Online recommendations at web-scale using matrix factorisation

Marcus Ljungblad
marcus@ljungblad.nu

Royal Institute of Technology, Stockholm, Sweden
Instituto Superior Técnico, Lisbon, Portugal
Universitat Politècnica de Catalunya, Barcelona, Spain

June 15, 2012

Supervisor: Professor José Luis Balcázar
Abstract

In social networks, e-commerce systems, and other web-services the sheer size of available content is overwhelming. Highlighting relevant content is the focus of recommender systems. Most previous research in the area has provided several algorithms for personalising the user experience, but few have addressed the issues of scalability. In this study we show how matrix factorisation, one of the more accurate recommendation techniques, can be used to serve recommendations online for millions of items and millions of users. An approach based on dividing all available items in clusters and restricting the computation to a selected few is outlined. Consequently, we developed a prototype using requirements from a production environment to demonstrate its feasibility. Experimental results show that 600 recommendation requests per second can be served with a latency below 30 ms. We conclude that matrix factorisation can be used online in large-scale settings but specific care has to be taken when clustering the items.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Popular itemsets</td>
<td>32</td>
</tr>
<tr>
<td>5.7</td>
<td>Failure scenarios</td>
<td>32</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Worker crashes</td>
<td>32</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Node crashes</td>
<td>33</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Failures mitigated by timeouts</td>
<td>33</td>
</tr>
<tr>
<td>5.8</td>
<td>New users and items</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>Results</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>Prototype</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>Accuracy of recommendations</td>
<td>35</td>
</tr>
<tr>
<td>6.3</td>
<td>Scalability and Performance</td>
<td>37</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Single node</td>
<td>37</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Multiple nodes</td>
<td>41</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Scaling the interface</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Discussion</td>
<td>48</td>
</tr>
<tr>
<td>7.1</td>
<td>Main findings</td>
<td>48</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Scalability</td>
<td>48</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Recommendation quality</td>
<td>49</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Summary</td>
<td>49</td>
</tr>
<tr>
<td>7.2</td>
<td>Positioning</td>
<td>50</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Comparison to existing systems</td>
<td>50</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Our prototype vs stream processing</td>
<td>51</td>
</tr>
<tr>
<td>7.3</td>
<td>Future work</td>
<td>51</td>
</tr>
<tr>
<td>7.3.1</td>
<td>System specific improvements</td>
<td>51</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Recommendation freshness</td>
<td>52</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Research directions for recommender systems</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>53</td>
</tr>
</tbody>
</table>
1 Introduction

On the Spanish social network Tuenti over 13 million users can, as of today, interact with 306 years worth of videos. Selecting the videos that a user might be interested in is incredibly challenging due to the vast size of the data and its daily growth. Special crafted recommendation algorithms churn over massive amounts of log data to propose interesting content to the network’s users. The recommendation algorithms have become an incredible resource for driving further interaction. In essence, they take the role of content editors creating a personalised experience of the social network unique to every user. The personalisation leads to an improved user-experience as well as a prolonged commitment to the service. For example, the CEO of the online retailer Amazon, the company who re-invented e-commerce, calls recommendations a key differentiating factor [28]. Likewise, everyday millions of videos are played on Tuenti. Each of those plays are logged such that the information can be analysed and used to propose other videos a user is likely to find interesting. At the same time, every view is taking place under unique circumstances and adapting the recommendations to the users current environment, the context, is necessary to provide more accurate suggestions [26, Ch. 7].

