Universiteit Leiden

ICT in Business and the Private Sector

Data mining in business

A case study at Squeezely

Name: Sander Ruijter
Student-no: S1415212

Date: 09/07/2018

1st supervisor: Dr. M. Baratchi
2nd supervisor: Dr. W.J. Kowalczyk

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

In this thesis, we discuss how to build a recommender system for an online marketing company: Squeezely. We start by comparing different filtering approaches and ways of recommending. Then we continue to loss and training functions. We combine these factors into recommendation algorithm and run multiple of them so we can combine them through blending. Afterwards, we talk about some common problems that should be taken into account when building a recommender system. Then we continue to test some different setups to see if using more recommendation algorithms for blending improves our system. We finalize the thesis by implementing the theoretical outcome in real life recommender system at Squeezely.

Keywords: Recommender systems, Blending, PredictionIO

Contents

1 Introduction

It can be hard for small companies to use all the data they have to their advantage. This can have multiple reasons. It could be that the company is oblivious to how valuable the data they possess is. They fail to see that it can be used as an advantage and thus do not invest in getting the most out of it. A more likely case is however that they do know that they can do more with the data but do not have the know-how to do this. This is the case for the majority of the clients of Squeezely.

Squeezely [1] is a company that sells a Data Management Platform and offers to assist their clients with all their online marketing needs. The main focus of Squeezely are small and medium-sized companies. Most of those companies do not have any knowledge of data science themselves, or a budget to set this up. That is why Squeezely is interested in expanding their software with a data mining system. This way the large amounts of data generated from online marketing can be used to get an advantage for their clients and the costs of setting it up can be shared.

Working with data from different small and medium-sized companies means that the data obtained is not extremely large, as the companies do not have the same number of users as, for example, Amazon or Bol.com. Furthermore, there is a high diversity in the number of users, data, and events. So to work with all these datasets in one system it should be able to work at least decent for all these differences in the data. This will also make the recommender systems well suited for small and medium-sized companies to implement themselves or for any other data management platform (or other places where data is just stored) that wants to further advise their clients on what to advertise or give general insights about their data.

The main reason to use recommender systems is to increase revenue. This can be done in multiple different ways. In our case, it is done by advising what items to advertise to which users. This will increase the number of users that come back to visit the site and the increase in traffic will hopefully lead to more sales. Recommendations on the site work in a similar fashion, only instead of getting people on the site, they are used to keep people browsing through the site in the hope they find something they are interested in. This is one thing Netflix invested in to get right and probably one of the reasons that helped them grow [2].

The question that Squeezely had was if it is possible to set up a recommender system that can work for a variety of different data sets and still give accurate predictions? The scientific question that we derive from that is the following: How can a small company with limited resources (time, money, knowledge and computers) set up a data mining environment for different clients and in how far can this compete with scientific approaches?

These questions will be answered by analyzing different possible recommender systems suited to the wishes of Squeezely. We want to find out which one performs better in certain conditions and pick the solution that has the best fit for the company. For this, recommendation approaches are combined through blending.

There are multiple different kinds of data mining that can be researched. In Table 1 an overview is given of the forms we discuss in this thesis and how they compare to each other. For our research, we focus on Recommender Systems because Squeezely wants to advise their clients in what advertisements to show to which person. To do this efficiently we predict what items a person might be interested in. For this thesis, the main focus will thus be on how recommender systems work and how they

can be implemented in a robust way so they work with multiple different datasets while making minimal changes to them.

An overview of how recommender systems work (included blending) is given in Figure 1. It shows the recommender system is made up of a recommendation algorithm with a training function and a loss function. If there are multiple recommendation algorithms the results can be combined through blending to lead to a final result.

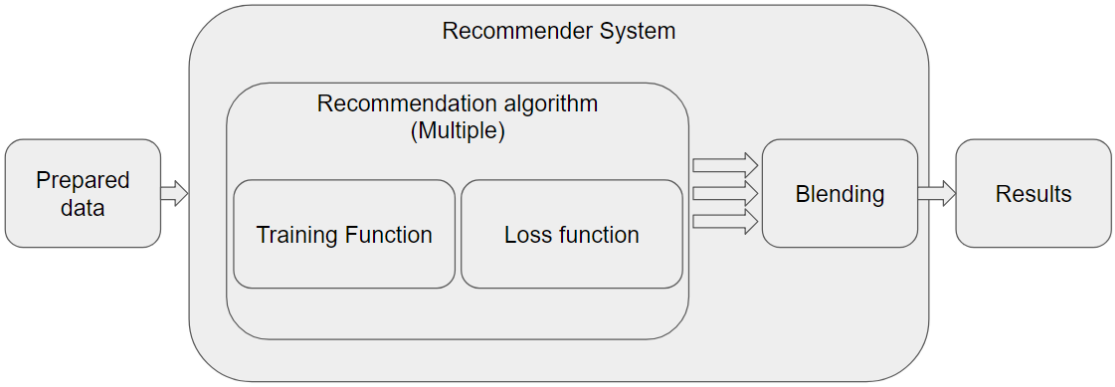


Figure 1: The general structure of a recommender system with blending

There are different kinds of Recommender Systems. In this thesis, we focus mainly on Collaborative Filtering and Content-Based Filtering [3]. In scientific research, the main focus for the last years has been on Collaborative Filtering as it gets higher accuracies in research and practical environments [4]. However, Content-Based Filtering might be easier to understand for humans as it is based on rules mined from the data. The different approaches will be explained further in Chapter 2. Other kinds of recommender systems are demographic, context-aware, and social-based recommender systems. These will be shortly explained to sketch a complete overview of the subject but because the data for this is lacking we shall not implement them.

To train a recommender system there should be some way to test how good they are, for this we will be using a loss function. In Chapter 3 we will discuss three possible loss functions, logarithmic loss, Bayesian personalized ranking loss, and weighted approximate-rank pairwise loss.

Another important part of a recommender system is a function to train the recommender systems. For this purpose, we will discuss alternating least squares and stochastic gradient descent. The discussion about this will take place in Chapter 4.

A way to further improve recommender systems is by blending [5]. With blending multiple recommendation algorithms are used at the same time and the results generated by them are mixed together to increase accuracy. This will be explained further in Chapter 5.

In Chapter 6 we discuss some known problems with recommender systems, and how we kept them in mind during the thesis. These problems include the cold start problem, data sparsity and working with huge data sets with limited resources.

After this, we continue with the methods and results in Chapter 7, and 8 where we explain what we did, why we did it and what we did to prove our ideas are useful.

We continue with a case study at Squeezely in Chapter 9. This to further emphasis on the practical uses of data mining in companies and how this can be implemented in a cost-efficient way.

We finalize the thesis in Chapter 10 with the conclusion and discussion. Here we answer our research question, explain what could be improved when doing similar research, and what would be interesting to look at in future studies.

2 Recommender systems

Recommender systems are used to analyze users, items and the relations between users and items [3]. The information needed to operate a recommender system can be obtained through multiple different means and is different depending on the type of outcome that is wanted. The main types of data used in recommender systems are explicit data, implicit data, personal data and item data.

Explicit data is data that a user explicitly gives. The most common variant of this is giving ratings, for example in Netflix. Another widely used form of explicit data is using likes and dislikes. This is commonly used by Facebook and Youtube. The recommender system can use all this data from a large number of users and analyze it to gain insights into how they might rate other items.

Implicit data is less straightforward than explicit data. Implicit data (with as most common form, clickstream data) are the actions a user takes when on a site that are logged. This could be buying something from a web-shop, but also just looking at an item. When these events are used as is this is a classification problem, however, by interpreting these events as a rating it will become more similar to explicit data. This can be done by assuming that when a user is searching for a specific item he is interested in it and thus it will be a positive rating. If the user bought it or put an item on his wish-list he will be very interested in it. Similarly, an item that has frequently popped up on the screen of the user but is never clicked can be a sign of the user being uninterested in it.

The difference between working with clickstream data and ratings is that ratings expect some kind of number to be given as a prediction whereas for clickstream an upcoming event should be predicted. This makes predicting for clickstream data closer to classification as a set of events leading up to a new event and thus that is what you want to predict. However, by taking into account all possible events for all possible items there are a lot of different outcomes that should be looked at.

The last two kinds of data that are of good use in making recommendations are personal and item data. Both kinds of data give extra information about the user or the product. This can be used in roughly two ways: The first one compares a set of users (or items), finds similar users and recommends what they have seen/liked. The second one analyzes all items a user has a positive view of and tries to finds characteristics that are frequent. This way even new items can be recommended. This can also be used the other way around, by looking at characteristics that link users that liked a specific item and so recommending it to other users.

Although these two ideas sound similar, they are two different ways of working with the data. The first idea is about finding sub-groups of similar people in a large set and the second approach is about finding patterns in a group, thus completely different algorithms should be used for both.

In this research, the choice is made to further research implicit recommender systems without any kind of additional information. We made this choice because of the data collected from Squeezely. Squeezely places a pixel on the site of their clients and can track what pages the user visits and if they bought an item (clickstream data). They do have some small amount of personal data but most of the time this was just partly filled in and thus was insufficient to actually work with.

To make working with our clickstream data easier we slightly modified the data. We gave each kind of event a corresponding rating. For view events this was a one

and for buy events a three. All empty combinations of a user and item are non-defined. If an item has multiple events the highest one is the one that is saved. By modifying the data in this way all prediction approaches used on explicit data can also be used for our clickstream.

The different sorts of recommender systems that will be discussed are content-based filtering and collaborative filtering. Although collaborative filtering is the best fit with the data of Squeezely it is possible that different varieties of data will be made available to work with and thus content-based filtering can also be implemented. Therefore this approach will still be relevant. Discussing both these algorithms also gives a clearer view of the current state of recommendation research and might inspire for new ideas.

For collaborative filtering, we also look into ways to apply other data next to the data available as of now. Doing so will give a better idea of what data can be used to further improve the accuracy of the system created and thus what data might be valuable to save. The other kinds of data that will be discussed can be seen as subsets of personal/item data.

2.1 Content based filtering

One way to use recommender systems is by implementing content-based (CB) filtering. The main difference is that CB filtering systems will try to match attributes of items to (attributes of) users. This differs in the k nearest neighbours (kNN) approach, that will be further discussed in chapter 2.2, in that it uses more data than just which user has seen/bought which item. For example, it may find the relation that women in their thirties like a particular style of dresses while men of the same age are completely uninterested in them.

As stated before in this theses however this data is not available to use for Squeezely. But currently, they are interested in implementing this and expanding their database if this could give them better predictions. Therefore we will still discuss this here.

In content-based filtering there are two main ways of making recommendations [6]. The first uses similarity measures to find similar items and users and use them to get recommendations. This works a lot like kNN but with different data. The second approach is by generating models with rule sets that large amounts of people fall in.

Especially the second one might be interesting for Squeezely. This is because the rules themselves may give some extra, unknown information about users and their habits. By giving these rules in a nice format to their clients Squeezely will be able to better inform them about what might trigger certain people to come back to their site and will move them to buy something.

The biggest problem however now is that these kinds of CB systems do not work in the current environment set up for Squeezely. This environment is made to just read all view/buy events from their database and use them to output a list of recommended items for a group of users. So that is why we will keep content-based filtering as an advice to further look into for another project.

2.2 Collaborative filtering

Another way to use recommender systems is through collaborative filtering. Collaborative filtering based recommender systems try to find similar people (or items). Collaborative filtering uses the items a user has seen, bought or rated to find similar users. So, in short, it compares the behaviour of different users instead of characteristics as with content-based filtering.

