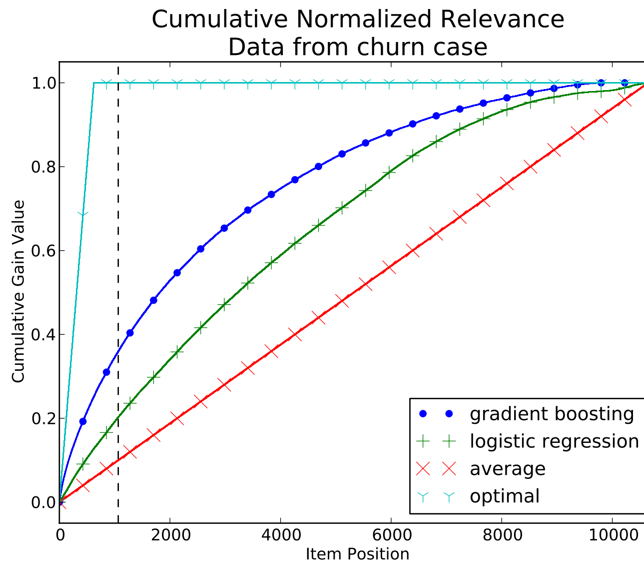


Figure 2.3: $CG(p), \forall p, 0 < p \leq N$

Lift is *CG* normalized against an average, i.e., divided by the *cumulative average* in all positions, seen in figure 2.4. The *cumulative average* is always a straight line with the *average* value as increment in each position. All *lift* computations should be 1 at the *N-th* position. As *cumulative average* is divided by itself in all positions it will always be exactly 1. A good recommendation would generally be a high multiple of the *cumulative average* in the beginning and eventually intersect with the *cumulative average*, with value 1, at the *N-th* position.

$$Lift(p) = CG(Rel, p)/CG(Avr(Rel), p), \forall p = \{1, 2, \ldots, N\} \tag{2.21}$$

Loss is, similarly to *lift*, a projection. *Loss* is projected on a *cumulative optimal* performance rather than a *cumulative average*, seen in figure 2.5. As with *lift*, all recommendations should intersect with 1 at the N-th position. A good recommendation would only drop slightly in the beginning, and soon be close to the *cumulative optimal*. As a *cumulative optimal* recommendation may not be possible to find it is common to use the *cumulative ideal* version of a recommendation. Since the *cumulative ideal* recommendation is based on a recommendation the projection is not the same for any recommendation. Dividing by *cumulative ideal* rather than *cumulative optimal* is a form of normalization.

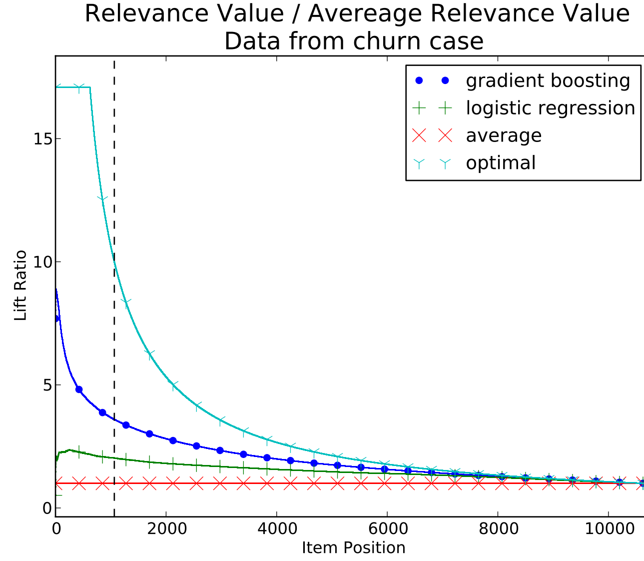


Figure 2.4: $Lift(p), \forall p, 0 < p \leq N$

$$Loss_{Idl}(p) = CG(Rel, p)/CG(Idl, p), \forall p = \{1, 2, \ldots, N\} \tag{2.22}$$

$$Loss_{Opt}(p) = CG(Rel, p)/CG(Opt, p), \forall p = \{1, 2, \ldots, N\} \tag{2.23}$$

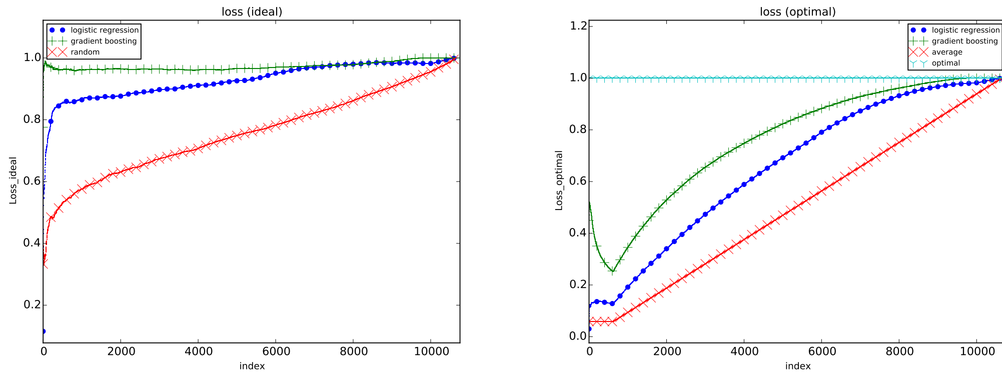


Figure 2.5: Left: $Loss_{Idl}(p), \forall p, 0 < p \leq N$,
Right: $Loss_{Opt}(p), \forall p, 0 < p \leq N$

Discounted cumulative gain (DCG) While *CG* computes a list that is easy to interpret it does not take into account the positional value in the recommendation. By discounting each position *DCG* penalizes relevance later in a list, seen in figure 2.6.

$$DCG(v, p) = v(1) + \sum_{i=2}^{N} v(i)/log_2(i), \forall i = \{2, 3, \ldots, p\} \tag{2.24}$$

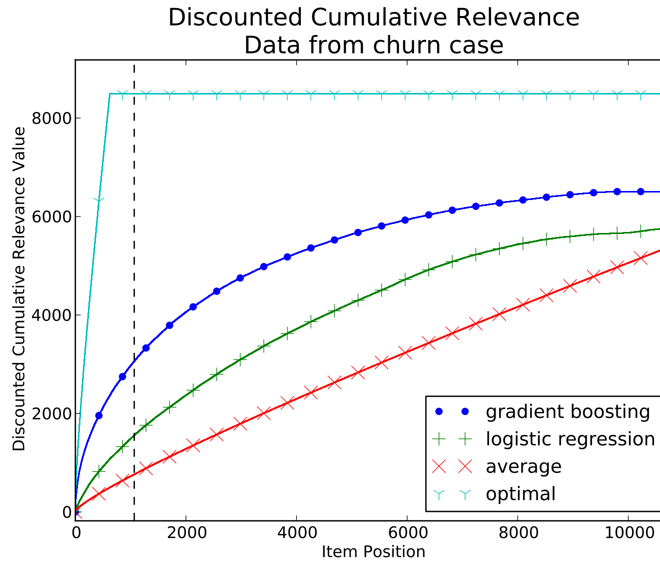


Figure 2.6: $DCG(p), \forall p, 0 < p \leq N$

Normalized discounted cumulative gain (nDCG) is like *loss* 2.2.2 a projection on a *discounted cumulative optimal*, or a *discounted cumulative ideal* recommendation, but unlike *loss*, *nDCG* utilizes *DCG* rather than *CG*, seen in figure 2.7. Further reading on *nDCG* can be found by Ravikumar, Tewari, Ambuj, and Yang [16], who discusses *nDCG* consistencty, Valizadegan, Jin, Zhang and Mao [22], who covers optimization of *nDCG*, and Wang, Wang, Li and He [25], who have conducted a theoretical analysis of *nDCG* type ranking measures.

$$nDCG_{Idl}(p) = DCG_{Rel}/DCG_{Idl}, \forall p, 0 < p \leq N \tag{2.25}$$

$$nDCG_{Opt}(p) = DCG_{Rel}/DCG_{Opt}, \forall p, 0 < p \leq N \tag{2.26}$$

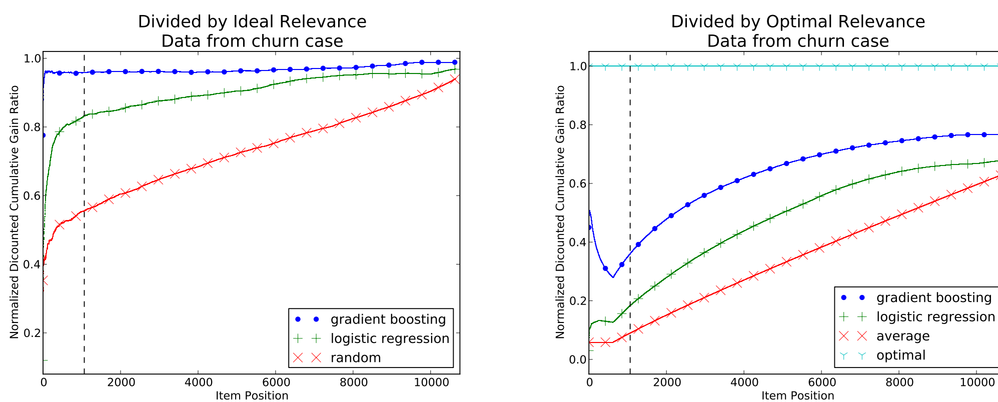


Figure 2.7: Left: $nDCG_{Idl}(p), \forall p, 0 < p \leq N$,
Right: $nDCG_{Opt}(p), \forall p, 0 < p \leq N$

2.2.3 Comparison

Shani and Gunawardana [20] explains that between any pair of recommendations two comparisons can be taken into account: *confidence* and *average*.