However, despite algorithmic advancements in recommender systems and the developments of distributed computing frameworks, limited research on applying these algorithms for large-scale datasets exists. Research has focused on improving the personalisations and comparing different approaches to each other [4]. Only minimal, and arguably insufficient, performance improvements to the algorithms to handle large datasets have been proposed. The few developments which exists to improve the algorithms’ scalability focus on adapting the algorithms for existing distributed frameworks [3, 21]. Sadly, these suggestions are still not on-par with the requirements of web-services such as Tuenti. Primarily because these attempts are batch-oriented and ill-fit for services that attracts millions of users and items and requires recommendations to be delivered online. Novel approaches are needed to address the problem: most algorithms are designed for itemsets with up to a few million items, and they are designed for offline usage. Moreover, supporting context in real-time is challenging and using the offline algorithms online is not straightforward endeavour.

The majority of recommenders are based on a split offline and online approach. During an offline phase large quantities of data is processed. For each user the algorithm produces a set of relevant content which, in the second phase, is sorted and served online. Some of these algorithms have been adapted to work with stream processing systems to improve the freshness of the recommendations. Nevertheless, matrix factorisation, one of the more accurate methods to provide recommendations, is still not readily available for large-scale online recommendations due to its scalability challenges. Therefore, we suggest an architecture of a recommender designed to serve millions of items to millions of users using this mathematical model. We demonstrate its feasibility through a prototype that is evaluated according to the requirements of Tuenti.

1.1 Tuenti

Tuenti is a Spanish social network founded in 2006. It uses an invitation-only model and its primary target group are 14-25 year olds. Popular services provided by Tuenti includes photo and video sharing, private and group chat, online games as well as a media portal. The latter is regularly updated with music videos and popular TV-shows.
targeted at the Spanish population.

The average user on Tuenti is 24 years old and have 135 friends. 60% of the users log in every day and the average session length is a staggering 90 minutes long. Tuenti, for many of its users, is the primary mean for communicating with friends.

### 1.2 Contributions

Given the limited extant research on large-scale recommender systems and the promising results of matrix factorisation, including its use with contextual data [7], we designed a system that serves recommendations online employing this approach. We needed to elicit a way to minimise online computations such that recommendations could be served as the user requested the recommendations, limiting the computation time to no more than 200 ms. Robust tools for data collection already exists, for example Scribe, thus we have not concerned ourselves with recording user activity, only serving recommendations. Moreover, there are a number of other factors which influence the quality of the recommendations, including the specifics of the matrix factorisation model. We use a model for implicit feedback data but our proposal is agnostic to the underlying matrix factorisation model. In this report we do not study which model is the best suited for the Tuenti use-case.

The primary use-case for the proposed recommender is formed around Tuenti’s video functionality. Hence, the majority of examples throughout this report will concern video recommendations. It should be noted that the terms video and item may be used interchangeably and the proposed recommender can in fact support any form of content. We provide a deeper explanation of this in section 3.6.

Our prototype handles on average 600 requests per second spread across three nodes and serves recommendations with some loss of quality due to reduced item coverage. It does so by creating clusters of items, and computing recommendations from a subset of those. Quality of recommendations are dependent on how well one selects the clusters most relevant to the current user and context. Accuracy loss was measured by comparing it to an offline test were all items were available.

Consequently, this report contains three primary contributions. An overview of existing methods used in large-scale recommender systems particularly focusing on implicit feedback and collaborative filtering. Second, a conceptual approach and demonstration of how matrix factorisation on an implicit feedback dataset can be used to create recommendations online which account for context at large-scale. And finally, an architecture and prototype implementation which supports hundreds of requests per second.

### 1.3 Structure of the report

We begin with a background to recommendation algorithms (section 2), primarily those of collaborative-filtering. Following an overview of current production systems using these algorithms (section 2.3) we explain our concept based on matrix factorisation (section 3). The subsequent section outlines the proposed architecture (section 4) and we follow up with implementation details in section 5. Experimental performance and accuracy results are presented in section 6, with a follow-up discussion in section 7. Eventually we conclude the report and summarize our findings (section 8).
2 Recommendation Techniques

In this section we overview existing recommendation techniques and explore classes of algorithms and their properties. Thereafter we survey recommendation systems used in the industry today, including both large-scale proprietary and open source alternatives. Finally a background to matrix factorisation is presented as that forms the basis for the recommendation approach proposed in this work.