During the time collaborative filtering is researched multiple different algorithms have been proposed, all with their own strong and weak points. In this thesis we want to discuss the following algorithms: Singular value decomposition [7][8] (SVD), and k nearest neighbours [9][10] (kNN). Both SVD and kNN are used to predict the likelihood of a user being interested in an item. The given probability will be denoted as x_{ui} for user-item-pair (u,i).

We have chosen SVD and kNN because they are the basic algorithms on which most other algorithms expand. By using these instead of the better, but more complicated variances we keep our recommender system easier to understand for companies with less knowledge on data mining.

Singular value decomposition works by interpreting the data as a matrix X of the size users by items (u×i). X will be decomposed with the following equation:

$$X = USV^T$$

Where U is of the size u×r, S is of the size r×r and V^T of the size r×i, and r is the number of *features* used to decompose X. Using more features will increase the accuracy of the model on the data it is trained on but may lead to over-fitting and thus being unable to predict unknown variables. To use this in collaborative filtering only the known values will be taken into account when decomposing X. Then X can be recomposed through matrix multiplication from U, S and V^T. This will also give an outcome to all values that were previously unknown, these outcomes are the estimations.

However, as there are multiple options to decompose a matrix when there are unknown values training functions can be used to find the optimal SVD. In this case, optimal is the SVD that has the lowest loss. For this, the loss and training functions from Chapter 3 and 4 can be used.

An advantage of SVD is that prediction is very fast once a model has been created. This means that predictions can be made in real time and thus it is very useful in online predictions in webshops and the likes. The problem, however, is that it takes longer for a model to be trained. This because of the many possible variables that need to be calculated for many different cases (all events).

The k nearest neighbours algorithm uses similar users (neighbours) to estimate unknown values for a certain user. In estimating the unknown values only the k most similar other users are used to give an estimation, hence the name k-nearest neighbours. This can easily be explained to people as people who looked at what you have seen also looked at this.

Standard kNN has the advantage that it is fast, easy to implement and understand, robust to noisy data en very effective on large training sets. Disadvantages are that it can be complex to compute, it can cost a lot of memory and can easily

focus on irrelevant connections [10]. Because of these disadvantages, multiple different nearest neighbour techniques are developed. For comparison between these techniques and their strong and weak points, the survey of N. Bhati [10] is very relevant.

Recommender systems can also be improved by taking extra information into account. Ideas to improve collaborative filtering in this way are demographic filtering, context-aware recommendations and social-based recommendation. To implement this demographic data, contextual data and social data are needed respectively. These approaches will be discussed in the next paragraphs.

2.2.1 Demographic filtering

Demographic filtering is based on the idea that similar people have similar preferences [11]. So all data from a certain user is compared to the other users to create a group of *neighbours*. Items that these neighbours are frequently interested in are given as a recommendation for the user. The ideas of demographic filtering are similar to collaborative filtering and kNN but it is less accurate on its own. Because of this, it is rarely used as a complete recommendation system but combining it with collaborative filtering techniques has proven to be useful [12].

The advantages of using demographic data are the lack of the cold start problem (further discussed in chapter 6.1). This is the case because you do not need the user to have some interaction with different items. Once you have the personal data of a specific user you are ready to go. It is also easier to prevent data sparsity (further discussed in chapter 6.2) as the data can come from the creation of an account.

Of course, using demographic data also brings risk with it. Especially now with the new laws on data protection in Europe, it might not be desirable to save extra personal data. Another problem is that you cannot force users to fill in too much personal data when they are creating an account as this will highly de-motivate them to use the services.

2.2.2 Context-aware recommendation

Context-aware recommender systems try to use the context in which the information is given to improve the accuracy of the prediction [13]. Just as with demographic filtering it is not something that can make predictions fully on his own but can improve accuracy when used in combination with standard collaborative filtering approaches [14].

In the paper by Karatzoglou et al [14], they do this by combining the recognition of different contexts with matrix factorization. To do this they create a matrix with three dimensions (Users, Items and Context). This can also be implemented in our system to work with different levels of sparsity. Users are split on the number of items they have seen/bought. On each split, an instance of the program can be trained. When recommendations are made it is first checked what the sparsity levels are and thus which instance of the model should be used. Finding out if this increases the overall accuracy of the system can be an interesting subject for further research.

2.2.3 Social-based recommendation

The last form of recommender system that will be discussed to improve collaborative filtering is social-based recommendation. Social-based recommender systems use data from social networks to improve the accuracy [15]. The idea is that people will better trust recommendations of people they know or feel connected to.

The usage of such a system looks very similar to the nearest neighbour approach. However, instead of calculating the neighbourhood in some way they get it from the social network [16]. As long as such a network is available and every user has sufficient neighbours that can be used for estimation it is easy to combine with the kNN approach described earlier. In a simple version, the similarity rating of two users can be boosted if they are in the same social network. More complicated combinations of the two are also possible and lead to better accuracy in most cases [16].

The main reason that we do not look into this further is that Squeezely does not have access to large social networks and thus is unable to efficiently implement this. Furthermore, Squeezely has some cooperation with Facebook, that can do this on a way larger scale than ever will be possible for Squeezely.

3 Loss Functions

To train a model-based collaborative filtering approach an essential part is the loss function. A loss function quantifies how good a prediction model is compared to the expected outcome [17]. In this thesis Logarithmic Loss (LogLoss), Bayesian Personalized Ranking [18] (BPR) loss, Weighted Approximate-Rank Pairwise [19] (WARP) loss, and Mean Absolute Error/Root Mean Squared Error (MAE/RMSE) are discussed. BPR and WARP loss are more specialized for predicting items for users in collaborative filtering. MAE and RMSE are both common error measures when evaluating models.

To get optimal results the loss functions should also be minimized for a test set and not just for the training set. This can be done by ignoring a subset of the data during training so it can later be used as test set. By optimizing the loss function for the test set over-fitting is countered.

3.1 Logarithmic loss

Logarithmic loss (LogLoss) is a basic loss function that can be used for a multitude of problems [17]. The idea behind LogLoss is that the more your estimation differs from the actual outcome, the worse it is. So instead of just taking the error, you take the log of the error, to extra penalize high errors.

The LogLoss is obtained through the following function:

$$LogLoss(p, a) = \begin{cases} -log(p), & \text{if } a = 1 \\ -log(1-p), & \text{if } a = 0 \end{cases}$$

Here p is the predicted value and a the actual value. This only works if the actual values are either 1 or 0, so if the actual value is not 0 or 1 and thus the predicted value should not be 0 or 1 we divided the predicted value by the actual value. This way the logarithmic loss will still be useful.

To make this work a clip function should be added to the predicted value. This as the log of zero is undefined and will thus give an error. The clip function will change possible undefined outcomes to a low value specified by the user so it will still be defined.

3.2 Bayesian personalized ranking loss

Bayesian personalized ranking (BPR) is a way to recommend items using only explicit feedback. It was developed by Rendle et al [18] to optimize ranking of items for (a group of) users. For this, they presented the generic optimization criteria BPR-Opt. BPR-Opt is the corresponding learning method that trains other algorithms by optimizing the BPR loss. The final formula derived for BPR loss is the following:

$$BPR - Opt = \sum_{(u,i,j)\epsilon D_S} ln\ \sigma(\hat{x}_{(u,i,j)}) - \lambda_\Theta ||\Theta||^2$$

Here $\sum_{(u,i,j)\epsilon D_S}$ means for every combination of a user u, positive item i and negative item j in the trainings data D_S, $\sigma(x)$ is the logistic Sigmoid function $\frac{1}{1+e^{-x}}$, $\hat{x}_{(u,i,j)}$ is the probability user u prefers item i over item j calculated by $\hat{x}_{(u,i)} - \hat{x}_{(u,j)}$,

λ_Θ are the mode-specific regularization parameters, and Θ is the parameter vector of an arbitrary mode class.

So this means that BPR loss looks at positive items, compares it to a negative item and increases the loss if the negative item has a higher estimation than the positive item. This leads to the positive items being at the top of the estimated items. Lastly, a penalty is given if the parameters of the model become too high, to prevent over-fitting.

The advantages of BPR are the optimization of the top of the list of recommendations. When giving recommendations to a user the top items are the most interesting, as those are the one that should be recommended. Secondly, BPR also has a form of preventing over-fitting. This is also an important point when building recommender systems as you want the system to give accurate estimations on all future cases and not just for the training data.

Of course, BPR also has its disadvantages: it works best when using positive and negative feedback. This is not something available for Squeezely. However, to still use BPR combinations that have no feedback can be compared to the positive events. It is even possible for Squeezely to add negative events in the future to improve the accuracy obtained through using BPR loss.

3.3 Weighted approximate-rank pairwise loss

Weighted approximate-rank pairwise (WARP) is another loss function that has been developed for implicit data, and specific for huge datasets. It is developed by Weston et al [19]. WARP loss focuses on getting the top estimations right in your program, as those are the ones that will be used most. The loss formula the use for this is the sum of all triplets with a user, a positive example and a higher rated negative example:

$$err(f(x), y, \bar{y}) = L(rank_y^1(f(x)))|1 - f_y(x) + f_{\bar{y}}(x)|_+$$

Here $(f(x), y, \bar{y})$ is a triplet of the parameters for a user x: $f(x)$, a positive item y, and a negative item $\bar{y}$. $f_i(x)$ gives a prediction score for item i, $\bar{y}$ is chosen in such a way that$f_{\bar{y}}(x) + 1 > f_y(x)$. $rank_y^1(f(x))$ is the margin penalized rank of y and $L(\cdot)$ converts this rank to a loss function. The rank is penalized so it will become continuous and thus be approximated. $L(\cdot)$ converts in such a way that the higher an item is in the ranking the worse an error is for that item. this leads to the top being more precise than the bottom. Lastly the part $|1 - f_y(x) + f_{\bar{y}}(x)|_+$ adjusts the error the closer the estimations of y and $\bar{y}$ are to each other.

WARP loss, just like BPR, works best with combinations of positive and negative feedback. To still work with WARP loss the same solution can thus be given, using unrated events as negative events until negative events are added in the future.

The advantages of using WARP loss are its focus on the top recommendations, just as with LogLoss and BPR loss. It is also better suited to work with data too large to load in memory [19]. This is the case because WARP loss only needs one specific triplet at a time. Although it will be faster to do when it everything can be fitted into memory, calculating this loss can easily be delegated to multiple cores or processors.

3.4 MAE/RMSE

The final loss functions that we will discuss are the Mean Absolute Error and the Root Mean Squared Error. Both are widely used when testing models and have some cases in which one will outperform the other.

The first one we will discuss in this subsection is the Mean Absolute Error (MAE). Just as the name suggests it takes the average of all non-negative errors. Mathematically it uses the following formula for n recommendations.

$$MAE = \frac{1}{n}\sum_{t=1}^{n}|e_t - a_t|$$

Here e_t is the estimated value and a_t the actual value of a user-item pair t. The main improvement over the actual average is that negative and positive errors do not cancel each other out.

The Root Mean Squared Error (RMSE) is another, closely related loss function. It was the loss function used during the Netflix prize contest [2]. The RMSE is calculated by taking the average over the square of the errors and taking the square root of this average. This leads to the following formula for n recommendations:

$$RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(e_t - a_t)t^2}$$

Here e_t is the estimated value and a_t the actual value of a user-item pair t. The advantage of the RMSE over MAE is that it increases the loss when estimations have a higher error. This helps the training functions to focus more on their weak points, the ones where there is the most room to improve.

RMSE got an increase in popularity because of the Netflix prize [2] and is nowadays widely used for recommendation purposes. However, there are still researchers that prefer to use MAE over RMSE as MAE is less prone to outliers [20]. But when the error follows a normal distribution RMSE is, in general, a better metric to use [21]. Furthermore, RMSE is also preferred in some cases where it is undesirable to use absolute values. In our case, there is a slight preference for the MAE as there may be outliers in the data and it is no problem here to work with absolute values.