Confidence

Confidence is computed as a per user performance measure. Given a pair of recommendations A and B, from different recommender models, the number of times A outperforms B is denoted n_A and the number of times B outperforms A is denoted n_B. The significance level is the probability that A is not truly better than B. Confidence is estimated as the probability of at least n_A out of $n = n_A + n_B$, 0.5-probability binomial trails succeeding [20], and is given by:

$$p = (0.5)^n \sum_{i=n_A}^{n} \binom{n}{i} \tag{2.27}$$

This approach shows whether A outperforms B with a 95% confidence, but does not show by what magnitude.

Average

Evaluating the magnitude by which a model outperforms another another can be done by comparing each model's average performance. With enough users an expected value (E) and variance (V) in the performance can be calculated, and a stochastic comparison of magnitude can done.

2.2.4 Complications with Evaluation

Recommender evaluation is dependant on the accuracy of the data. *Off-line experiments* rely on historical data and are subject to changes in the user base. In cases where personal predictions are made, such as in Expermaker's retailer project, the system evaluation depends on users' change of opinions and whims. It has been suggested that a recommender model cannot be more accurate than the variance in the users' habits [7].

Off-line experiments cannot take into account new items, or users who have no history. Introducing a new item that does not exist in the test data will not give it any score by itself. A serendipitous item will thus be neglected despite the possibility of being liked, and can only be tested with a *content-based* similarity score. Models promoting such items will more than likely perform worse in an *off-line experimental* setting. These items have been shown to be important for recommender models [20].

In *user studies* evaluation must trust the congruency of user scores, some users may rate all questions the same, while others may rate arbitrarily. To counter this methods to detect incongruous behaviour can be applied, and remove those users. One such method is to provide contradictory questions. Additional complication with *user studies* come from the users' interpretation of the grading. Normalization of users' grades can be done to allow comparison between users.

Chapter 3
Approach

While the main task is to set up a platform that can be used to evaluate any recommender system with only minor work, there are a few sub problems that need attention. The idea used to build a generic platform is to use a modular, and configurable, framework. Section 3.1 introduces the philosophical approach taken, and describes the problems encounteredt. This chapter also formally introduces the churn 3.2 and retailer 3.3 cases.

The two cases, churn and retailer, are useful as comparison as they need different approaches. The churn case targets a single recommendation, and can easily be compared to a classification problem. The retailer case need to handle multiple personal recommendations, and must be evaluated globally with the individual recommendation evaluations as base. Both can be viewed as ranking problems, and can thus be effectively evaluated with the same metrics.

3.1 Philosophy

The requirements for a generic evaluation platform for recommender systems have to be separated into a few sub problems. Modularity, described in section 3.1.1, is a first step of making sure the framework can be used to solve different types of problems. Section 3.1.2 introduces the approach taken to evaluate recommender systems.

3.1.1 Modularity

Modularity is not a problem with a specific result, but rather a problem which needs to be solved to evaluate different recommender problems. By building the framework modularly, different recommender problems can be approached with vastly decreased overhead work. Exploring the design and implementation of the framework, with two different cases, the

churn case, and the retailer case, kept in mind, the parts of the framework that can be generic, and the parts that have to be modular, can be identified.

To make a framework accommodate different types of problems, the problems, and their possible differences, must first be identified. The churn case is actually a classification problem, for which the most probable churners, customers who cancel their contract with the service provider, can be recommended. The retailer case is a multiple recommendation problem. Each customer should be recommended a list of products they may be interested in. The singular recommendation versus the multiple one is one of the main differences identified between the two examples.

3.1.2 Average

Evaluating a recommender system in practice is by its nature not deterministic. The test data is generally biased towards the recommendation. A user is less likely to select an item that has not been recommended, and evaluating a system which already has a recommender, off-line, without using that recommender, will therefore produce a systemic error. A replication of the actually used model will likely get a better result than it should.

The absolute knowledge of the churn case, with no dependencies on previous schemes to change the outcome of the recommendation, is another important factor to consider. The retailer case depends strongly on the recommendation system in place, i.e., the customers behaviour is affected by what they have seen. The approach taken in this project is to simulate the churn case as a multiple recommendation problem, more similar to the retailer case.

Cross-validation [9] is used to motive the accuracy of an average evaluation. Through comparison of a classification problem, the churn case, as a recommendation with a simulated average of the same problem. It must be shown that the average can correspond to a single evaluation. Applying a naive recommender on the data of the churn case, with the classification probabilities as weights, a predictable result can be retrieved. By using 100-fold[1] cross evaluation it can be shown how average corresponds to the single validation. An $N-$fold cross evaluation, unlike a single validation, would also allow computation of confidence.

As a proof of concept the average evaluation method will then be applied to the retailer system, where we have no method of singularly evaluating the system.

[1] $10-20$-fold should be enough, according to Kohavi [9], but as there is no significant performance reduction using 100-fold it is deemed good enough to use.

3.2 Churn

Readily available for this thesis to use as an example is the churn dataset that Johan Wensäter used in [26]. The churn dataset is a clear cut case, with a training period, and known result. A churning customer is a customer who cancels a plan, most likely to move to a different provider. The churn case is essentially a classification problem, where the goal is to find which customers are going to churn. By training a classification model on a subset of the customers, with known result, whether or not the remaining customers will churn, can be predicted. This can then be tested against the knowledge of their actual churning. The churning example is a good platform to start exploring a recommender problem, as this classification problem can easily be transformed into a recommender problem by recommending customers predicted, i.e. classed, as churners.

Wensäter describes the churn dataset in [26]:

> "One of the most common business problems involving statistical classification and machine learning is within an area called customer churn prediction. The term churn, most widely used in the telecommunication industry, describes a customers tendency to cancel a service at the end of a contract. What makes churn such a hot topic is the fact that the western market is saturated, with many vibrant competitors fighting to win customers over. The harsh competition results in a five to six times higher cost associated with acquiring new customer, compared with retaining old. In general, long-term customers are also associated with many beneficial attributes such as higher profits and less likely to be influenced by competitive marketing [24]. These factors contribute to the case that small improvements in customer retention lead to significant increases in profits [23]."
>
> "With this in mind, it is not hard to understand the interest in using machine learning algorithms on customer data in order to find a predictive classification model capable of identifying the most likely churners among customers. With such a model at hand, a telecommunication company (operator) can take preventive actions such as calling the most likely churners with a special offer to try to retain them. Even if most customers can not be persuaded the effort is still worthwhile considering the costs of attracting new customers and the benefits of retaining as many as possible."

3.2.1 Data

The churn data found in appendix A consists of a set of around a hundred thousand customers, with customer data and their activity over the period of three months, August until October 1997, and whether or not they churned from the service within four months after that period. The customer data are features describing the customer. The activity data are features describing customer usage. Lastly is a binary feature telling weather a customer churned or not within the four month period following the data sampling period. More on this specific churn dataset can be found in [26].

3.2.2 Recommender Purpose

The target is to create a recommendation of customers for marketing focus. By predicting what customers are most likely to churn the operator gets an idea of who to contact to prevent churning. A theoretical scenario for using the churner recommender system could be described as follows:

The total amount of customers, 106495, and the number of churners are 6231, with $p = \frac{106495}{6231} \approx 5.9\%$ as the average probability that a customer is a churner. Using 90% of the dataset for training, and 10% for prediction and testing. The test is done on around ten thousand customers, $N \approx 10000$. The expected number of churners for that test would be around 600, $X = N * p = 10000 * 5.9\% = 590$ The operator is assumed to budget a scheme, with a budgeting modifier $k = 2$ to contact $X * k = 590 * 2 = 1180$ customers, to increase the chance that all churners are contacted without having to contact all customers.

3.3 The Retailer

The retailer is one of the biggest grocery store chains in the United States. To attract new, and keep old, customers they provide discount coupons. New coupons are created a couple of times every week, and the lifetime of a coupon is generally between one week to a few weeks. To make it easy for customers to see, and choose, coupons, they provide a mobile application, and a web interface. In both the mobile, and the web interface, users can clip (a metaphor for actually clipping a coupon from a magazine) any coupon in the interface. One of the views in the interface is based on a personalized recommender system that tries predict which coupons the user is most likely to use. It shows a sorted list with the most likely at the top. The goal for the recommender system is to get as many clips as possible.

3.3.1 Data

The available data for this project is transaction history for around seven thousand users between January 1st 2013 until December 31st 2013. This data contains user ids (called household ids), and receipts, which in turn contains product id (called UPC or Universal Product Code), and time stamp. Household ids are anonymous to protect the integrity of the users, and the retailer.

Users in the dataset are uniformly distributed in terms of activity from all of the retailer users. The data available from this project is sampled from test group available to Expertmaker for their development. Around three houndred thousand households are included in this development set. Sampling was done by randomly taking around seven hundred users from each of ten groups, split by number of transactions. The total set used for this project contains 6553 households.

The item mapping contains over two and a half million products. A preprocessing is done to reduce the list of products to only include those that exist in any receipt in the transaction history. This turns out to be about fifty thousand products, where the bulk of the transactions were represented by only 10% of them.