2.1 Approaches to recommendations

The generation of recommendations can generally be divided into two extremes: content-based recommendations and collaborative filtering. Naturally, a merge between the two composes a large middle-ground and are often referred to as hybrid models. The de-facto standard employed today is collaborative filtering as it tend to yield better results than content-based approaches. Historically, however, the first attempts to personalisation were entirely content-based [8]. Today the content-based approach is insufficient to deal with the diverse range of content available since those methods rely on an interpretation of the quality of an item. Interpreting textual data is a matter of great study, let alone analysing video or music. However, by focusing on user-item interactions we can provide predictions without the need to fully understanding the items themselves. In other words, what items users interact with, in which order they interact with them, and how they interact with them enables algorithms to be agnostic to the content itself. This interaction-approach is the foundation of collaborative filtering and will be the focus in this work.

Collaborative filtering works on the premise that two users who have interacted with the same item are likely to have similar preference for an item which one of them has not yet interacted with. It is the most popular approach to recommendations and is used in several large applications such as Netflix [18], Amazon [20], and Youtube [11]. If, for example, you were to buy a book on Amazon, the recommender would suggest books that other people, who bought the same book, also found interesting. There is between the books recommended a relationship based on the fact that they were bought together. The purchase does not necessarily have to happen at the same point in time, but it could. The link between items in the Amazon case is formed by the users who buy them. Depending on your similarity with those users (or interactions of those users) you may be recommended other interesting books.

Memory-based algorithms base predictions on entire entrysets of previously rated items. Given a number of similar users, an aggregate value of their ratings for a specific item is computed. A similarity measure between users are commonly used to weigh the importance of the respective ratings. In other words, the more similar two users are, the more important are the other user’s rating. The aggregate acts as an heuristic to predict future ratings. Furthermore, memory-based algorithms may choose between two different starting points when predicting recommendations for a user: either from other similar users, or from item relationships.

- User-item relationship - this approach begins with finding similar users to the one who we are trying to provide recommendations for. Ratings from those similar users determine the predictions. This works on the assumption that users like you interacted with these items, and so you might also want to interact with them because you are similar.
• Item-item relationship - in contrast to the user-item relationship this approach is based on the idea that items are interacted with together. It is succinctly captured in the following ”People who viewed this item also looked at ...” It is the method used by Amazon and Youtube.

One drawback with memory-based methods is that it is hard to scale to millions of items according to [4, 8]. However, Amazon is a good counter-example which successfully uses item-to-item collaborative filtering with large amounts of data [20].

Model-based algorithms usually stem from machine learning techniques, such as matrix factorisation, and use the collection of previous ratings to learn a model from which predictions are made. Alternative methods to matrix factorisation include probabilistic models using Bayesian networks, or statistical models using clustering. The main difference from memory-based algorithms is that the model is discovered, or trained, using the collected ratings and no heuristic is used to predict the future ratings. Moreover, [8] argues that model-based algorithms are more resilient to the cold-start and data-sparsity problem. In most cases the models tend to take a substantial time to generate, and complexity increases significantly with number of variables and data size.

A combination of both content-based and collaborative filtering is often used to overcome the so called ”cold start” problem. The problem mostly affects collaborative filtering approaches as those rely on the interactions with existing items. In other words, items that are never rated may never be recommended. A simple solution is to assign random ratings to new items such that they can be recommended too. [4] found that a hybrid often produces better results compared to any of the two other approaches used individually, especially with respect to the cold start problem.

2.2 Preparing for recommendations

Before building any form of recommendation engine it is essential to define what the users goals and tasks are. How will the recommendations be used? In [14] the authors identified a number of common tasks: annotation, finding good items, and browsing around, to name a few. Once the overall goal is defined, it is necessary to characterise the users and the environment they operate in. As in all form of user-evaluation, this becomes necessary to evaluate the quality of the recommender’s results. In the end it is the user’s level of satisfaction that matters the most measuring that the system achieves this should be our goal.