4 Training Functions

The last part important for the recommender system is the training itself. In this chapter we are going to discuss two different ways of training recommender systems: alternating least squares (ALS) [22], and stochastic gradient descent [23] (SGD). For both training methods, there are different variations, here we will only discuss standard ALS and SGD.

4.1 Alternating least squares

The idea of ALS comes from astronomy and geodesy, during the 18th century [24]. It is a method to approximate an optimal outcome for functions with multiple variables. It does so in four steps:

1. Freeze all but one variable
2. Solve the resulting ordinary least squares problem
3. Freeze the current variable and thaw another variable
4. Repeat steps two and three until convergence

So by freezing all but one variable, the optimization problem becomes much easier, it becomes a linear regression problem. By doing this multiple times for the different variables a (local) minimum can be found. This is the case because at every step the loss value either decreases or stays the same, it will never increase. Therefore we say that convergence is reached after the loss value stays the same after a certain amount runs through the algorithm.

Once a local minimum is found the loss value will not further decrease, as no possible solution can be found from that point forward. Changes are high, however, that this is not the best solution but the complexity and needed time decrease drastically.

4.2 Stochastic gradient descend

SGD is a simplified version of the original gradient descend, that has been proposed in multiple situations to optimize variables. It does so in the following way:

1. Find a direction that decreases the loss function the most
2. Take a small step in that direction
3. Repeat steps one and two until convergence

Stochastic gradient descent differs from gradient descend in that it does not calculate the exact gradient but makes an estimation based on a single example. Estimating the gradient in such a way is less costly in computational power and time [23]. Furthermore, the Robbins-Siegmund theorem [25] gives guidelines to let stochastic gradient descent lead to convergence most of the time. The main idea it uses to achieve this is decreasing the size of the steps it takes and thus ending in a local minimum. However, this progress can be slowed because of noise in the

data, as it will try to compare to these noisy data points that will not lead to an improvement.

When choosing between these two algorithms there are a few points to keep in mind. SGD is, in general, easier to use and has a shorter training time [22]. However, ALS can optimally use parallelization as multiple variables can be estimated at the same time [26] what can lead to huge speedups. As a server with 4 cores will be available training time might be shorter for ALS on this. However, after some testing on the server, this was not by a large enough degree.

Collaborative Filtering algorithms				
	Accuracy	Training time	Prediction time	Understandable
SVD	Low-High	Long	Short	Hard
kNN	Medium-High	Medium	Medium	Easy
Loss functions				
	Accuracy	Training time	Prediction time	Understandable
Log	Low	Low	-	Easy
BPR	Medium-High	Medium	-	Medium
WARP	Medium-High	Medium	-	Medium
MAE	High	Low	-	Easy
RMSE	High	Low	-	Easy
Training functions				
	Accuracy	Training time	Prediction time	Understandable
ALS	Medium-High	Medium-High	-	Easy
SGD	Low-Medium	Low	-	Easy

Table 1: Comparing different recommendation approaches on their accuracy, training time, prediction time and how understandable they are

5 Blending

As stated in Jahrer, Töscher and Legenstein's paper on blending for the Netflix dataset [5] blending can drastically increase the accuracy of the predictions made. It does so by running multiple recommendation algorithms and using a second algorithm to combine all these estimations into one final estimation.

From the same paper, we got the inspiration for selecting the following forms of blending ((binned) linear regression, neural network, and bagged gradient boosted decision tree). They used these blending approaches on the following CF algorithms: k-nearest neighbors (Item-Item and User-User), matrix factorization (SVD), SVD extended, asymmetric factor model, restricted Boltzmann machine, global effects and residual training. In total, they blended 18 different CF algorithms by using different parameters on the aforementioned algorithms. Blending increased the Root mean squared error by 1,4-6,7% depending on the blending method used.

In this thesis, we compare four kinds of blending, logistic regression, binned logistic regression, neural networks, and bagged gradient boosted decision trees. A short overview of all four blending methods will be given in Table 2, including their advantages and disadvantages.

5.1 Logistic regression

The first easiest form of blending is logistic regression (LR). Logistic regression takes the recommendations and translates them into a probability from 0 to 1. The higher the probability is the more interesting it is.

Although blending by logistic regression is not particularly special it does improve the average accuracy in comparison to just using one model. Therefore we will use this as a benchmark to test other blending algorithms against.

5.2 Binned logistic regression

A small step more complicated is binned logistic regression [5] (BLR). The idea of bagging is that through the splitting of the data a better average prediction will be made as only a part of the bags will be influenced by outliers and noise. Bagging [27] can be used for every supervised learning project to increase accuracy. It is used by training N copies of the model on slightly different training sets. The samples that are not inside the training set are then used for validation. By averaging on all copies an out-of-bag estimation is made for the whole data set. This, however, does increase the training time as multiple copies of the model are trained.

In binned logistic regression the training set is divided into B disjoint subsets. For each subset $b = 1,...,$ $\mathbf{B}$ a blending weight $\mathbf{w}_b$ is denoted. The training set can be split on support (number of votes by a user), time (the day of the data point, this makes it easier to do time-dependent blending) and/or frequency (the number of ratings of a user on a day t so it can adjust weights if a person is more active on certain days). The prediction is given by $\hat{r}_{ui} = \mathbf{x}^T \mathbf{w}_b$.

5.3 Neural network

Another widely-used and fairly popular method are neural networks [28] (NN). Small neural networks can be trained efficiently on the large data sets available in this

thesis. The output of the network is a sigmoid activation function in the range [-1, 1]. A simple output transformation can be used to change this any preferred outcome. For the complete mathematical explanation behind NN, the book by R.J. Schalkoff can be consulted or another on-line available source.

The idea from NN comes from the human brain. It is made up of multiple small *cells* called neurons. These neurons receive multiple inputs and give back a single output depending on how they were trained. By combining multiple layers of multiple neurons they can be trained for complex tasks, just like in a brain.

The main problem, however, is that these neurons do need to be trained, which can cost quite some time and a huge amount of data is needed. In the case of the data in this thesis, it is possible to train a smaller NN. This will probably suffice as larger NN will have a high change to over-fit for our purpose.

An advantage of neural networks is that the term itself is very popular in business. As there is a lot of populist articles that the term as the answer to everything that is the idea that people get. For marketing purposes, it will thus become a bit easier to sell the platform of Squeezely as it will be seen as more *high-tech* than other platforms. However, this does not add any scientific value to neural networks.

5.4 Bagged gradient boosted decision tree

Another interesting approach to blending is the use of decision trees. However, using a single tree has a fairly low accuracy in general. To overcome this more trees can be used. The two ways used for bagged gradient boosted decision trees [29] (BGBDT) are Bagging (Just as with bagged linear regression) and Gradient Boosting [30].

Gradient Boosting uses multiple learners of the same type (in this case using the same bag) in a chain to all learn a fraction of the desired function Ω, controlled by the learning rate η. This works as follows: the first model Ω_1 is trained on the unmodified targets $\mathbf{y}_1 = \mathbf{y}$ of the dataset. The second model trains on $\mathbf{y}_2 = \mathbf{y}$ - $\eta\mathbf{y}_1$. So targets in each next $\mathbf{y}_i = \mathbf{y} - \sum_{j=1}^{i=1} \eta\mathbf{y}_j$. The final prediction model is $\Omega(\mathbf{x}) = \sum_{i=1}^{M} \eta\Omega(\mathbf{x})$, where M is the length of the chain.

By combining bagging and gradient boosting a large number of trees are trained (the amount of bags times the number of gradient boosts). This will lead to an increased time needed for training. Furthermore, the data set also has to be sufficiently large as every tree uses a slightly different part of the data. Thus this part still has to be large enough that a tree can be trained on it, and still be relevant for the whole data set.

	Accuracy	Training time	Prediction time	Understandable
LR	Low	Low	Low	Easy
BLR	Medium	Medium	Low	Easy
NN	High	High	Low	Medium
BGBDT	High	Very high	Low	Hard

Table 2: Comparing different blending approaches on their accuracy, training time, prediction time and how understandable they are

6 Problems with recommender systems

6.1 Cold start problem

A problem for recommender systems, in general, is giving recommendations for new users or items [31]. This is called the cold start problem. The problem occurs because the amount of data available is too low to make an accurate prediction. On this, there have been multiple papers that further explain this ([32][33]). In our case, we want to predict what items a user might like and thus what we want to advertise to a specific user.

When there is a very small amount of data from a user they did not have much contact with the web-shop (or any other environment). Therefore, it might be more worthwhile to advertise to get them back to the web-shop with general advertisements instead of trying to recommend with the minimal data available. Following this train of thought, we choose to omit further looking into the cold start problem during this thesis.

6.2 Data sparsity

Another thing to keep in mind when working with recommender systems is data sparsity, especially when working with collaborative filtering approaches as it can drastically decrease the accuracy of the approach [34]. Sparsity is the amount of data missing in a data set. This can influence the predictions by making a few ratings/events way more important than others as they are for a less rated item.

A nice example of this was given by Herlocker et al. as early as in 2002 [35]. It concludes to that if only one person has viewed an item, and instantly bought it, then for every other user the one buy event will be the only guideline and thus the rating given will be very high. This leads to biases in the prediction. Although this is quite an extreme case and most algorithms do have some ways to nuance this it is still something that has to be kept in mind.

In the datasets in this thesis, this is something very relevant, it is as good as impossible for a user to have seen/bought every item in the data set, most users haven't even rated 1% of the items. So there will be sparsity. The levels of sparsity will also drastically differ from user to user and client to client. There are multiple techniques that can help with sparsity in data. The ones that will be discussed in this thesis are ignoring data, data enrichment and hybrid systems.

Ignoring data is not a way of reducing the sparsity in data but it does help with countering the negative effects of sparsity [36]. The idea is to not rate items that have too few ratings/events. This will completely ignore cases as described earlier in this chapter. Although this is quite a drastic approach it is efficient in only keeping accurate predictions at the top of the list. And as this can be done after the predictions have been made it does not influence the other parts of the prediction process.

Data enrichment, on the other hand, works by adding extra values to the data or improving the quality in another way. The most workable idea is adding extra data about the users or items to your data set, this makes it possible to use additional approaches. By doing so a hybrid system can be made combining content-based filtering and collaborative filtering. As described in chapter 2 this is a future plan

to do at Squeezely, so the platform that is being built in this thesis should allow for extra data to be added.

6.3 Working with huge data on limited resources

The last possible issue is working with huge amounts of data while working with minimal resources. In a lot of cases, this will lead to performance bottlenecks on certain stages in the recommendation process. These bottlenecks can be on either time/computational power or on memory. For developing a platform these concerns should be taken into account when selecting an approach. For most recommender system it is doable if the training takes more time, as long as after the training recommendations can be made extremely quick (recommendations in webshops). In other cases neither time nor memory will be a constraint, an example of this is sending advertisements for a large company through the mail. It is rarely a problem when an advertisement mail arrives an hour later or earlier and a large company should have a sufficiently powerful computer that can work around memory constraints.

In the business case for Squeezely, the system is running on a server from Amazon Web Services (AWS). It is easy to get more memory or computing power on AWS as they are pay per use. Nevertheless, because it is pay per use you do not want the system to be needlessly complex and taking up way more resources while gaining minimal improvement in results. Therefore it is important to take this into consideration when building the recommender system.

The time constraint is much more lenient for this project. At the start, these recommender systems will only be used to build advertisements and send out e-mails to specific users. Therefore it would not be a problem for the system to start at the end of the day and be ready the next morning. For recommending purposes this is an extremely long time as a lot of systems want to be ready in real time.