3.3.2 Recommender Purpose

The real system Expertmaker is creating for the retailer recommends coupons. These coupons can be a discount for any single product, but is often a discount for a set of products. Coupons can be national, local for region, or for a single store. The recommendations is a personalized list list of coupons relevant for each individual customer. Customers are assumed to view only around five to a hundred coupons, with an average of about twenty.

To simplify the task of the example recommender system to build for this project the requirements are loosened. Instead of coupons the system recommends products. The recommender system produces a list of recommended products in order of relevance, for each customer. For this purpose it is assumed customers will purchase around one hundred products on average. The total list recommended contains around one thousand products. While this does not exactly correspond to the project Expermaker is doing, it is an approximation of such a project.

Chapter 4

Implementation

This chapter describes the implementation done for this project. Section 4.1 introduces the reader to the programming language chosen, and some of the machine learning packages used, for this project. Section 4.2 describes the framework implemented for recommender system evaluation. Sections 4.3 and 4.4 describes the specifics for the churn and retailer respectively.

4.1 Development

This section introduces the python scripting language, and the available machine learning packages that is available with it.

Since the language used at Expertmaker is Python 2.x [15], it was the obvious choice for this project as well. All implementation of this project executes with Python 2.7. Python is a scripting language with powerful, and easy to use, ways for handling lists, dictionaries, and tuples. There are useful packages available for numerical computations, NumPy [14], and graph generation, pyplot in matplotlib [12].

For machine learning the scikit-learn [19] has been used. Scikit-learn contains implementations of multiple different classification, regression, and clustering models. Recommended by the Expertmaker supervisor, a few of different models are used for this project: among the linear models *Logistic Regression*; among the ensemble models *Random Forest*, *Extra Trees Classifier* and *Gradient Boosting Classifier*; and among the neighbors models *Nearest Neighbors*.

4.2 Framework

To fully experiment with, and understand, recommender systems, and their evaluation, a modular framework was constructed from scratch. The background on which the framework is built is explored in the section 4.2.1. In section 4.2.2 the sequence of the project framework is described, and in section 4.2.3 its modularity is explained. Section 4.2.4 describes configuration of the framework, and finally section 4.2.5 describes other utilities in the framework.

4.2.1 Recommender System Evaluation

A recommender system, in its simplest form, can be seen as a sorting function. The input is an unsorted list of items, the allocation, *A*, and the output is the sorted list, the recommendation, *R*. The recommended items are sorted by importance, from most to least important:

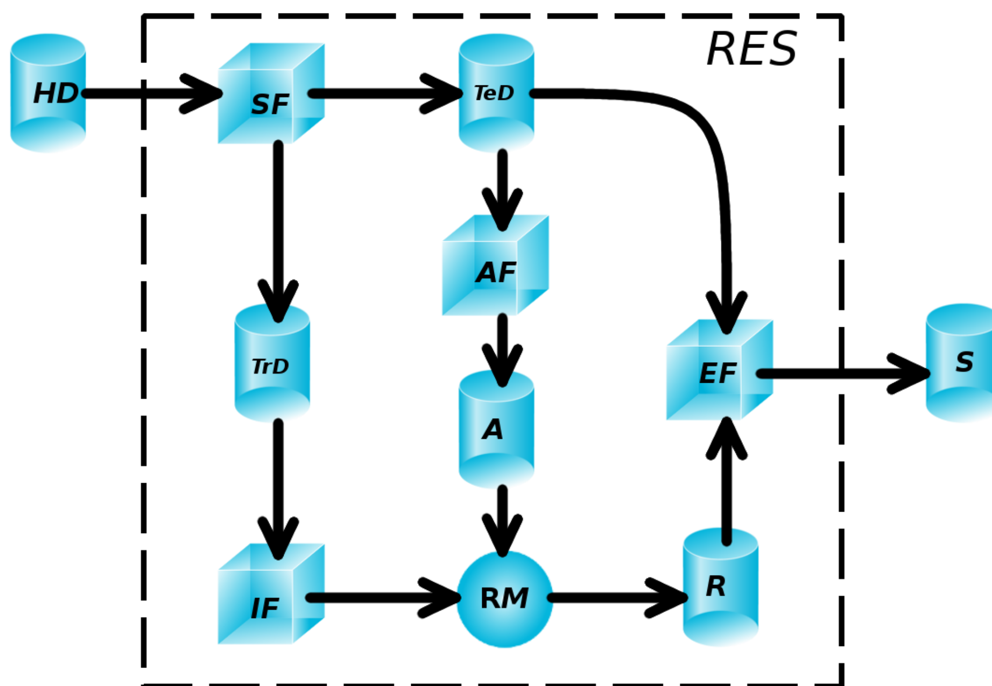


Figure 4.1: Recommender Evaluation System, *RES*, flow chart

$$RM(A) \Rightarrow R \tag{4.1}$$

The recommender model, seen in figure 4.1, is usually a dynamically updated system based on some training data, *TrD*. The model is created by an initialization function, *IF*:

$$IF(TrD) \Rightarrow RM \tag{4.2}$$

In terms of evaluation, the resulting recommendation is compared to some test data, *TeD* with an evaluation function, *EF*, which result in some score, *S*. The test data can be collected from a live system, a user study, or a subset of historical data. The score can be a value such as precision, or recall or a list of ranking scores such as gain, or *nDCG*. *EF* can thus be written:

$$EF(TeD, R) \Rightarrow S \tag{4.3}$$

When using a historical data, *HD*, for both training, and testing, the data must be split into training, and test, subsets. How to do the split is itself an important step, and can be done in several different ways. It's preferable to find a natural split, such as a cut-off in time. If that is not possible the split can done randomly, or better with a balancing factor that ensures ratios, such as classification target values between items, are the same in both training, and test data. A split function *SF* can be written:

$$SF(HD) - > TrD, TeD \tag{4.4}$$

The allocation is then in turn extracted from the test data with an allocation function, *AF*:

$$AF(TeD) \Rightarrow A \tag{4.5}$$

Combining all 4.1, 4.2, 4.3, 4.4, 4.5 is essentially a complete recommender evaluation system, *RES*:

$$RES(HD) \Rightarrow S \tag{4.6}$$

To get a reasonably significant result the evaluation should be done for multiple users. This may force each step to be done for each user. Even when it is not necessary to re-run the data split, and model creation, the manual work to evaluate multiple models with multiple users is at best tedious.

4.2.2 Sequence

Given the steps for creating a recommender model, and evaluating it, it is not unreasonable to create a framework. Steps can be taken to do everything in one sequence. Considerations are taken to allow configuration; storing, and saving, of partial results for faster re-run; and having multiple, and modular, ways of inter-comparisons between models. A sequential process, seen in figure 4.2, with the following steps is introduced:

- XML validation & internal configuration creation
- Data extraction and cleaning
- Sampling
    - Sampling training and test data

  - Sampling users
  - Allocating items

- Model construction
- Model execution
- Simulating user behaviour
- Evaluating model result

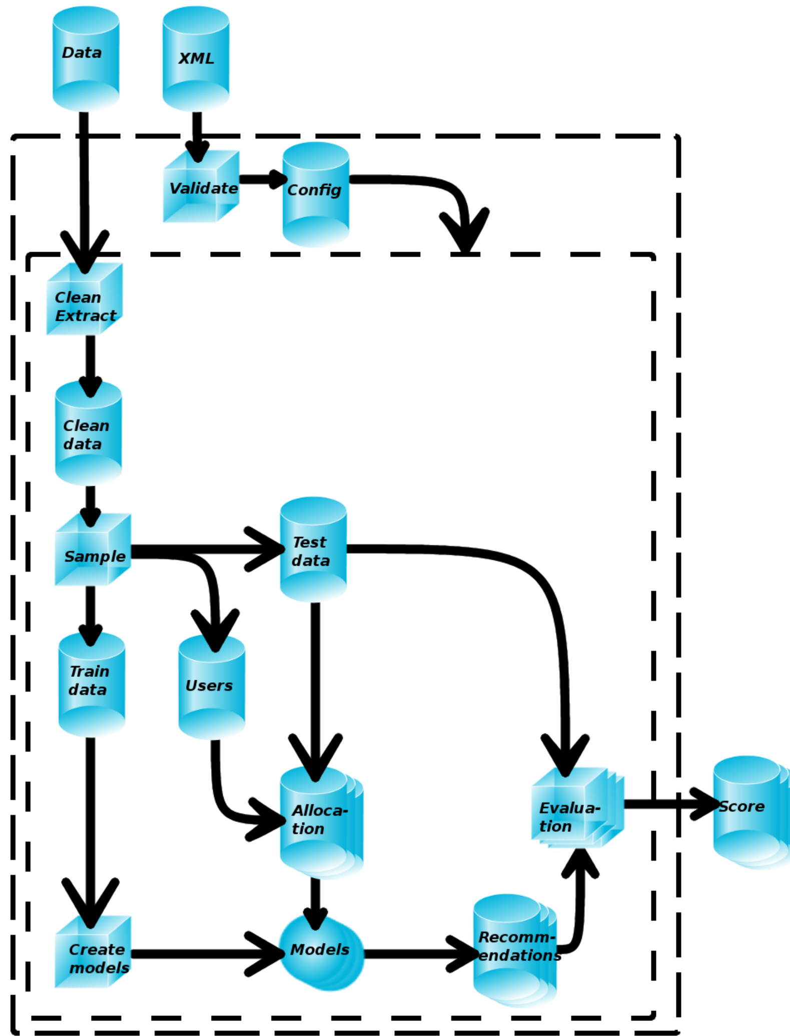


Figure 4.2: Framework flow chart

XML validation

To allow for small changes in this system, like changing model, sampling, or evaluation method, a configuration XML file is loaded when starting the framework. This also allows for easy saving and loading of previously executed steps by comparing hashes of sequence sections in configuration with saved section. Once loaded, the configuration is checked to control whether it contains all essential information.