The properties of the feedback data must be considered [14] when designing a recommender. Feedback from the user with respect to items can be collected in two ways: implicit and explicit. Implicit data is, for example, clicks on an item, number of item views, or time spent interacting with an item. There is inherit noise in this data as it may not truly reflect the user’s interest in a particular item. You cannot know for sure whether a user liked the item or not just because she clicked on it. Explicit data, on the other hand, is data such as likes, up or down votes, or numerical ratings. While more reliable than implicit feedback, explicit feedback usually contains a bias. For example, a five-star rating for one user may not mean the same for other. Ratings may also change over time as new items become more popular and replaces older. Similarly, ratings are often one-dimensional, representing the overall impression of an item. Commonly items may be represented by more dimensions, for example, a restaurant may be judged by food quality, service, and interior atmosphere.
2.3 Generating recommendations online

Depending on the context in which you operate it may be sufficient to calculate recommendations periodically offline. Nevertheless, with a growing demand for staying up to date it is increasingly necessary to include users’s context when providing recommendations. This can be done if the recommendations are calculated in an online fashion. Context could be incorporated in offline models too, but they would not reflect the current situation only the situation in which the user interacted with an item. Algorithms such as [15, 17, 18] were never designed to be computed online directly. According to the survey [27] collaborative filtering methods can be used for thousands of items and users online. However, modern systems requires algorithms that scales to millions of users and tens of millions of items, and online computations are often limited by the dataset size.

It is acknowledged in [1] that most extant research focuses on the algorithms and little research on the systems supporting these algorithms exists. Based on three separate recommender implementations they encapsulate a common reference architecture. It consists of three components: a watcher (observing user activity), a learner (predicting relevant content), and an advisor (provides recommendations to users). However, the proposed architecture lacks recognition of large-scale datasets and is formed under the assumption that one machine is sufficient to handle all requests, users and items.

Two approaches have emerged for online recommendations: one which we call split models, where some part of the recommendation computations are performed offline, and the other stream models where data is continually analysed. In the following two sections we present an overview of each model focusing particularly on large-scale deployments.

2.3.1 Split models

There exists relatively little published material about production recommendation systems and their architecture. It is no secret that several web companies employ recommendation techniques and one of the more prominent systems paper is [10] which provides personalised news at Google News. They combine the results from three different algorithms according to a weighting scheme. Similarly to the proposed system their system is divided into two phases: offline periodic clustering of users based on click history, and an online phase for serving recommendations and updating item visitation counts. It is, however, substantially different as it relies on clustering techniques for generating recommendations instead of matrix factorisation. The strengths of the Google system is its ability to quickly incorporate new items, which is important for certain content such as news, and serve recommendations in only a few milliseconds.

Youtube and Amazon are also two notable players in the field of recommendations. Amazon has for a long time pioneered the area when they introduced product to product recommendations, ensuring that each user had a personalised view of the store. As expected Amazon must be able to scale to millions of users and millions of items and does so by computing the most expensive functions offline [20]. It is done by computing the cosine similarity between every single product pair where the vectors contain customers who bought each item respectively. This, obviously, is a very time-consuming task. The authors also point out the scalability limitations of traditional collaborative filtering algorithms, especially when the datasets are huge. Youtube, similar to Amazon, relies on an item to item approach but their model uses covisitation as the primary similarity measure [11]. Covisitation means videos that are seen directly
after another one. This is combined with a each user’s most recent activity on the site to achieve personalisation. Youtube smartly creates more recommendations than those seen immediately on the site. This means that every time a user loads a page they do not need to recompute the recommendation set. Furthermore, the video similarities are computed offline using batch jobs several times a day. The resulting recommendation dataset is served from read-only Bigtable servers.