7 Methods

7.1 Datasets

For this thesis, the main focus is on how to make a recommendation system for Squeezely. However, to be able to compare this with other research another dataset be tested upon as the data of Squeezely is not publicly available.

The dataset we have selected to use is the RecSys 2015 Yoochoose dataset [37]. This data set is made up of two files, one with all view events, and one with all buy events. These are the same kind of events as the data from Squeezely and thus the same loss functions and accuracy measures can be used. The main differences are that the RecSys dataset has a larger number of users and items by about a thousand times and has a hundred times more events in it. This leads to a sparsity of 99.995% in the RecSys dataset against a sparsity of 99.956% in Squeezely's dataset.

The differences in size mean that although the results should be more or less the same, it will take longer to get recommendations. As this is the case for all research that uses this dataset it is not a problem in comparing the results. Furthermore, it is also useful for Squeezely to see how the recommender systems work with larger datasets, and thus if it can be used in its current form for even larger clients.

After trying to work with the dataset the first few times, it turned out that it was too large for the server to handle. Because of this, we worked with a subset of the RecSys data set. This subset includes the first 30000 users and all items and events associated with these users. This lead to a dataset with 10769 items and 82208 events in it. The total sparsity was now 99.975% which makes it more similar to the dataset from Squeezely.

Working with the different levels of sparsity in the two data sets may still lead to different results. Here the part of context-aware systems might be used in further studies to make the differences as small as possible. By selecting recommendation algorithms and blending approaches better suited for high levels of sparsity the estimations made for the RecSys Yoochoose dataset should be able to be on par with the recommendation for Squeezely.

7.2 Picking the algorithm

After another dataset was selected that is freely available and allows us to compare our results with other solutions it is time to build our recommendation system out of all the blocks we talked about until now. The python version of the program is available through **Github, Leiden University, me?** The PredictionIO files are only available for Squeezely.

As we are training a lot of different recommendation algorithms on a large amount of data this will take most of our resources (time and memory). For each recommendation either the model should be saved or all possible outcomes to make quick recommendations. This makes it thus that in a worst case scenario there should be $N \times U \times I$ doubles saved, where N is the number of recommendation algorithms used, U the number of users and I the number of items.

As discussed earlier, we use a standard collaborative filtering approach for our recommender system because our data is most suited for it. For the loss function, we pick the logarithmic loss for one part of the recommendations and WARP-loss for the other part. WARP loss is used to increase the top of the predictions as much

as possible. We do this because only the top of the recommendations is used for blending. Logarithmic loss is used to punish larger errors more than small errors so our predictions will be closer to the true answers in general.

The training function we use for this is stochastic gradient descent. We have chosen SGD as it is fairly easy to use and has a fast training time on all set-ups. As for the recommendation method, we chose singular value decomposition. This choice was made because of the library (LightFM [38]) found in python that could implement our earlier made decisions.

An important note here is that a preferred solution is one that uses a set of similar algorithms for making the first round of predictions. This because it keeps the system clearer and thus better to understand. For Squeezely it would be a big plus for the system to be less complex as there is no one who is very knowledgeable about data mining. So by using one kind of recommendation algorithm, it is easier to learn how the algorithm works and the program can be kept up to date by one of the current employees with less effort. Furthermore, it is proven that even blending on multiple slightly different model of the same algorithm will also improve the accuracy [39].

Because of the reasons mentioned in the previous paragraph we decided to work with the python module LightFM [38]. This library can train an SVD based model on either a logarithmic loss function, a BPR loss function or a WARP loss function. It does so by using adagrad [40] or adadelta [41], both variances on stochastic gradient descent. This thus has a good fit with the decisions made previously. Furthermore, it is also possible to add the BPR-loss, but as there are no negative examples that improve the working of BPR based approaches this is not done.

8 Results

Here we analyze the results we got from our recommender system. We do so by comparing the MAE, RMSE and logarithmic loss for the different amount of recommendations that are blended. We also analyzed how much time it took to train for blending different numbers of recommendation algorithms. All data shown are the results obtained through something similar to 5-fold cross-validation.

For the recommendation algorithms used to blend we took the combinations of SVD as the base algorithm, SGD as the training function and either logarithmic loss or WARP loss as the loss functions. This to keep it in line with the decisions made in Chapter 7.

8.1 RecSys dataset

First, we had to find the most suitable amount of algorithms to use for the RecSys dataset. For this, we tested multiple numbers of algorithms, with the results in Figure 2 and Figure 3. The results were for 70 users, with multiple levels of sparsity in the training and test sets. The results are for all different kinds of blending combined to give a general overview.

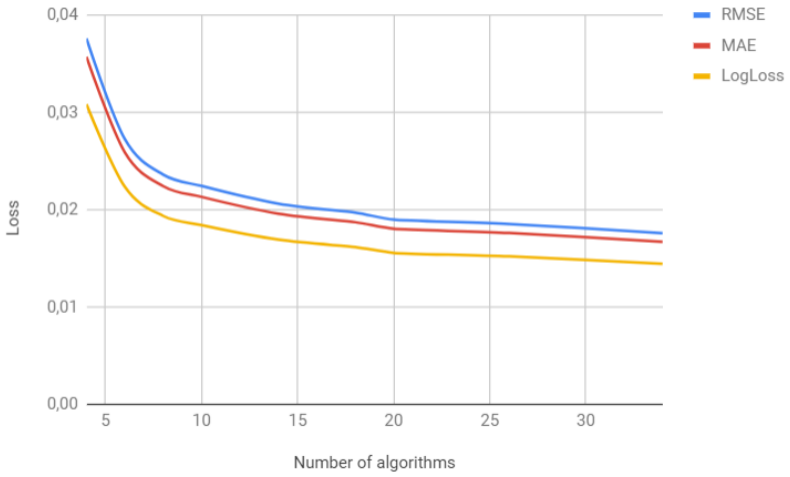


Figure 2: Comparing RMSE, MAE and LogLoss for 4 - 36 algorithms, LogLoss is a factor 40 smaller to fit on the scale

Figure 2 shows how good the predictions are. The figure clearly shows that more recommendations lead to better results. All results do show the same form in their line, this is expected when the errors are so low as there is some dependency between them. The reason that the errors are so low might be because of the sparsity. This data is extremely sparse, thus most estimations should be 0, this leads to the estimation being mostly zero too, as are the actual results.

In Figure 3 the time it took to train the system is plotted against the number of different recommendation algorithms used. The time it takes to train looks to be linear to the number of algorithms used. This is the case because every recommender algorithm takes the same time to train. The time it takes to blend the algorithms is not taken into account here, as this never exceeded 0,3 seconds.

Although the optimal number of algorithms for accuracy turned out to be as high as possible, for comparing the different levels of sparsity we wanted to test

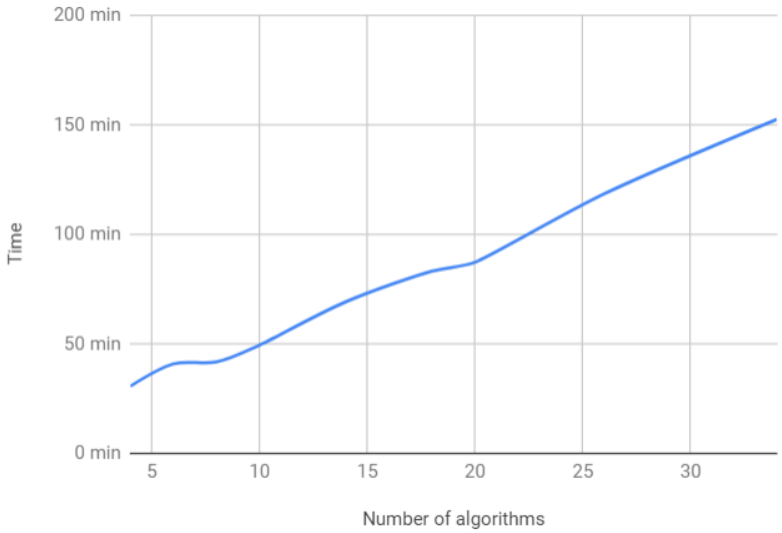


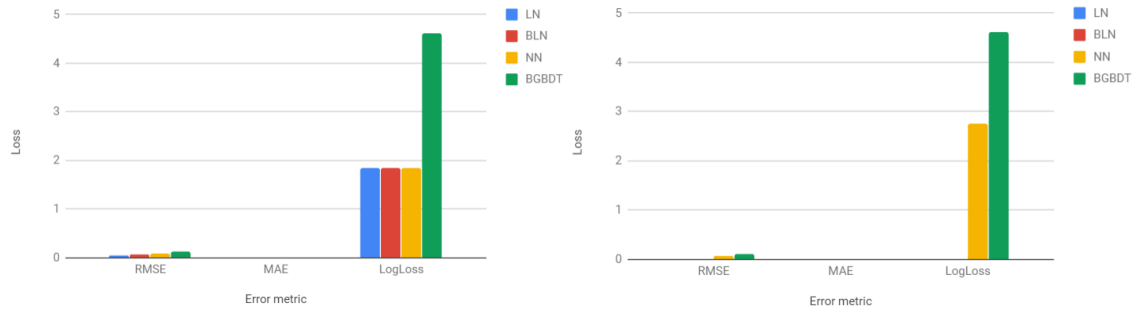
Figure 3: Comparing time to train the system for different algorithms

with a small number of algorithms (6 algorithms) and a large number of algorithms (36 algorithms). We did this for a group of users with a sparsity <0.01%, between 0.01% and 0.02%, between 0.02% and 0.05%, between 0.05% and 0.1%, between 0.1% and 0.2%, between 0.2%, and 0.5%and more than 0.5%.

The results of these experiments can be seen in Figure 4-10. Each figure is made up of two graphs, graph a is made by using 6 recommendation algorithms and graph b by using 36. AS described earlier the recommendation algorithms use a combination of SVD and SGD. As a loss function, half of them use logarithmic loss and the other half use WARP loss. The algorithms are implemented through LightFM with a different number of features and epochs to train to get somewhat different results. When six algorithms are used 9-10 features and 30-31 epochs are used, in the combinations 9 features with 30 epochs, and 10 features with 30 and 31 epochs. For 36 algorithms there were batches of 9, 10 and 11 features with 28-33 epochs.

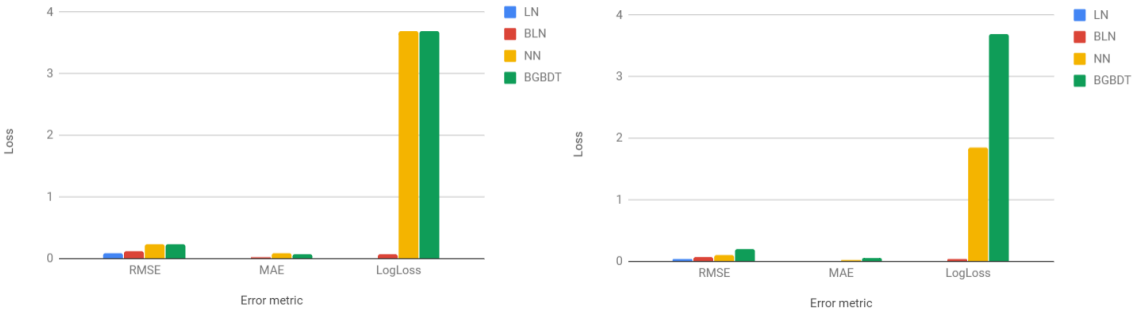
The graphs (Figure 4-10) are build by taking the average of 10 users on the RMSE, MAE and LogLoss for the different blending approaches. Graph a is made by blending 6 recommendation algorithms and graph b by blending 36 algorithms. We chose the levels of sparsity because any larger amounts of sparsity would lead to too small testing groups to get results. Furthermore, in the lowest group, with an average of 0.009% of items seen/bought, only one item has been seen/bought, so any lower would not be possible.