Data extraction and cleaning

As with most data from external sources, the data used need to be preprocessed. This section removes unwanted information, as well as optimizes the data for fast iteration. Extraction, and cleaning, are described in further detail in 4.3.1, and 4.4.1, for the churn, and the retailer datasets, respectively.

Sampling

Sampling is done in three steps to set up a modular workspace that can be run with different training data, test data, users, and allocations.

- Sampling training, and test data
- Sampling users
- Allocating items

Sampling is described in further detail in 4.3.2, and 4.4.2, for the churn, and the retailer datasets, respectively.

Sampling train- and test- data To separate training and test data, specific methods for a given dataset must be implemented. This can be done as random extraction, or by separation in continuity, for example in time, as in the retailer case 4.4.2. Too exhaustive data can also be limited by extracting a portion, or too limited data can be extended by generating different permutations of the data. Generally a dataset is separated into a large training set, and a smaller test set.

The training data is directly tied to the model construction, while the test data is directly tied to the evaluation. They must as such be constructed to suit their respective purposes. With different models, and simulations, several different topologies may have to be constructed, as is the case with the retailer described in 4.4.2.

Sampling users Users are sampled to correspond to a training- and test- pair. A unique model will eventually be generated for each user, and the evaluation is done on a per user basis. In datasets where there are many users, a sample may be selected to run the evaluation system in a reasonable time, and memory, frame.

Allocating items The items allocated as input for the recommender model is a subset of the test data without any additional information. The subset can essentially be anything between a single item, and every single item, in the test data.

Model execution and testing

Models and evaluation are run in sequence for each user in an arbitrary order. Each user has an individual allocation that is provided to the model along with the user id. The model provides a recommendation for that user that is passed along, with the user id, to the simulation and evaluation method.

Each model is executed with data, and parameters, according to the configuration. The result is a sorted list of items with weights given by the model.

Simulation is done by giving each item in the test data a value. The value is created by individually scoring each item in each user's recommendation list, and adding them up per position.

Simulation is described in further detail in 4.3.3, and 4.4.3, for the churn, and the retailer datasets, respectively.

Evaluate model results

Since both the churn case, and the retailer case, are recommendations with preordained lengths, ranking accuracy was selected as focus for evaluation. *Lift*, and *nDCG*, were selected among the available metrics. *Lift* was selected because it has been frequently used at Expertmaker, while *nDCG* is a natural development of *Lift*, but also takes position into account. The *classification accuracy* metrics, *Precision*, *Recall*, *Specificity*, and *Accuracy* are also computed.

While *Lift*, and *nDCG*, computes lists that can be displayed as graphs, they can also provide a single metric value. This is a value at a certain preordained position, decided based on knowledge of the data. The *classification accuracies* are computed at the same preordained position. By considering all items before, and including, that position to be predicted as the positive class and all items after to be predicted as the negative class.

4.2.3 Modularity

Each step is essentially modular, and can be replaced independently. There is, however, dependence between the training data, and the model creation. The purpose of this is to allow usage of completely different datasets, implementations of different models, and evaluation, for each dataset.

Visualization

Each evaluation method computes a graph with all models used, and creates a numeric score for them. Scores are normalized by dividing each numeric score by the lowest score, e.g. the lowest score is 1 after normalization, and all other scores are larger than one resulting in a simple comparison.

The purpose of evaluation is to compare two or more things. When comparing recommender systems the specific purpose is to find which algorithm produces the best recommendations. This could be the comparison between several significantly different candidate algorithms, or the comparison of one algorithm with different parameters. Two important ways of comparisons have been identified for this project. A visual comparison: a figure, or graph; allows a developer a quick overview of the evaluation. It is not difficult

for a human to determine difference to a certain degree. To automatically compare, a single numeric value is the most straightforward. By calculating a score for each evaluation, they can be sorted from best to worst.

The drawback with visual comparison is that it can be unclear what actually is better. Two or more algorithms can give very similar results, making it difficult to determine which one is better. A single score makes it easy to sort algorithms and see exactly which one is better, but the score will always depend on, and be limited by, the method of scoring.

4.2.4 Configuration

Configuration is written as *XML* in a file that is called at execution, as a parameter. Several configuration files can be called in sequence:

```
$ python app.py config/config_1.xml config/config_2.xml
```

Each configuration requires setting the path to the framework, and the path to the data repository. Implementation of an import function, specific for each case, defines how to find data within the data repository. Additionally each part in the sequence stores results in the data repository, allowing loading import, and sample, results without the need to execute the functions each time. In the configuration it is also defined which sample methods, with parameters, to use. Sampling is separated into data sampling, separation of training and test datasets, user sampling when doing multiple personal recommendations, and offer sampling, i.e., creating allocation as input for models.

Finally, and most importantly, it is defined in the configuration file what models to use, evaluation methods, and how to display scores. Models are configured with references to the data necessary and parameter values.

A configuration file template can be found in appendix B

4.2.5 Utilities

A logging system was also created to allow easy access to execution meta data.

4.3 Churn

As a single recommendation problem the churn case must be extended to simulate multiple users. By extracting multiple permutations of the data it can be executed as if there were multiple recommendations.

4.3.1 Data cleaning

The churn input provided by Wensäter [26] is remodelled to a numpy array, needed for the scikit-learn models used.

4.3.2 Sampling

Extracts subsets of the data for N-fold cross evaluation. Both with, and without, replacement is implemented. After experimentation it was decided there is no benefit to not using replacement. Fractions of the data can be selected for training, and test, which will be repeated N times. Each permutation is the user data, and the test data is directly applied as the recommender allocation.

Models

The following models, from scikit learn classification models [19].

- *extra tree* [19]

  is a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

- *gradient boosting* [19]

  builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage n_classes_ regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

- *logistic regression* [19]

  implements regularized logistic regression using the liblinear library, newton-cg and lbfgs solvers. It can handle both dense and sparse input.

- *nearest neighbors* [19]

  provides functionality for unsupervised and supervised neighbors-based learning methods. Unsupervised nearest neighbors is the foundation of many other learning methods, notably manifold learning and spectral clustering. Supervised neighbors-based learning comes in two flavors: classification for data with discrete labels, and regression for data with continuous labels.

- *random forest* [19]

  is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

Three additional models are used, *random*, *average*, and *optimal*.

- *random* will give random weights to each item
- *average* will score exactly average for every item for every user
- *optimal* will score optimally for every user

4.3.3 Simulation

Each recommendation is scored according to the known result in the test data.

4.3.4 Evaluation

Evaluation is set at 10% of the total data. This is arbitrarily selected by the author, but as the amount of churners is known to be around 6% it stands to reason a margin of error is useful for the result. It is imagined that the mobile phone operator is willing to budget money to contact 10% of its customers to reduces the amount of churners.

4.4 The Retailer

Without having any pre-made recommender systems to use it was necessary to first create a few recommender models. These models are based on *collaborative filters*. In the real use of the system a set of coupons is provided to a recommender system which outputs individual, per household, sorting of the coupons, excluding all coupons that have already been used. Since the coupons data was limited to only two months (November and December 2013), and the transaction history covered a longer period (January until December) it was decided to approach the problem using only the transaction history data.

4.4.1 Data cleaning

- date to transaction
  A data structure with a mapping system $date \rightarrow hh_id \rightarrow upc_id$
- user to transaction
  A data structure with a mapping system $hh_id \rightarrow date \rightarrow upc_id$
- items
  A data set with all existing upc_ids

4.4.2 Sampling

The following explains the configuration, and the details of the specific configuration for the results in section 5.2.

Sampling training and test data

For each user the clean data consist of a list of dates with items bought at that date. In the configuration two date intervals can be set, one for the training data, and one for test data. Allowing for arbitrary training, and test, intervals allows for short tests as well as more than one training, or test, interval, within the period of the data, without overlapping. For the results in section 5.2 the training set is from 2013-01-01 until 2013-09-30, and the test set is from 2013-10-31 until 2013-11-30.

Sampling users

Users can be sampled at random, or using the full set of 6553 users. The results in section 5.2 includes all users.

Allocating items

The item data include more than two hundred and fifty million items. From this a set of around fifty thousand items has been extracted. Those are all items that had been used at any time in the historical data. Even this is too much to process in a reasonable amount of time, on the machines provided, and most items are rarely used. To make a useful, and manageable, dataset of items for the recommender systems a sample was needed. Three ways of sampling items is implemented:

- Random
  Selects N items randomly from the item dataset.
- First
  Selects the N most common item from the item dataset.
- All
  Selects all the, around fifty thousand, items in the item dataset.

Models

All models take a list of offers as input, and return the sorted list, as well as the weights used for sorting internally.

- *mostbought* is a global model produces the same recommendations for all users. It sorts the allocated items after what is most bought by all users over the train period.
- *individualmostbought* is an individual model that sorts the allocated items after what is most bought by a user during the train period. Any item that is not in the users individual history is placed after the individual recommendations, and sorted by the mostbought model.
- *randomweights* generate random weights for each item and sorts them accordingly.