It would not be fair to leave Netflix out from this discussion. In 2006 the movie rental service Netflix posed a challenge to researchers and volunteers around the world to improve their recommendation engine. The prize consisted of USD 1 million to those who could design an algorithm that performed \(10\%\)\(^1\) better than their existing algorithm. This incentive caused a dramatic spur in the academic recommendation community, and one of the most noteworthy take-aways from the challenge was the popularisation of matrix factorisation as a technique for predicting accurate recommendations [13]. Today it remains as one of the most accurate ways of generating recommendations and has been further refined by [15, 18, 25] to name a few. We shall look further at matrix factorisation for recommendations in section 2.4.

Hulu, a service similar to Netflix, employs a recommendation engine to highlight TV-shows that their customers may enjoy. The architecture of the recommendation engine is described in [31] and is split in two parts, an online serving and filtering part, and one computation and data processing part. A recommendation request is served by observing a user’s historical preferences and forming recommendations from those using item-based collaborative filtering. Afterwards, the raw recommendations are filtered from items the user has already seen. On the remaining items the recommendation engine tries to find an explanation as to why the following items were recommended. Hulu also shows that providing explanations can increase click-through significantly and emphasizes the CTR over conventional recommendation metrics such as RMSE, precision and recall (section 6.1).

Finally, another recommendation engine serving large-scale datasets is developed by Foursquare [22]. A feature called Explore allows a user to discover places around them. With Foursquare’s 10+ million users, item-based similarity scoring would result in over 100 trillion computations. The solution was to use the distributed framework Mahout and restricting it to process only relevant scores, for example by ignoring places which have not been checked into recently. Similarly to Hulu, Foursquare provides an explanation as to why a user should visit a place, for example, in “You’ve been to X, other users who went there also liked Y”. Hence, a split model is used where similarities between places are computed offline, and to serve a request the items are re-ranked according to the user’s geographical position.

2.3.2 Stream processing

An alternative to the aforementioned split models is being explored more and more with the increased availability and popularity of stream processing systems such as S4 (Yahoo!), Storm (Twitter), and StreamInsight (Microsoft). These data processing frameworks work by defining a set of queries through which data is continuously streamed, making them a good fit for problems with real-time requirements. Like MapReduce was built to process large amounts of data in batch jobs, these stream processing frameworks are meant to process large amounts of data in real-time.

\(^1\)Based on the root mean squared error.
In [12] an experimental recommendation model using stream processing frameworks was proposed. Their work focuses on capitalising on the real-time web for personalising news. By incorporating both users’ Twitter feeds and Yahoo! News data they filter and personalise news for users. For new users, when data is scarce, they explore the user’s social circle to learn more about the preferences the user might be interested in. The recommendation model uses content-similarity and a request is served by re-ranking a candidate set of the most relevant news stories. The authors go as far as saying that Google News personalisation [10] is insufficient as updates may arrive too late to detect emerging items quickly enough. It is obvious that the creators behind the two systems have a different opinion about what ”fast enough” is and as such the solutions are significantly different from each other. Arguably, their proposed system is not so much a recommendation engine as a system for detecting trending topics for individual users and finding similar content to those trending topics.

StreamRec [9] makes recommendations on request using item-based collaborative filtering by using a complex event processing approach. In contrast to traditional recommendation systems it incrementally updates the items most similar to each other based on user interactions. At any given time a bucket of similar items per item is maintained. It is the items in these buckets which are updated when interactions occur. Thus, when a request for recommendations arrive, it uses the last item the user interacted with as a reference, and re-ranks the items in its bucket according to the user. An interesting implementation when making recommendation requests in StreamRec is that they can be both push- and pull-based. The former behaves similar to the publisher-subscriber pattern, such that when interactions that modifies the ranking occur they are immediately propagated to the client. The pull-based method is equivalent to making a request for recommendations in any other recommender system.