As seen in the Figure 4-10 the loss decreases in nearly all when using more blending algorithms. This is in line with the results described in the first paragraph of this chapter, where more recommendation algorithms lead to better results. The only cases that it leads to worse results is in the log loss of NN or BGBDT. This might be the case because log loss is very sensitive to wrong estimations. If an estimation is given to recommend an item while it should not be recommended the eventual loss value will drastically increase (by $log(0.000000000000001) \approx 34.5$ / the number of other users).



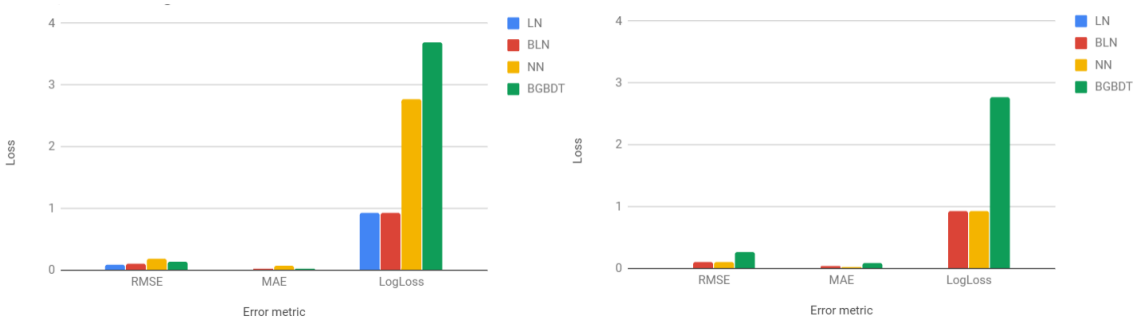
(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 4: Maximum of 0.01% of data filled in per user, average of 0.009%, 44% of the users



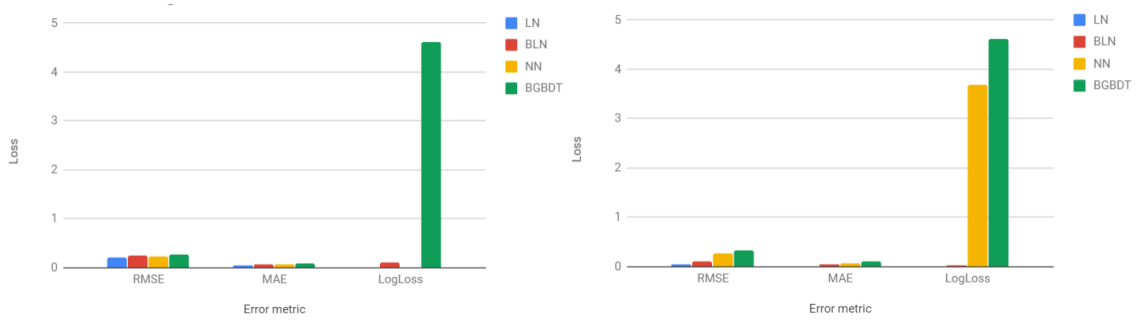
(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 5: Between 0.01% and 0.02% of data filled in per user, average of 0.019%, 30% of the users



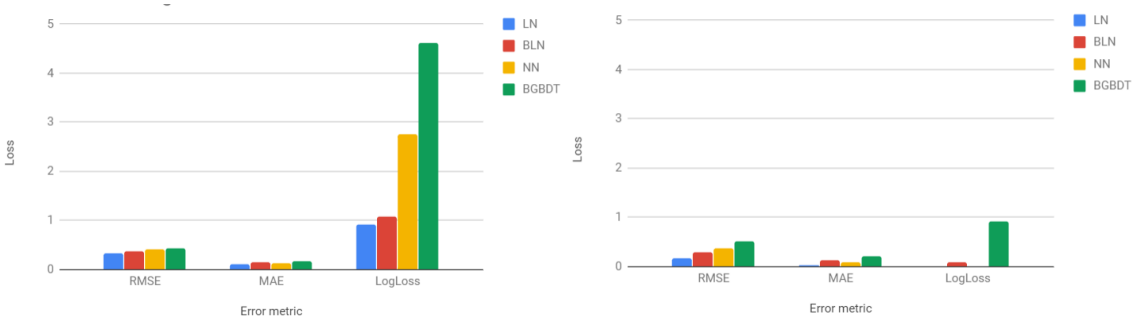
(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 6: Between 0.02% and 0.05% of data filled in per user, average of 0.035%, 21% of the users



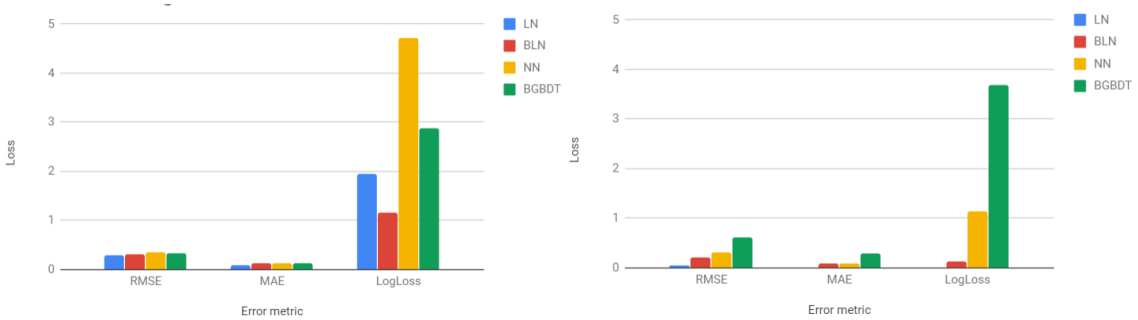
(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 7: Between 0.05% and 0.1% of data filled in per user, average of 0.066%, 3.9% of the users



(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 8: Between 0.1% and 0.2% of data filled in per user, average of 0.13%, 0.74% of the users



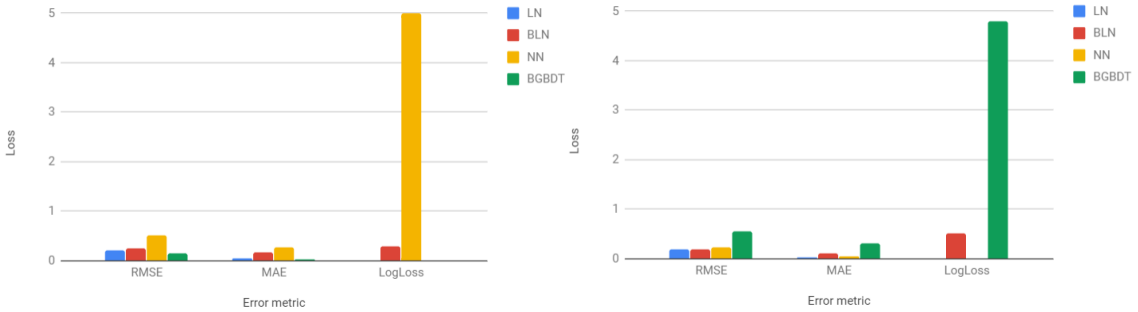
(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 9: Between 0.2% and 0.5% of data filled in per user, average of 0.26%, 0.05% of the users

8.2 Squeezely's dataset

For Squeezely's dataset, we once again started with finding the most suitable amount of algorithms to use. For this, we did the same tests, with the results in Figure 11. An explanation for this can be that using too many algorithms can lead to overfitting and thus worse results.

Figure 11 looks different than Figure 2 at first sight, but when taking a closer look and by smoothing out the bumps the same general trend can be found. For RMSE and MAE more algorithms once again have the best results. For LogLoss



(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 10: More than 0.5% of data filled in per user, average of 0.55%, 0.003% of the users

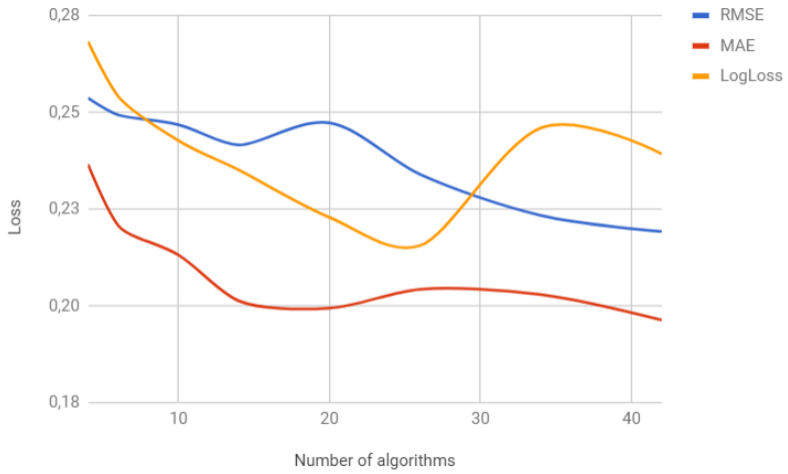


Figure 11: Comparing RMSE, MAE and LogLoss for 4 - 36 algorithms, LogLoss is a factor 20 smaller to fit on the scale

there is some abnormality between 25 and 35 that suddenly makes the LogLoss increase when a decrease is expected. This is most likely because of the users picked for the test. This is a subgroup with only 70 users and because of this small size, it can be unrepresentative.

In Figure 12 we once again show the time it took to train all models against the number of algorithms used. This is in line with the results of the RecSys data set so this is as expected.

The results of the experiments with a small and large number of recommendation algorithms can be seen in Figure 13-19. The algorithms used in these experiments are identical to the ones described in Chapter 8.2. Each figure is also once again made up of two graphs, graph a is made by using 6 recommendation algorithms and graph b by using 36. As seen in the Figure 13-19 the RMSE and MAE decreases in nearly all cases when more recommendation algorithms are used except when using Bagged Gradient Boosted Decision Trees. The most likely reason is that BGBDT needs a much larger amount of training data to blend 36 results than for 6 results, which leads to it under-performing when more recommendation algorithms are used.