4.4.3 Simulation

Three user models are implemented to simulate user behaviour, *exist_value*, *sum_value* and *time_value*, described below. The basic ranking value list (individual relevance list) can be considered both individually per user, or for all users (combined relevance list). Each simulation tries to model user behaviour based on historical test data. When time is considered the first date in the test period is considered the test date.

- *exist_value* scores items if they exist in the test period for each user.
- *sum_value* scores items how many times they exist in the test period for each user.
- *time_value* scores items based on temporal distance from test date in the test period for each user.

4.4.4 Evaluation

Evaluation is set at 10% of the total data. This is arbitrarily selected by the author, but it is deemed reasonable that a user would buy around a hundred different items, $1000 * 10\%$, on average during the one month test period a store.

Chapter 5
Results

This chapter presents the results of the executed implementations. The churn case 5.1 is done in two parts: training the models one time, and evaluating their recommendations against a validation set; running the same models multiple times with different permutations of the test data, and evaluated against a corresponding test data. The retailer case in section 5.2 shows the results of evaluating the more complicated system.

5.1 Churn

The churn case shows that the framework can be run with different types of data, and with different models suitable for the data without excessive amount of work. It also shows that an average evaluation of multiple recommendations gives results similar to, and essentially the same relative score as, an evaluation of a single recommendation. Since the different ranking evaluations used are only transforms of the *gain* value only two, *lift*, and *ideal nDCG*, are shown. They tend to be the most clear displays and are still slightly different, *nDCG* has a discount function, while *lift* does not. A table with the *classification* metrics; *precision*, *recall*, *specificity*, and *accuracy*, at 10%, is also displayed for each execution.

Section 5.1.1 shows the result from running a single validation execution. Section 5.1.2 shows the results from running the evaluation with 100-fold cross evaluation.

5.1.1 Single Evaluation

This section shows results from a single validation execution. The model has been trained on the full dataset, that in section 5.1.2 has been split into multiple training, and test, subset pairs. Evaluation has been done against a validation set that was isolated before import.

Scores at the 10% target of 5.1:

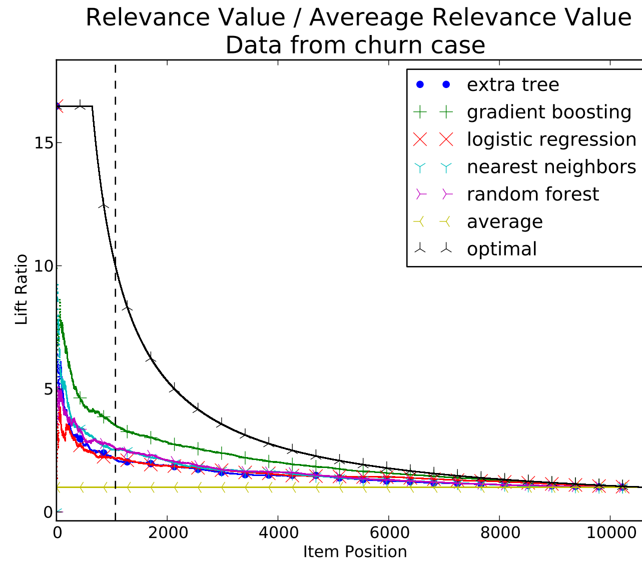


Figure 5.1: *Lift* for churn validation

- *extra tree* value at position $Lift(1065) = 2.1362$
- *gradient boosting* $Lift(1065) = 3.5294$
- *logistic regression* $Lift(1065) = 2.1672$
- *nearest neighbors* $Lift(1065) = 2.5387$
- *random forest* $Lift(1065) = 2.5697$
- *average* $Lift(1065) = 1.0000$
- *optimal* $Lift(1065) = 10.0000$

By displaying *average*, and *optimal*, a good frame is set for the contending models. All models are expected to perform better than *average* and worse than *optimal*. By using *average*, and *optimal*, as references a reasonable performance can easier be seen.

Scores at the 10% target of 5.2:

- *extra tree* $nDCG(1065) = 0.2218$
- *gradient boosting* $nDCG(1065) = 0.3501$
- *logistic regression* $nDCG(1065) = 0.2106$
- *nearest neighbors* $nDCG(1065) = 0.2573$
- *random forest* $nDCG(1065) = 0.2530$
- *random* $nDCG(1065) = 0.0927$

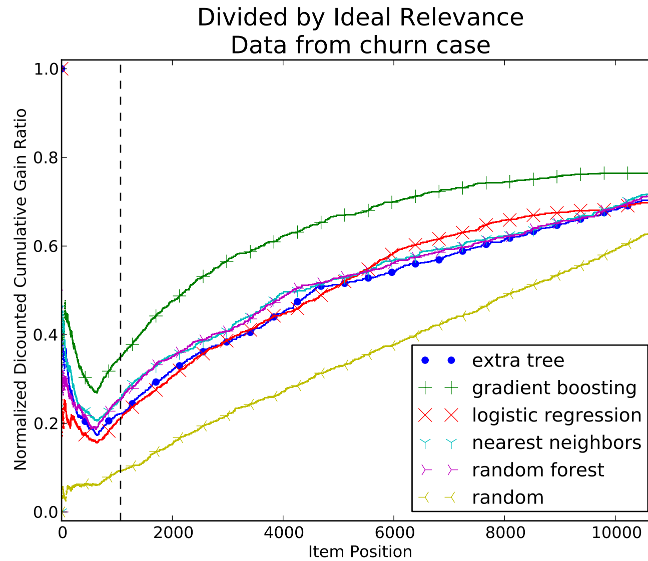


Figure 5.2: *nDCG* for churn validation

Since *average* has the same values in every position, and *optimal* is perfectly sorted, they will both be equal to their ideal sorting. Equality with ideal means *nDCG* will give a perfect result, i.e. the value will be 1 in every position. This makes them unusable as reference. To compensate for this a *random* sorting is provided as reference instead.

Module	Precision	Recall	Specificity	Accuracy
nearest neighbors	0.254	0.154	0.910	0.870
average	0.100	0.061	0.900	0.851
random forest	0.257	0.156	0.911	0.871
extra tree	0.214	0.130	0.907	0.865
randomweight	0.105	0.064	0.900	0.852
gradient boosting	0.353	0.214	0.916	0.882
logit reg	0.217	0.131	0.907	0.866
optimal	1.000	0.607	0.958	0.961

Table 5.1: Classification accuracy for churn validation

It can clearly be seen from the *Lift* figure 5.1, the *nDCG* figure 5.2, and the *classification accuracy* table 5.1, metrics, that *gradient boosting* performs the best among the models. *Extra tree*, *nearest neighbors*, *random forest*, and *logistic regression*, performs very similarly, in particular in the highest ranks. At 10% there is a difference between those models, but it is difficult to determine the significance of that difference. With the *optimal* model *accuracy* metric it can be seen that 95.9% of the first 10% items are relevant.

5.1.2 Multiple Permutations Evaluation

This section shows the results of running a 100-fold cross evaluation of the churn case dataset. The models has been trained on 90% of the dataset, and tested against 10%, with 100 randomly split permutations of the data.

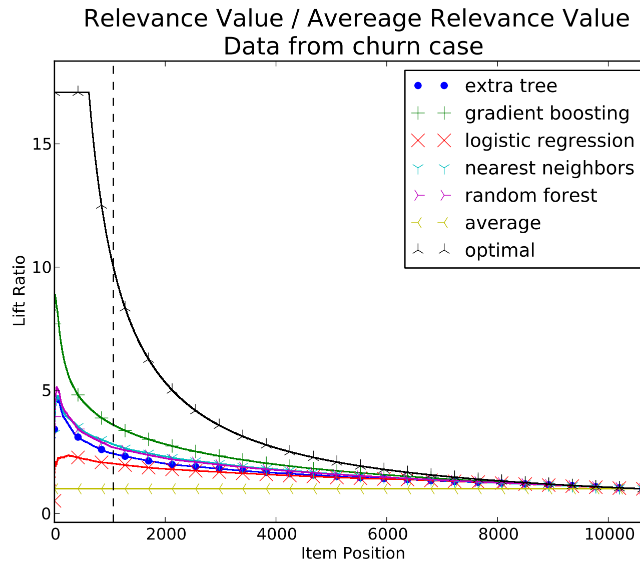


Figure 5.3: *Lift* for churn 100-fold cross evaluation

Scores at the 10% target of figure 5.3:

- *extra tree* $Lift(1064) = 2.4289$
- *gradient boosting* $Lift(1064) = 3.5953$
- *logistic regression* $Lift(1064) = 2.0277$
- *nearest neighbors* $Lift(1064) = 2.8027$
- *random forest* $Lift(1064) = 2.6822$
- *average* $Lift(1064) = 1.0000$
- *optimal* $Lift(1064) = 10.0085$

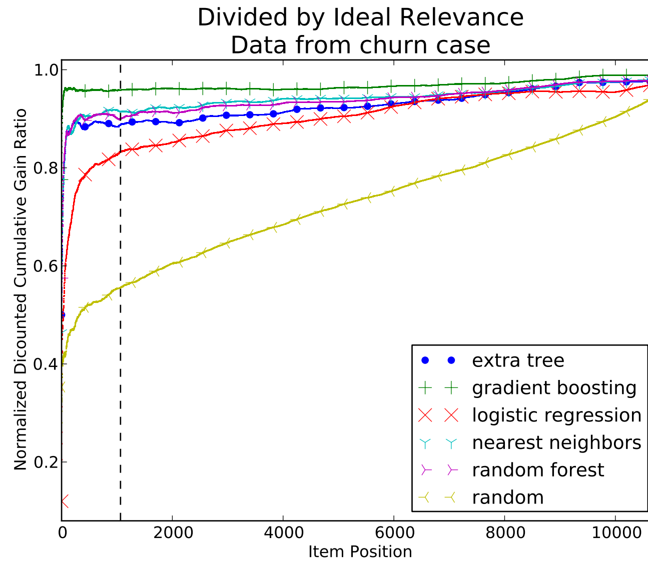


Figure 5.4: *nDCG* for churn 100-fold cross evaluation

Scores at the 10% target of figure 5.4:

- *extra tree* $nDCG(1064) = 0.8874$
- *gradient boosting* $nDCG(1064) = 0.9586$
- *logistic regression* $nDCG(1064) = 0.8325$
- *nearest neighbors* $nDCG(1064) = 0.9147$
- *random forest* $nDCG(1064) = 0.9003$
- *random* $nDCG(1064) = 0.5552$

As with the validation set *gradient boosting* performs better than the other models, seen in figures 5.3, and 5.4, and table 5.2. The main difference that can be observed between validation and 100-fold cross evaluation is that *logistic regression* tends to perform worse, in the later case. Using the cross evaluation method it is possible to compute the *significance* of the difference between models, as well as the *confidence* by how often a model outperforms another. This is, however, at this point not implemented in this system.

Module	Precision	Recall	Specificity	Accuracy
nearest neighbors	0.280	0.164	0.911	0.874
average	0.100	0.059	0.900	0.853
random forest	0.268	0.157	0.911	0.873
extra tree	0.243	0.142	0.909	0.870
randomweight	0.100	0.058	0.900	0.853
gradient boosting	0.359	0.210	0.916	0.884
logit reg	0.203	0.119	0.906	0.865
optimal	1.000	0.586	0.956	0.959

Table 5.2: Classification accuracy for churn 100-fold cross evaluation

5.2 The Retailer

This section shows the *nDCG*, and *classification accuracy*, results for an execution of the framework with all 6553 users, and 1000 items at the 100 position for the three models, "*mostbought*", "*individualmostbought*", and "*randomweight*". The models are described in section 4.4.

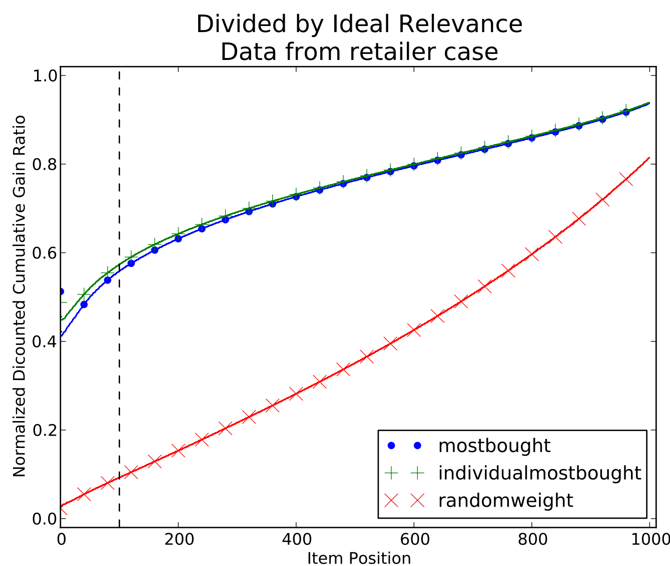


Figure 5.5: $nDCG$ for three models with all users and the 1000 most common items

Scores at the 10% target of figure 5.5:

- *mostbought* $nDCG(100) = 0.5577$
- *individualmostbought* $nDCG(100) = 0.5727$
- *randomweight* $nDCG(100) = 0.0921$

Module	Precision	Recall	Specificity	Accuracy
individualmostbought	0.070	0.134	0.892	0.735
mostbought	0.069	0.126	0.894	0.743
randomweight	0.019	0.013	0.894	0.833

Table 5.3: Classification accuracy metrics for three models with all users and the 1000 most common items

By analysing figure 5.5, and table 5.3 it can be seen that the two models, "*individualmostbought*", and "*mostbought*", are similar, compared to the random model. Similarity is not unexpected as they are essentially based on the same algorithm. It can clearly be seen though, that "*individualmostbought*" performs slightly better at all points, with 2.7% better in *nDCG* at the 10% mark. It may be prudent to statistically compute if is significantly better, but as the result is consistent for the full length of the recommendation it at least hints at a significant difference.

Precision, and *Recall*, similarly shows that "*individualmostbought*" is slightly better than "*mostbought*". The random model has a better score in accuracy, this is expected as each position has a high probability to have an item assigned to it. With 6553 randomly sorted permutations of 1000 items it stands to reason that each position will get a correct item once in a while. High *Specificity* for all models shows that most items among the top 100 sorted are relevant to some users.

Chapter 6
Discussion

6.1 Results

It is the opinion of the author that the results of the evaluations are inconclusive. The analysis is not sufficient to determine if any of the models significantly outperforms the other. From the knowledge gained from this project, however, it stands to reason that the results are useful as a part of a larger evaluation. To get complete results the confidence of each model should be tested against each other.

6.2 Evaluation

Evaluating with only historical data will only give a quantitative score, but given other properties of both users, and items, it does not seem impossible to create a user model that can give a qualitative score as well. With the power of a modular evaluation framework it is then easy to create a combined evaluation. The qualitative scoring is, however, strongly application dependant and will take further research to determine how to do properly.

6.3 Framework

The framework and how it executes is without a doubt a success. It is easy to see that the platform created is generic enough to evaluate the two cases used. From those results it can be extrapolated that it should function with only minor overhead work for any similar recommender problem. For recommender problems of other nature, such as systems recommending whole sets of items rather than lists with order importance, additional evaluation metrics is necessary to implement. It is also likely that, when applying the framework to more different recommender problems, adjustments may be needed. These adjustments will, however, only make the framework more generic.

The modularity of the framework has turned out to be useful for evaluation of recommender systems. With the configuration methodology it is easy to add models, and change parameters, of models, as well as implementing specific models. Models are easy to compare by both graph, and single metric values extracted from the graphs. Different graphs can be displayed to provide a good overview of the performance of models. With the implementation of randomized models it is also easy to show a baseline of a least expected performance.

By implementing variance on multiple evaluation tests it should be possible to evaluate the significance of any comparison. It should also be possible to run pairwise comparisons between models through the configuration system and from that compute the confidence of which a model outperforms another.

6.4 Future

As the framework was implemented for experimentation, it does not seem prudent to continue the development of it specifically. However, the structure, and design, of the framework is stable, and with the knowledge of how it is built a vastly improved version should not be a difficult task. It is the hope of the author to implement the techniques learned from this project in the big data framework Expertmaker is using for their live projects.

Chapter 7
Conclusion

While an evaluation framework for recommender systems cannot be completely generic, there are several parts that can be, and others that only need slight modifications depending on application. With a modular framework it is only a matter of implementing suitable modules to accommodate evaluation of a new recommender system. Data can often easily be structured as numeric matrices, which can be used for basic models, and evaluation. With the singular value score, and displayed graphics, a developer can get an idea of the relative power of the models tested. It is only a matter of implementing more metrics to make such a framework applicable with more different recommender systems. *Ranking accuracy* metrics may be useful for certain problems, such as the churn and retailer cases, but for other problems, *classification accuracy* and *prediction accuracy* may be more useful. How to compare metrics must be considered to a greater degree than done in this project, but as the focus of the project was to design and structure a generic platform, this is only a consideration for the future.

Bibliography

[1] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734--749, June 2005.

[2] Amazon. `http://www.amazon.com`, 2014. [Online; accessed 03-December-2014].

[3] Edgar Anderson. The species problem in iris. *Missouri Botanical Garden*, 23:457--509, 1936. `http://biostor.org/reference/11559`.

[4] Joost de Wit. Evaluating recommender systems. Master's thesis, University of Twente, May 2008. TNO Information and Communication Technology.

[5] Expertmaker. `http://www.expertmaker.com`. Online; accessed 24-May-2015.

[6] Filmtipset. `http://nyheter24.se/filmtipset/`. Online; accessed 05-May-2015.

[7] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and T. Riedl John. Evaluating Collaborative Filtering Recommender Systems. *ACM Transactions on Information Systems*, 22(1):5--53, 1 2004.

[8] Will Hill, Larry Stead, Mark Rosenstein, and George Furnas. Recommending and evaluating choices in a virtual community of use. `http://www.sigchi.org/chi95/proceedings/papers/wch_bdy.htm`, 1995. Online; accessed 05-May-2015.

[9] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[10] Yehuda Koren and Robert Bell. *Advances in Collaborative Filtering*, chapter 5. Springer New York Dordrecht Heidelberg London, 2011. Recommender Systems Handbook [17].

[11] Pasquale Lops, Marco de Gemmis, and Giovanni Semeraro. *Content-based Recommender Systems: State of the Art and Trends*, chapter 3. Springer New York Dordrecht Heidelberg London, 2011. Recommender Systems Handbook [17].

[12] matplotlib 1.2.1. `http://matplotlib.org/`. [Online; accessed 09-June-2015].