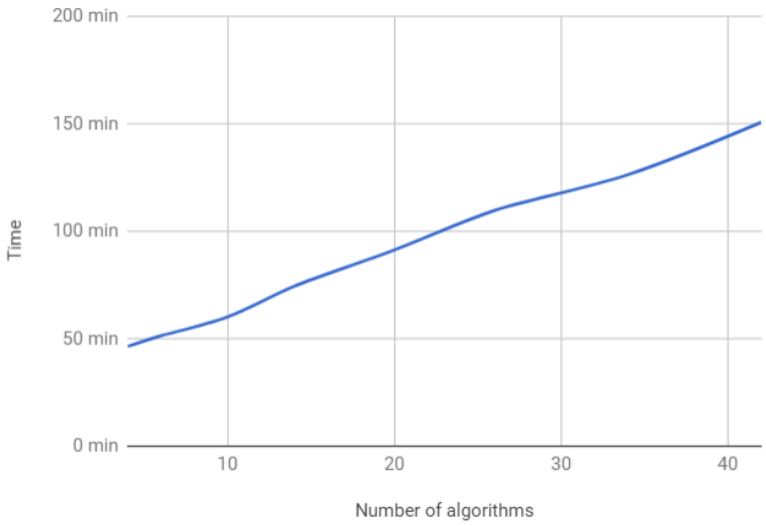
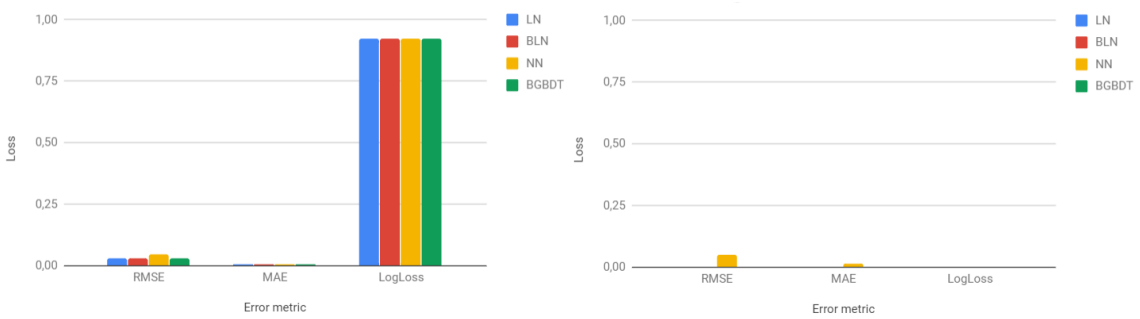


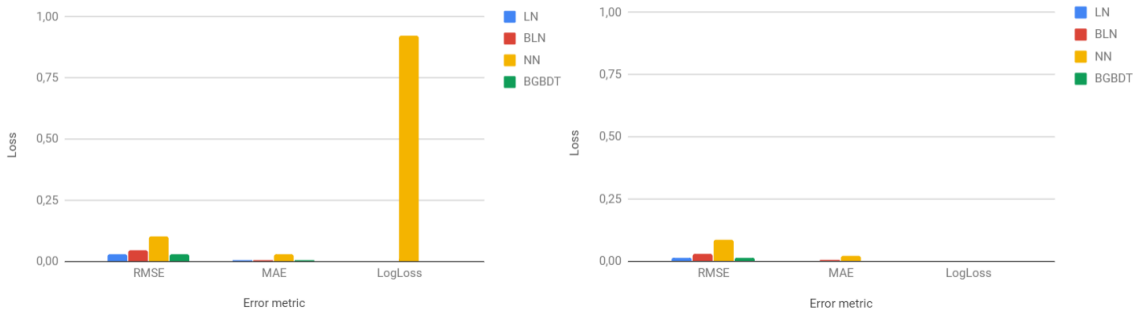
Figure 12: Comparing time to train the system for different algorithms

Furthermore, there are still some abnormalities with LogLoss. This is once again probably because of the one or two users that got wrong estimations on more recommendation algorithms and heavily influence the outcome because it is so large. Because of this, we advise in Chapter 10 to use another measure instead of LogLoss to rate the results.



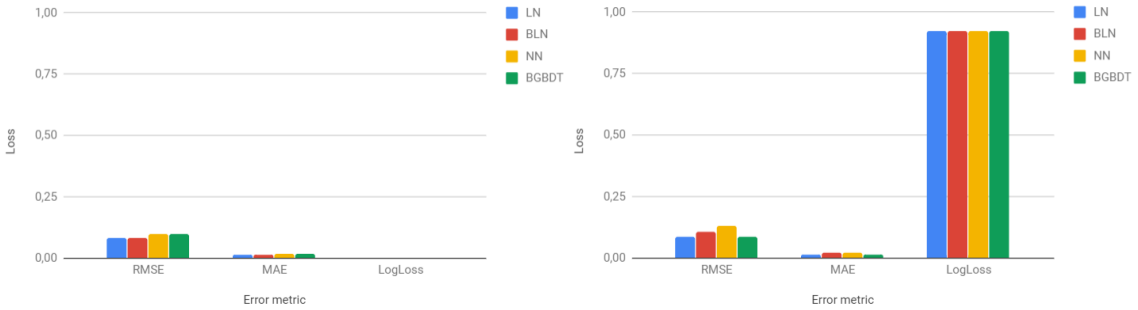
(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 13: Maximum of 0.01% of data filled in per user, average of 0.005%, 30.0% of the users



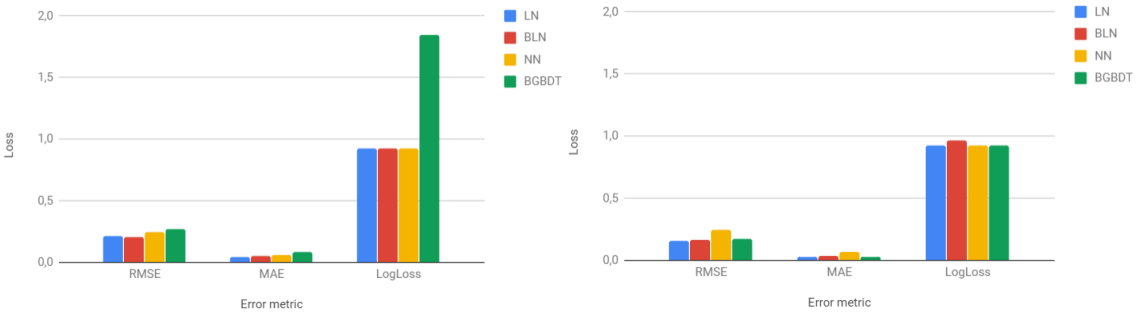
(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 14: Between 0.01% and 0.02% of data filled in per user, average of 0.012%, 21.5% of the users



(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 15: Between 0.02% and 0.05% of data filled in per user, average of 0.032%, 27.7% of the users

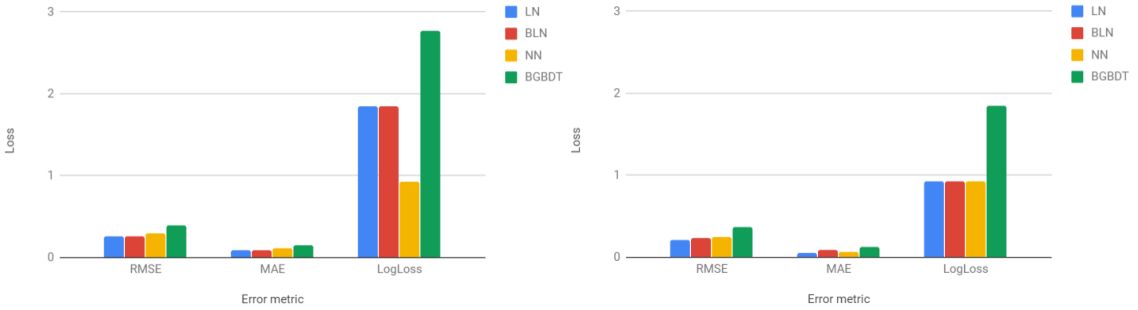


(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 16: Between 0.05% and 0.1% of data filled in per user, average of 0.071%, 11.3% of the users

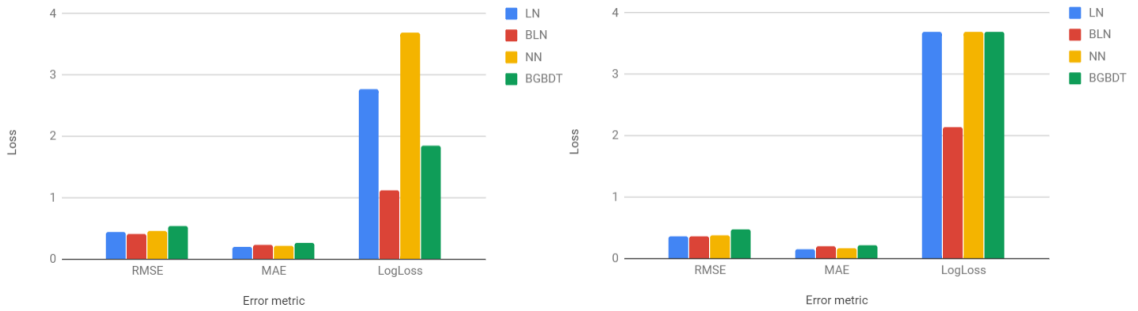
(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 17: Between 0.2% and 0.5% of data filled in per user, average of 0.35%, 7.9% of the users



(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 18: Between 0.5% and 1.0% of data filled in per user, average of 0.60%, 5.4% of the users



(a) Blending 6 recommendation algorithms (b) Blending 36 recommendation algorithms

Figure 19: More than 1% of data filled in per user, average of 1.22%, 2.2% of the users

9 Case Study: Squeezely

As mentioned before this thesis is written in cooperation with Squeezely, a startup company in online marketing. Their question revolves around how they can set up a data mining platform, linked to their own data, and usable for different clients without changing too much for each client. In this chapter, we will discuss which platform we chose, why we chose it and how we implemented it. This includes hurdles that came up during developing the platform.

9.1 Selecting a platform

To achieve these results we looked into three different possibilities of setting up such a platform: PredictionIO [42], Microsoft Azure AI [43], and a simple server with a completely homemade recommender system making use of different python libraries [38][44]. Here we will compare the different possibilities and motivate why we have chosen to work in PredictionIO in the end.

9.1.1 PredictionIO

PredictionIO [42] is an open-source platform from the Apache software foundation. It can be installed as a full machine learning stack, bundled with Apache Spark, MLlib, HBase, Spray and Elasticsearch. There are multiple templates that are ready to use for different prediction purposes. It also allows for setting up a database to store your data and save trained models for fast prediction.

PredictionIO is split into 4 components, they call this DASE. DASE stands for *Data source and preparator*, *Algorithm*, *Serving*, and *Evaluation metrics*. A representation of how these components work together is given in Figure 20. To work with PredictionIO the data should first be loaded in the Data source. This is the place where all data is saved. This includes the query that needs to be answered and the event data. This data is then prepared by splitting it into a test and a training group. The training data is sent to the algorithms, while the test data is given to the evaluation part.

The greatest advantage of PredictionIO is in the algorithm part. PredictionIO is built in such a way that multiple models can be trained and used at the same time. These algorithms can be freely exchanged with each other and for nearly all algorithms in the spark machine learning library (MLlib) there is already an algorithm ready to run that can be copied from a template. Furthermore, PredictionIO supports Python, Scala and Java so, in theory, all libraries from these can be used to create algorithms in PredictionIO.

After all, algorithms have given a result, these results will be combined in the serving part. In theory, this is where you want to use blending to combine all results to get your final estimation. From here on you can either directly output your results as a prediction or you continue to the evaluation metrics. In the evaluation metrics, the results are compared against the actual results and an estimation is given of how good your predictions are. The serving and evaluation metrics are completely customizable to serve the needs of the user.

The main disadvantages of PredictionIO are the difficulty of getting into the code and the need for prior knowledge of data mining and recommender systems. Where Microsoft Azure AI has everything pre-made when algorithms are concerned, PredictionIO works optimally if you manually add new algorithms and tune parameters. For this, you have to completely understand how PredictionIO works as a lot of things work over different files. This also asks for a wider knowledge of how all algorithms are used, what data is needed and where that data is stored in the system. Lastly, there is no real support you can contact when things are unclear, the best thing you can do is post your problems and questions on dedicated forums and hope the community around it can help. The documentation on the site is clear of itself but only covers the bare minimum of the possibilities, and even then there are some parts left out to complete a change.