[13] Netflix. `https://www.netflix.com/`. [Online; accessed 05-May-2015].

[14] Numpy 1.7.1. `http://www.numpy.org/`. [Online; accessed 09-June-2015].

[15] Python 2.7. `https://www.python.org/`. [Online; accessed 09-June-2015].

[16] Pradeep Ravikumar, Ambuj Tewari, and Eunho Yang. On NDCG Consistency of Listwise Ranking Methods. In *14th International Conference on Artificial Intelligence and Statistics*, volume 15, Fort Lauderdale, FL, USA, 2011. AISTATS.

[17] F. Recci, L. Rokach, B Shapira, and P. B. Kantor, editors. *Recommender Systems Handbook*. Springer New York Dordrecht Heidelberg London, 2011.

[18] Paul Resnick, Neophtos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. `http://ccs.mit.edu/papers/CCSWP165.html`, 1994. [Online; accessed 05-May-2015].

[19] scikit-learn 0.13.1, Machine Learning in Python. `http://scikit-learn.org/stable/`, 2015. [Online; accessed 09-June-2015].

[20] Guy Shani and Asela Gunawardana. *Evaluating Recommendation Systems*, chapter 8. Springer New York Dordrecht Heidelberg London, 2011. Recommender Systems Handbook [17].

[21] The New Yorker. `http://www.newyorker.com`, 2014. [Online; accessed 03-December-2014].

[22] Hamed Valizadegan, Rong Jin, Ruofei Zhang, and Jianchang Mao. Learning to rank by optimizing ndcg measure. Master's thesis, Computer Science and Engineering, Michigan State University and Advertising Sciences, Yahoo! Labs, 2010.

[23] Dirk Van den Peol and B. Larivière. Customer attrition analysis for financial services using proportional hazard models. *European Journal of Operational Research*, 157(1):196--217, 2004.

[24] W. Verbeke, K. Dejaeger, D. Martens, and B. Hur. New insights into churn prediction in the telecommunication sector: A profit driven data mining approach. *European Journal of Operational Research,*, 218(1):2011--229, 2012.

[25] Yining Wang, Liwei Wang, Yuanzhi Li, and Di He. A theoretical analysis of ndcg type ranking measures. Master's thesis, Institute for Interdisciplinary Information Sciences, Tsinghua University and School of Electronics Engineering and Computer Science, Peking University, April 2013.

[26] Johan Wensäter. Approaching business problems with machine learning - churn prediction using a three stage decision tree. Master's thesis, Lund Institute of Technology, 2015.

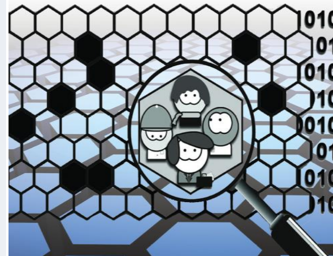
[27] Ke Zhou, HongYuan Zha, Yi Chang, and Gui-Rong Xue. Learning the gain values and discount factors of discounted cumulative gains, March 2013. http://www.cc.gatech.edu/~zha/papers/dcg.pdf.

Appendices

Appendix A
Churn

The following document comes with the churn prediction dataset and includes introduction and guidelines on how to analyse it. It also contains a summary of the different variables provided in the original dataset. The dataset used in this project has been preprocessed according to [26].

Mobile Telephony Churn Prediction Dataset



Introduction

This is a dataset from a Telecom operator with approximately 100.000 customers (active and disconnected) in a tab delimited text file. Traffic type (outgoing, incoming, voice, sms, data), traffic destination (on net, competition), Rate plan, loyalty, traffic behavior etc. are the main attributes of this dataset.

This dataset is divided into two sub-datasets: The first one (churn_dataset1) with the traffic figures for 3 months history (approx 300.000 records) and the second one (churn_dataset2) with the profile variables for each customer (Rate plan, contract renewal date, status, Deactivation_date, value segment etc).

Customer_ID is the key variable for the two sub-datasets

The dependent variable is the Status (active or disconnected) of the customer for the next 4 months from the month that traffic occurred (ACTIVE, CHURN). Status must be combined with the Deactivation month (The month that customer deactivated his contract which is zero for customers that remained active during the specific time period).

It can be used for deriving new variables, descriptive analysis, association analysis and propensity modeling in order to predict potential churners.

Analysis Guidelines

Create new variables that describe traffic trend, (for example: percentage of traffic drop for the last 2 months vs previous 4 months, percentage of traffic (voice or SMS) to competition vs total traffic, Average call duration etc.)

Use multiple propensity modeling techniques such logistic regression and classification trees to estimate the propensity for a customer to be disconnected in the following 4 months.

Use 31/10/1997 as reference date in order to calculate the tenure of the customer (tenure = reference_date – renewal_date)

It is mentioned that you can succeed 5-6 times better targeting (LIFT) than a random sample.

Applications

- Simple Data Management
- Descriptive Statistics
- Association Analysis
- Cohort Analysis
- Cluster Analysis
- Propensity Modeling

CUSTOMERS-DNA

Fields Description (TAB DELIMITED TXT FILES)

Churn Dataset 1	
CUSTOMER_ID	This is the unique key variable for each customer
MONTH	The month that traffic figures refer to
ACTIVITY_DAYS_INC	Number of days per month with incoming traffic (1-30)
ACTIVITY_DAYS_OUT	Number of days per month with outgoing traffic (1-30)
DATA_VOLUME	Total volume from internet sessions (KB)
DISTINCT_CALLERS_INC	Number of distinct callers per month that called the subscriber
DISTINCT_CALLERS_OUT	Number of distinct callers per month that the subscriber called
DURATION_CMP_GSM_OUT	Total minutes from outgoing calls to competition (GSM)
DURATION_CMP_INC	Total minutes from incoming calls from competition (GSM)
DURATION_FIXED_INC	Total minutes from incoming calls from fixed lines
DURATION_FIXED_OUT	Total minutes from outgoing calls to fixed lines
DURATION_INTER_INC	Total incoming minutes from international
DURATION_INTER_OUT	Total outgoing minutes to international
DURATION_ONNET_INC	Total incoming minutes from On net customers
DURATION_ONNET_OUT	Total outgoing minutes On net
DURATION_VAS_OUT	Total minutes from outgoing calls to Value Added Services
NUM_CALLS_CMP_GSM_OUT	Number of outgoing calls to Competition (GSM)
NUM_CALLS_CMP_INC	Number of incoming calls from Competition (GSM)
NUM_CALLS_FIXED_INC	Number of incoming calls from fixed lines
NUM_CALLS_FIXED_OUT	Number of outgoing calls to fixed lines
NUM_CALLS_INTER_INC	Number of incoming calls from international
NUM_CALLS_INTER_OUT	Number of outgoing calls to international
NUM_CALLS_ONNET_INC	Number of incoming calls from On net customers
NUM_CALLS_ONNET_OUT	Number of outgoing calls On net
NUM_MMS_OUT	Number of outgoing MMS events
NUM_SMS_CMP_INC	Number of incoming SMSs from competition (GSM)
NUM_SMS_CMP_OUT	Number of outgoing SMSs to competition (GSM)
NUM_SMS_INTER_INC	Number of incoming SMS events from international
NUM_SMS_INTER_OUT	Number of outgoing SMS events to international
NUM_SMS_ONNET_INC	Number of incoming SMS events from On net customers
NUM_SMS_ONNET_OUT	Number of outgoing SMS events On net
NUM_SMS_VAS_INC	Number of incoming SMSs from Value Added Services
NUM_SMS_VAS_OUT	Number of outgoing SMSs to Value Added Services
NUM_VAS_OUT	Number of outgoing calls to Value Added Services
NUM_EVENTS_WKDAYS_OUT	Number of outgoing events (calls or SMSs) during week days (Monday-Friday)
NUM_EVENTS_WKENDS_OUT	Number of outgoing events (calls or SMSs) during weekends (Sunday-Saturday)

Customers Status Dataset (use this file for creating the Churn Flag)

Use this file for selecting appropriate disconnection months in order to manage latency (see below).

Churn Dataset 2	
CUSTOMER_ID	This is the unique key variable for each customer
VALUE_SEGMENT	Total revenue (monthly fees &outgoing revenue& incoming revenue) per month in € (very high, high, medium, low)
RATE_PLAN	The rate plan for each customer
FIRST_RENEWAL_DATE	The date of the first contract
LAST_RENEWAL_DATE	The date of the last contract's renewal
STATUS	The status of the customer (ACTIVE,CHURN)
DEACTIVATION_MONTH	The month of contract's deactivation
PAYMENT_TYPE	Cash, credit card or bank
CREDIT_SCORE	The credit score of the customer from zero risk (0) to high risk (100)
RATE_PLAN_CHANGES	Number of rate plan changes during the last 6 months
PENALTIES_FOR_NON_PAYMENT	Number of penalties due non payment
AREA	The area that customer belongs

Special Tips & Tricks

Use latency period: It is very critical to use time-lag in your churn prediction models in order to have time to fully synchronize CRM actions (try to predict customers who deactivated their contract in 3rd or 4th month and ignore customers that deactivated their contract during the first 2 months).

You can use the book 'Data Mining Techniques in CRM' (pages 33-36) for more details.