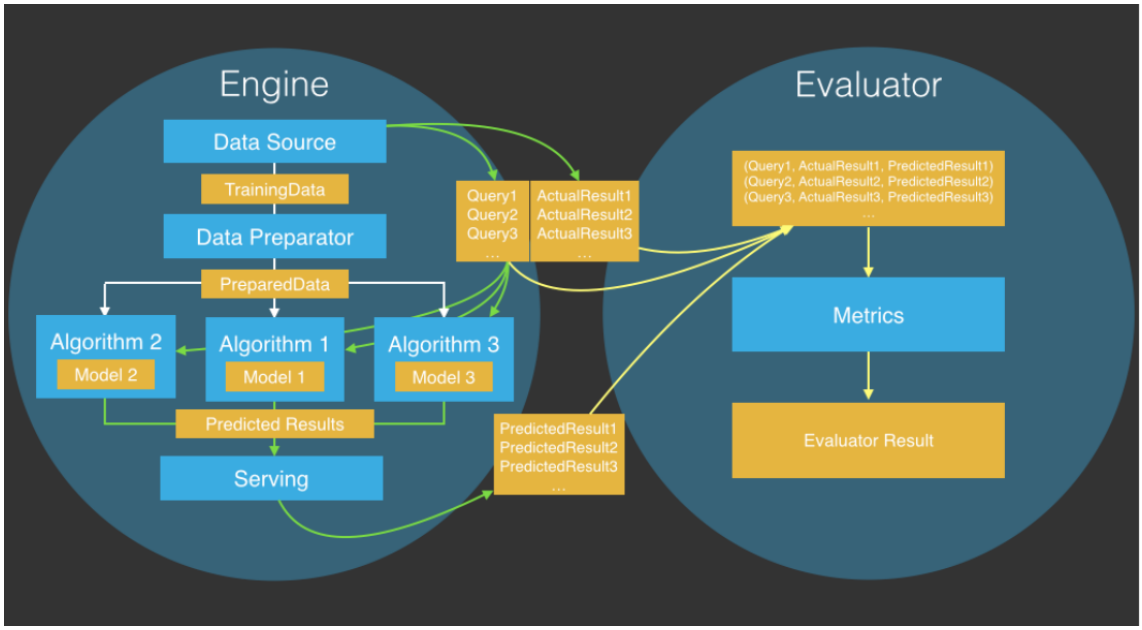


Figure 20: The general structure of PredictionIO

All in all, this makes PredictionIO a powerful prediction platform when used right but it does take some time to learn how it can be used and what is possible in which step of the DASE components. This makes it so that to work with it in a company setting there should be someone made available who knows how everything works, but once the ins and outs of the system are clear changes can be made fast and it will rarely lead to unforeseen problems.

9.1.2 Microsoft Azure AI

Another option that has been discussed for building a recommendation platform is Microsoft Azure AI [43]. Microsoft Azure AI combines the data storing of Microsoft Azure and machine learning. For this, they have a large amount of pre-built algorithms. The only job of the user of the system is to link all parts together. For this, only a small amount of knowledge about data mining is needed as the implementation of algorithms is already done. The only part that still needs some basic ideas on data mining is the selection of the data mining algorithm that suits your needs. To assist in this Microsoft has made the *cheat sheet* in Figure 21. This includes all possible kinds of algorithms and gives an idea of what they should be used for.

The closest thing Microsoft Azure AI has to offer for building a recommender system are regression and classification algorithms. Originally regression algorithms are used to predict values instead of real recommendations. But by saving the different events as a numeric value it is possible to use it as a recommender system. To make this work the events should be given a number representative to how likely it is that a person is interested in a certain item. An example could be that when a person has seen an item this value will be 1 if it was added to the cart it will be 3 and if it was bought it will be 5. Every other case will be 0. The regression algorithm can then be used to calculate an expected value for certain user-item combinations and find the best fits. As we saved the event as a numeric value anyways in the database this is not a problem.

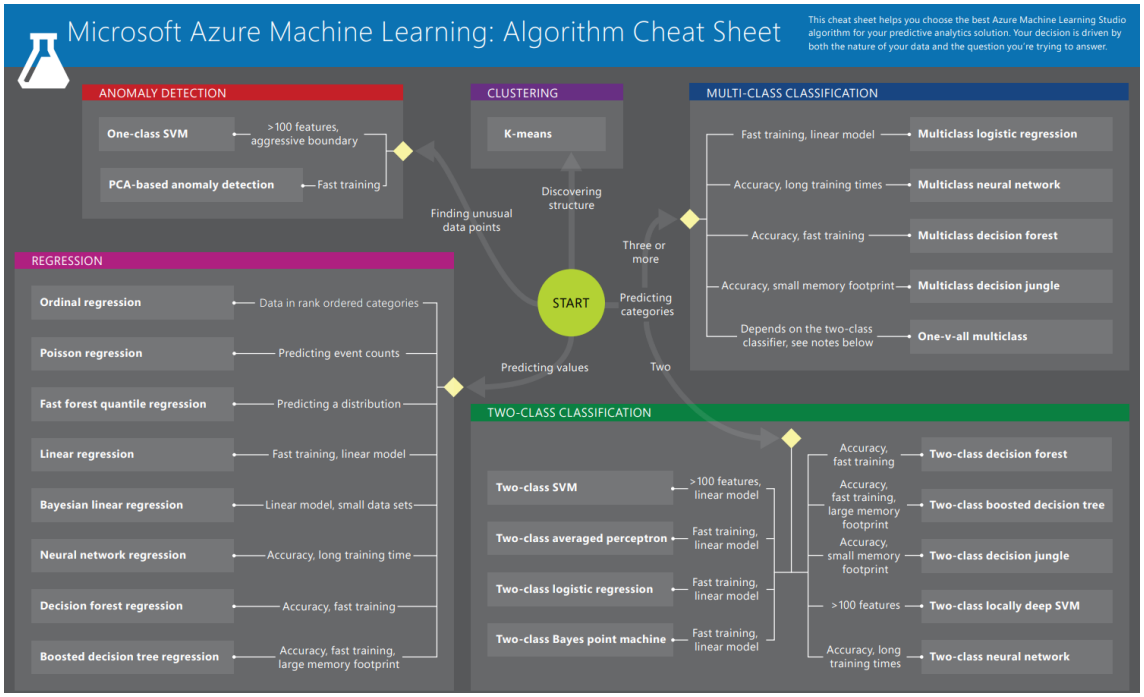


Figure 21: The algorithms of Microsoft Azure AI

Classification algorithms can be used when thinking about every event as a category you want a user-item combination to fall in. In the simplest case, this will be interested and not interested but it may also be the exact events that a user can take (view, buy, none). Classification algorithms are in our case used after the recommendation algorithms to blend the results and map them to events. In Microsoft Azure AI this can thus also be done in the same way.

In the end, we did not choose to work with Microsoft Azure AI. This had the following reasons: Although it is easy to work with, it is not possible to add your own algorithms to it or modify existing algorithms. For the most part, the strength of the prediction is thus depending on the algorithms Microsoft offers. Secondly, it is a paid service, thus it is more expensive than PredictionIO that is an open source project. Furthermore, it is closely connected to Microsoft Azure and thus limits the possibility to work together with competitors of Microsoft on hosting and recommendation.

9.1.3 A simple server with python

A final possible option to build a recommendation system is through a simple server with python. The idea behind it is to use multiple libraries in Python to get the recommendation and blending algorithms. The data can be uploaded to this server, or if more time is given this can be done automatically by building an API for the server.

The advantages of this are that it is completely customizable to the needs of Squeezely, only knowledge about the API is needed when using it and for changing algorithms other libraries in python can be used. Furthermore, as it does not use an open source platform like PredictionIO it gives Squeezely a better competitive edge, most of the knowledge on how to build and use such a system is not freely available.

There are, however, also disadvantages to using such a server. As there is no real documentation or help from third parties Squeezely will need someone that knows how everything works to write documentation on how to work with it. It is also needed to write the API to easily work with it, this adds an extra project before it can be fully functional. Lastly, none of the current employees at Squeezely has the knowledge to expand on the data mining algorithms, so the server will only be used as is and ideas from new researches will not be implemented.

Because of these reasons, the idea of building such a system was shut down by Squeezely. It would take too much time to have someone become familiar with everything relevant in data mining for them. In PredictionIO it will be easier to implement new algorithms as others are continuously developing them en making them available for others to implement.

9.1.4 Conclusion

In Table 3 an overview is given of the scalability, flexibility, adaptability, and affordability of PredictionIO, Microsoft Azure AI and a python server. With scalability, we mean how easy it is to add more memory and computing power. For PredictionIO and Microsoft Azure AI, this is included in the system itself, for a python server, this can either be also with AWS or by dedicating a local server to it.

With flexibility, we mean the how in how far the system can be changed to your own needs. Adaptability is how easy it is to make those changes. Microsoft Azure AI has the advantage that the whole interface is made to easily change how predictions are made, in PredictionIO and on the python server these changes have to be made through adjusting the code itself. Although this is harder to understand in most cases and more prone to mistakes it gives way more freedom to build your own product.

Lastly, there is the affordability, this is how much it will cost to build the platform and keep it up and running for each client. Using Microsoft Azure AI is quite expensive as it needs a lot of supporting products from Microsoft, which are costly. For working with a python server, a dedicated programmer and data scientist is needed. This because there is not much documentation of the program itself so the programmer should be knowledgeable about data mining and the algorithms applied. A programmer like this is expensive in training and/or salary. For working with PredictionIO this is less the case as there is more documentation and multiple online forums where help can be found.

After some discussion, the choice was made to work with PredictionIO. This choice was made because PredictionIO had no need to build an API, as it was already available. Furthermore, the source code is freely available to edit that gives a great adaptability. Moreover, by running it on AWS it can easily be scaled up if needed, without the need to move the system. Building all this on a simple server would take too much time as the communication with the Squeezely platform also needed to be built. Working with Microsoft Azure AI gives up too much freedom in choosing the algorithms and environment to run the predictions.

PredictionIO was well structured, the steps in the prediction process looked like a good fit to the combination of recommender systems and blending stated in this paper. There were also a lot of different examples already implemented for us to continue working on and making them suitable for Squeezely. Furthermore, if other people added a new algorithm to a template we can use those algorithms to improve

our own system. Lastly, it also supported Python which was the language I am most familiar with and the first tests on recommender systems were done in.

	Scalability	Flexibility	Adaptability	Affordability
PredictionIO	High	Medium	Medium	Low
Microsoft Azure AI	High	Low	High	High
Python server	Medium	High	Medium	Medium

Table 3: Comparing different ways to implement a recommender system in the business

9.2 Building an engine in PredictionIO

As a starting point, we used a template working with the IRIS dataset. This because it was one of the few templates available in Python for PredictionIO. The first hurdle we had to take was reading our own data into the system. For this, we chose to do a batch import through a JSON file. Any further updates on the dataset could be done through the API of PredictionIO once it was life. On the site of PredictionIO, there was a tutorial on how to change the program to work with different kinds of data. However, that assumed a certain standard setup that was of a previous version of PredictionIO. This lead to this costing more time than originally planned.

After the data was imported it was unclear where and how the data was stored. As it turned out one of the essential functions to read other forms of data than just the IRIS data was missing in Python. This lead to the actual events not being available to feed to the algorithms in PredictionIO. Because of this reason we made a switch to Scala. There were more examples/templates available for Scala, even though the python libraries used previously needed to be replaced.

When working in Scala there were a multitude of files that all add a small part to the whole platform of PredictionIO. These files followed more or less the structure of DASE. There was a python file to read all data in a batch import and the rest of the files were all in Scala. The data source and preperator are split in DataSource.scala and Preperator.scala. DataSource.scala reads and, interprets the data. Preperator.scala stores all the data in such a way that it is accessible. The algorithms are split into Algorithm.scala and Model.scala where all the models of the used algorithms are initiated in Model.scala and their functions further specified in Algorithm.scala. Serving is done in Serving.scala and evaluation in Evaluation.scala. Initiating all variables is done through Engine.scala and Engine.json.

The one problem faced when editing all these files is that it is less modular than it looks. A lot of files share classes and once one is changed it will give errors in other files. This makes it harder to add new algorithms of a different kind into the system. To better prepare Squeezely to do this themselves the places where changes need to be made for implementing a new algorithm were written down in a separate file. These places are:

- In Model.scala add the model parameters and variables, the input used here for the model is initialized in Engine.json. The functions to save and load (apply) the model should also be specified here.

- In Algorithm.scala the training and prediction algorithms should be added. As the input, it can use the variables specified in Model.scala.
- In Engine.scala there should be a mapping in the form of "name" -> classOf[algorithmClass] to create an instance of the algorithm.
- In Engine.json the parameters to call the instance should be specified.

Another problem that appeared was that it is impossible to train a model for serving (as is needed with blending) before the serving component. Even if it is trained during serving it is only possible by using an "algorithm" that predicts the known actual values. This also brings the extra disadvantage of having to train the model again for every query asked, which takes some extra time.

Another possible solution is to move the blending a step back from the serving phase to the algorithm phase. This will make it possible to train the blending algorithm at the same time as the other algorithms. However, this makes it much harder to add algorithms, as they either have to be implemented in this even more complex structure or completely ignore them when blending and take some kind of weighted average in serving.

As the time to make a recommendation is less of a factor than the accuracy, clarity and ease of implementing for Squeezely we have chosen to try the first solution to this issue. However, after working on implementing blending for about two weeks in total (including testing multiple approaches) we came to the conclusion that PredictionIO was not suited to facilitate blending. This was quite a disappointment as that was what we wanted to implement for Squeezely.

Without blending the different accuracies where only about half as good, as now instead of blending a weighted average is taken. The weights are calculated by how much standard differentiations an estimation is from the mean. This way an outstandingly good or bad estimation will be prioritized or penalized accordingly. This could be done as no training or added data is needed in the serving part.

Now the final thing to do is to try to add different algorithms to PredictionIO. Preferably these algorithms are taken from already existing examples and require minimal adaptations to work. However, with the experience gained we think that will still take up quite some extra time. Therefore the choice was made to leave the program as it is now.

Another reason that helped to motivate this decision was by using another evaluation metric, precision at k. Precision at k means how much of the top k estimations that should be recommended are recommended. This was on average about 80% for $k = 10$. Squeezely was satisfied with this result as they do not want to give too much different recommendations in their advertisements.

To answer the business question we can conclude the following: Although PredictionIO missed some important points in not facilitating blending and it takes quite some time to learn how to access all data, the resulting system can easily receive new data through an already build API and can give the estimations in real-time through the same API with an accuracy that is satisfactory while running within the time and memory constraints set by the company. There might be a lot of room to improve our current implementation in PredictionIO but the system is up and running and now it is waiting for the first real-life test cases to prove that it is a success.

10 Conclusion and Discussion

After analyzing our mock-up program in python we can now finally answer our research question: How can a small company with limited resources (time, money, knowledge and computers) set up a data mining environment for different clients and in how far can this compete with scientific approaches? We decided to use the results from the mock-up program in Python as the program in PredictionIO was not suitable to output all details that we wanted.

We have shown that it is possible to build a recommender system from scratch in Python by using different libraries for data mining, as well as in Scala through PredictionIO. These programs were running on a laptop with an i7 processor with four cores and 16 GB RAM, and a server with 4 cores and 8 GB of RAM from Amazon Web Services (AWS). This proves that with fairly low resources good results can be achieved. However, because of the nature of AWS, it is easy to get more memory and computing power by paying more. So if in future tests higher requirements are needed this can be made available. But as this is just the first round of testing it is not further taken into account.

How far these were able to compete with scientific results, however, remains unclear. This is the case because most papers about the RecSys dataset use a special kind of scoring, as this was used during the RecSys challenge [45]. Other papers mainly used recall@k or Mean Reciprocal Rank at k (MRR@k) [46]. Recall@k looks if the top k predicted items and calculates the proportion of cases that these items are in the top k in all test cases. MMR@k also includes the ranks that certain items should have had.

Recall@k is similar to the precision at k used by PredictionIO, the main difference is however that recall focuses on specific items in the top and precision on just positive items. Therefore these two are not directly comparable. For any further study, we should thus include better results on recall@k, MRR@k, and/or precision at k.

One thing that could still be improved in this thesis is adding the precision at k to the results. This as this metric is used to motivate the decisions for Squeezely and is, in general, easier to understand than LogLoss. Furthermore, LogLoss is too much influenced by single wrong estimations and thus gives a distorted view of the differences between setups. By replacing LogLoss with precision at k the graphs are also able to fit on a scale of 0 to 1 which makes it clearer to see the differences between the RMSE and MAE.

A question that is still left open for further research is how demographic filtering, context-aware filtering and social-based filtering can be added to the current structure and how does this influence the resources needed. This could be asked for the research program in python as well as the business case in PredictionIO. The main restrictions for using this will still be getting the most out of the low amount of computing power while still being able to predict in real-time.

On the other hand, it might also be possible to improve the current program by further testing other, not discussed recommendation algorithms, loss functions, training functions, and blending approaches. For example, more advanced variations of gradient descent can be used for training or newly invented loss functions can be used.

With this thesis, we gave a general overview of how recommender systems work and the individual parts they are made of. We also compared multiple different variations of recommender systems and explored their strong and weak points. Lastly, we analyzed a business in online marketing and build a recommender system to help their customers give advises on what to advertise.

References

[1] Squeezely, "Squeezely data management platform," 2016. Last accessed 16 May 2018.

[2] J. Bennett, S. Lanning, *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, p. 35, New York, NY, USA, 2007.

[3] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowledge-based systems*, vol. 46, pp. 109–132, 2013.

[4] J. Niu, L. Wang, X. Liu, and S. Yu, "Fuir: Fusing user and item information to deal with data sparsity by using side information in recommendation systems," *Journal of Network and Computer Applications*, vol. 70, pp. 41–50, 2016.

[5] M. Jahrer, A. Töscher, and R. Legenstein, "Combining predictions for accurate recommender systems," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 693–702, ACM, 2010.

[6] J. Lu, D. Wu, M. Mao, W. Wang, and G. Zhang, "Recommender system application developments: a survey," *Decision Support Systems*, vol. 74, pp. 12–32, 2015.

[7] M. E. Wall, A. Rechtsteiner, and L. M. Rocha, "Singular value decomposition and principal component analysis," in *A practical approach to microarray data analysis*, pp. 91–109, Springer, 2003.

[8] A. Paterek, "Improving regularized singular value decomposition for collaborative filtering," in *Proceedings of KDD cup and workshop*, vol. 2007, pp. 5–8, 2007.

[9] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[10] N. Bhatia *et al.*, "Survey of nearest neighbor techniques," *arXiv preprint arXiv:1007.0085*, 2010.

[11] M. J. Pazzani and D. Billsus, "Content-based recommendation systems," in *The adaptive web*, pp. 325–341, Springer, 2007.

[12] M. G. Vozalis and K. G. Margaritis, "Using svd and demographic data for the enhancement of generalized collaborative filtering," *Information Sciences*, vol. 177, no. 15, pp. 3017–3037, 2007.

[13] G. Adomavicius and A. Tuzhilin, "Context-aware recommender systems," in *Recommender systems handbook*, pp. 191–226, Springer, 2015.

[14] A. Karatzoglou, X. Amatriain, L. Baltrunas, and N. Oliver, "Multiverse recommendation: n-dimensional tensor factorization for context-aware collaborative filtering," in *Proceedings of the fourth ACM conference on Recommender systems*, pp. 79–86, ACM, 2010.

[15] H. Ma, D. Zhou, C. Liu, M. R. Lyu, and I. King, "Recommender systems with social regularization," in *Proceedings of the fourth ACM international conference on Web search and data mining*, pp. 287–296, ACM, 2011.

[16] F. E. Walter, S. Battiston, and F. Schweitzer, "A model of a trust-based recommendation system on a social network," *Autonomous Agents and Multi-Agent Systems*, vol. 16, no. 1, pp. 57–74, 2008.

[17] T. Hofmann, "Latent semantic models for collaborative filtering," *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 89–115, 2004.

[18] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "Bpr: Bayesian personalized ranking from implicit feedback," in *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*, pp. 452–461, AUAI Press, 2009.

[19] J. Weston, S. Bengio, and N. Usunier, "Wsabie: Scaling up to large vocabulary image annotation," in *IJCAI*, vol. 11, pp. 2764–2770, 2011.

[20] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate research*, vol. 30, no. 1, pp. 79–82, 2005.

[21] T. Chai and R. R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)?–arguments against avoiding rmse in the literature," *Geoscientific model development*, vol. 7, no. 3, pp. 1247–1250, 2014.

[22] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, 2009.

[23] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*, pp. 421–436, Springer, 2012.

[24] S. M. Stigler, *The history of statistics: The measurement of uncertainty before 1900*. Harvard University Press, 1986.

[25] H. Robbins and D. Siegmund, "A convergence theorem for non negative almost supermartingales and some applications," in *Optimizing methods in statistics*, pp. 233–257, Elsevier, 1971.

[26] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *International Conference on Algorithmic Applications in Management*, pp. 337–348, Springer, 2008.

[27] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[28] R. J. Schalkoff, *Artificial neural networks*, vol. 1. McGraw-Hill New York, 1997.

[29] Y. Ganjisaffar, R. Caruana, and C. V. Lopes, "Bagging gradient-boosted trees for high precision, low variance ranking models," in *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pp. 85–94, ACM, 2011.

[30] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[31] B. Lika, K. Kolomvatsos, and S. Hadjiefthymiades, "Facing the cold start problem in recommender systems," *Expert Systems with Applications*, vol. 41, no. 4, pp. 2065–2073, 2014.

[32] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock, "Methods and metrics for cold-start recommendations," in *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 253–260, ACM, 2002.

[33] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, 2009.

[34] D. Anand and K. K. Bharadwaj, "Utilizing various sparsity measures for enhancing accuracy of collaborative recommender systems based on local and global similarities," *Expert systems with applications*, vol. 38, no. 5, pp. 5101–5109, 2011.

[35] J. Herlocker, J. A. Konstan, and J. Riedl, "An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms," *Information retrieval*, vol. 5, no. 4, pp. 287–310, 2002.

[36] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, "Evaluating collaborative filtering recommender systems," *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 5–53, 2004.

[37] R. Challenge, "Recsys challenge 2015 yoochoose," 2015. Last accessed 15 June 2018.

[38] M. Kula, "Metadata embeddings for user and item cold-start recommendations," in *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.* (T. Bogers and M. Koolen, eds.), vol. 1448 of *CEUR Workshop Proceedings*, pp. 14–21, CEUR-WS.org, 2015.

[39] R. M. Bell, Y. Koren, and C. Volinsky, "The bellkor 2008 solution to the netflix prize," *Statistics Research Department at AT&T Research*, vol. 1, 2008.

[40] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[41] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[42] T. A. software foundation, "Predictionio," 2018. Last accessed 8 March 2018.

[43] Microsoft, "Ai-platform microsoft azure," 2018. Last accessed 8 March 2018.

[44] Github, "Implicit library for python," 2013. Last accessed 26 March 2018.

[45] D. Ben-Shimon, A. Tsikinovsky, M. Friedmann, B. Shapira, L. Rokach, and J. Hoerle, "Recsys challenge 2015 and the yoochoose dataset," in *Proceedings of the 9th ACM Conference on Recommender Systems*, pp. 357–358, ACM, 2015.

[46] B. Hidasi, A. Karatzoglou, L. Baltrunas, and D. Tikk, "Session-based recommendations with recurrent neural networks," *arXiv preprint arXiv:1511.06939*, 2015.