Send us your conclusions to info@customers-dna.com

Appendix B

Configuration File

Listing B.1: Test

```
<?xml version="1.0" ?>
<root>
    <analyse>
        <!-- if empty finds current run in samples and analyses-->
        <!-- if non-empty finds folder in samples and analyses -->
        <!-- requires data_path and eval tags -->
        <!-- uses plot tag -->
    </analyse>

    <git_path> <!-- full path to library -->
        /home/johan/Documents/git/
    </git_path>

    <data_path> <!-- full path to data -->
        /home/johan/Documents/data/
    </data_path>

    <data_sampler name="interval">
        <train>
            <start>
                2013-01-01
            </start>
            <stop>
                2013-11-30
            </stop>
        </train>
        <test>
            <start>
                2013-12-01
            </start>
            <stop>
                2013-12-31
```

```
33          </stop>
34       </test>
35    </data_sampler>
36
37    <hh_sampler name="random">
38       <n> 2000 </n>
39    </hh_sampler>
40    <hh_sampler name="all"/>
41    <hh_sampler name="old">
42       <hash> ## </hash>
43    </hh_sampler>
44    <hh_sampler name="app">
45       <date> MMDD </date>
46    </hh_sampler>
47
48    <offer_sampler name="random">
49       <n> 4000 </n>
50       <default_weight> 0 </default_weight>
51    </offer_sampler>
52    <offer_sampler name="first">
53       <n> 4000 </n>
54       <default_weight> 0 </default_weight>
55    </offer_sampler>
56    <offer_sampler name="all">
57       <default_weight> 0 </default_weight>
58    </offer_sampler>
59    <offer_sampler name="old">
60       <hash> ## </hash>
61       <default_weight> 0 </default_weight>
62    </offer_sampler>
63
64    <impls>
65       <transsort>
66          <data> hh2trans </data>
67          <file> impl_transsort </file>
68          <class> ImplTransSort </class>
69          <plot>
70             <marker> r- </marker>
71          </plot>
72       </transsort>
73       <transsort_rd>
74          <data> hh2trans </data>
75          <file> impl_transsort </file>
76          <class> ImplTransSort </class>
77          <plot>
78             <marker> r- </marker>
79          </plot>
80       </transsort_rd>
81       <mostbought>
82          <data> upc </data>
83          <file> impl_mostbought </file>
84          <class> ImplMostBought </class>
85          <plot>
86             <marker> g- </marker>
87          </plot>
88       </mostbought>
```

```
89        <individualmostbought>
90           <data> hh2trans </data>
91           <file> impl_individualmostbought </file>
92           <class> ImplIndividualMostBought </class>
93           <plot>
94              <marker> g- </marker>
95           </plot>
96        </individualmostbought>
97        <random>
98           <data> nothing </data>
99           <file> impl_random </file>
100          <class> ImplRandom </class>
101          <weight> 0 </weight>
102          <plot>
103             <marker> k- </marker>
104          </plot>
105       </random>
106       <randomweight>
107          <data> nothing </data>
108          <file> impl_randomweight </file>
109          <class> ImplRandomWeight </class>
110       </randomweight>
111       <wlm0612_X init="wlm0612">
112          <data> scores </data>
113          <file> impl_wlm0612 </file>
114          <class> ImplWLM0612 </class>
115          <!-- Used if scoring is wanted, can be hash or xml
                filename -->
116          <hash> config/config.xml </hash>
117          <!-- type of score. one=one value per impl, all=each pos
                has value, none=no precomputed scoring but values in
                impls will still be used. default=none-->
118          <scoring>one</scoring>
119          <!-- List of used impls, required, can have values -->
120          <impls>
121             <transsort>3</transsort>
122             <transsort_rd/>
123             <mostbought/>
124             <individualmostbought/>
125             <random/>
126             <random_weight/>
127          </impls>
128          <!-- possible metrics: min, max, sum, norm2. default max
               -->
129          <metric> max </metric>
130          <plot>
131             <marker> b- </marker>
132          </plot>
133       </wlm0612_X>
134       <!-- Not working
135       <pregenerated>
136          <data> pregen </data>
137          <file> impl_pregenerated </file>
138          <class> Pregenerated </class>
139          <date> MMDD </date>
140       </pregenerated> -->
```

```
141     </impls>
142     <eval>
143         <data> hh2trans </data>
144         <!-- simulation values: intrans , transvalue -->
145         <simulation> intrans </simulation>
146         <!-- possible analysis. lift , cg, dcg, ndcg -->
147         <!-- OBSOLETE
148         <analysis> ndcg </analysis> -->
149     </eval>
150     <eval>
151         <simulation> transtime </simulation>
152         <data> hh2trans_times </data>
153     </eval>
154     <eval>
155         <simulation> live </simulation>
156         <data> live </data>
157         <date> YYYYMMDD </date>
158     <eval>
159         <data> hh2trans </data> <!-- must correspond to current <
             subsim> -->
160         <data> hh2trans_times </data> <!-- must correspond to
             current <subsim> -->
161         <simulation> ndcg </simulation>
162         <!-- use one <subsim> -->
163         <subsim> intrans </subsim>
164         <subsim> transvalue </subsim>
165         <subsim> transtime </subsim>
166         <analysis> lift </analysis> <!-- must use lift for ndcg
             simulation -->
167     </eval>
168     <plot>
169       <!-- Create a figure with contents.
170          if "save_png" in config: png files with name
171          <figure_name_#> will be generated -->
172       <figure_name_#>
173         <include>
174           <impl_names/>
175         </include>
176         <exclude>
177           <impl_names/>
178         </exclude>
179         <name> name </name>
180         <subplot_#>
181           <!-- lift , cg, dcg or ndcg -->
182           <analysis> lift </analysis>
183         </subplot_#>
184       </figure_name_#>
185       <figure_name_#>
186         <name> name </name>
187         <!-- lift , cg, dcg or ndcg -->
188         <analysis> lift </analysis>
189       </figure_name_#>
190       <not/> <!-- add tag to not do any plot -->
191     </plot>
192     <save/> <!-- use if save data -->
193     <show/> <!-- use to show figures -->
```

```
194     <save_png/> <!-- save figures to png -->
195 </root>
```

EXAMENSARBETE Recommender System Validation Platform
STUDENT Johan Ullén
HANDLEDARE Jacek Malec (LTH), Johan Wensäter (Expertmaker)
EXAMINATOR Jonas Skeppstedt (LTH)

Generell utvärderingsplattform för rekommendationssystem

POPULÄRVETENSKAPLIG SAMMANFATTNING **Johan Ullén**

Hur man utvärderar rekommendationssystem finns det mycket litteratur om. Att göra det i praktiken är dock mindre tydligt. Den akademiska metoden bygger ofta på en skillnad mellan in- och test-data. I praktiska sammanhang är det sällan så enkelt.

Rekommendationssystem är automatiska system som genererar förslag utifrån tidigare information eller bestämda regler. En modell byggs upp baserat på information om användare. Därefter används modellen till att värdera föremål. De högst värderade föremålen kommer sen visas för användarna. På t.ex. Amazon får kunder som tittar på produkter förslag om andra produkter som de kan tänkas vara intresserade av.

Målet är att upprätta en plattform där information och algoritmer* ska kunna appliceras. Plattformen ska fungera till olika projekt och kunna avgöra vilka algoritmer som fungerar bäst. Till projektet finns det tillgång till två praktiska problem. Det ena är ett rekommendationssystem för att ge kunder i en snabbköpskedja användbara rabattkuponger. Det andra är ett rekommendationssystem för en mobiloperatör. Operatören vill ha förslag på vilka kunder som behöver kontaktas för att de inte ska säga upp kontrakt.

Svenska IT-företaget Expertmaker utvecklar ett rekommendationssystem för en stor snabbköpskedja i USA. Kunder får tillgång till rabattkuponger genom en mobilapplikation. Det tenderar att finnas hundratals kuponger men det är endast ett fåtal som förväntas vara användbara för varje enskild kund. Från både butikens och kundens perspektiv är det viktigt att kunden inte ska behöva leta efter intressanta kuponger.

Mobiloperatören är ett vanligt exempel på hur den här typen av system kan byggas. Med information om kunders mobilanvändande kan det försöka förutspås vilka kunder som kommer säga upp sina kontrakt. Förslagen som ges till operatören är de kunder som är mest benägna att säga upp kontrakt.

Att hitta vilka delar av en sådan plattform som kan göras utan förändringar för olika miljöer blev centralt för hela projektet. Först och främst behöver informationen vara i ett standardiserat format, detta görs enklast genom att strukturera om den till ett lämpligt format. När informationen väl är strukturerad kan de flesta steg göras oberoende av miljö.

Informationen ska delas upp i indata, för att träna modeller, och testdata, för att generera och utvärdera förslagen. Modeller ska initieras och förslag ska genereras från modellerna. Slutligen måste förslagen utvärderas och även om själva utvärderingen är specifik kan metoden fungera oavsett problemmiljö. Att plattformen fungerar för både snabbköpskedjan och mobiloperatören visar att plattformen fungerar för olika problem.

Nästa steg är att bygga ett liknande system i Expertmakers ramverk. Därefter är det inte långt ifrån att börja använda systemet, initialt för att ta fram lämpliga algoritmer bland kandidater, och senare pågående under ett projekt för att förbättra en algoritm.

**En algoritm är ett sätt att lösa ett problem. Algoritmer i rekommendationssystem utvärderar ofta föremål som ska föreslås genom att jämföra hur lika de är andra föremål. Alternativt kan en algoritm jämföra rekommendationssystemets användare. Då ges användare förslag som liknande användare värderat högt.*