Prismatic, a newcomer in the recommendations area, also adopts a stream processing model. The service provides personalised news using the input from your Twitter feed as well as how you interact with the items it serves you [24]. User feedback is implicit as all interaction with the items feed future recommendations. Though their system is proprietary, they provide some insight into how they are able to integrate signals on the fly. For example, in one blog post they describe real-time clustering of related news stories. From a system’s perspective it is interesting to note that they have chosen to develop own tools over using existing well-known frameworks such as MapReduce or Mahout. Finally, Prismatic also acknowledge that requests have to be served within about 200 ms. We shall address this in more detail in section 3.2. It will be interesting to see in the future how Prismatic handles increased load as they attract more users.

2.4 Matrix factorisation

Here we illustrate with an example how matrix factorisation can be used to estimate unknown ratings between a set of users and items. We will also look briefly on how context can be incorporated in matrix factorisation models.

2.4.1 General overview

Matrix factorisation is a mathematical technique to approximate unknown values based on some known ones. Let us provide a small example of movie ratings. Ratings are represented by a real number between 0 and 5. Now, imagine that we have a set of users and the movies these users have rated. No user have rated all movies, but all movies
contain at least some ratings. As seen in equation 1 we can store the user-movie rating relationship in a matrix. Here users are on the x-axis and the items they have rated on the y-axis. In this example we will assume that the ratings given are true and that no bias exists in the them. Some ratings are nevertheless missing, represented by a question mark in the matrix, and the first task to provide a set of recommendations is to predict how the user may have rated these movies if they had seen them.

The goal of matrix factorisation is two find two matrices such that when they are combined, they provide estimations of the known and unknown values. It will produce estimations of the known values too. Since some values are known, we can compare the same estimated values with those we know to guide the modelling process. This is done by minimising the error between the estimations and the true values. When the error is small enough, this will provide us with some confidence that the estimations of the unknown values are also good. By testing the approximation against the known values it is possible to determine when the estimations are "good enough". Several error estimation functions can be used, but one popular and commonly used is the root mean squared error (RMSE) which we will also use here.

\[
\begin{bmatrix}
  2 & 4 & 4 & ? & 1 \\
  3 & 5 & ? & ? & 1 \\
  ? & 4 & 2 & 1 & ? \\
  1 & ? & 1 & 3 & 3
\end{bmatrix}
\]  

(1)

There exists a variety of matrix factorisation algorithms but to illustrate how we can estimate the missing ratings in equation 1 we shall use alternating least squares (algorithm 1\(^2\)). The algorithm initialises two matrices with a predefined number of hidden features that we want to discover. The hidden features are real numbers that describe the characteristics of a user or video. The more features used, the more precise are the descriptions. Normally the number of hidden features ranges between 5 and 50. It goes without saying that the more features used, the more computationally heavy the algorithm becomes. Initially each position in the two matrices are initialised with a random integer.

As it turns out the two projection matrices represent users preferences and movie characteristics respectively (denoted \(\text{upm} \) and \(\text{mcm} \)). A Python implementation\(^3\) of Algorithm 1 illustrates how the unknown values are approximated (Equation 2). In this example the constants were set to the following: \(\text{noLatentFeatures} = 5, \lambda = 0.0002 \) and \(\alpha = 0.02\). The last two define the learning rate, i.e in which speed the algorithm adjusts the initially random numbers, and the regularisation factor, which is used to ensure that the values stay within a given function.

\[
\begin{bmatrix}
  2.05 & 4.07 & 3.91 & 2.32 & 0.85 \\
  2.94 & 4.95 & 4.63 & 3.18 & 1.12 \\
  0.89 & 3.95 & 2.02 & 1.05 & 1.23 \\
  1.01 & 4.93 & 1.02 & 2.96 & 2.97
\end{bmatrix}
\]

(2)

For this small example we can easily see that the predicted values for the already known values are close to each other. Concretely, the known value \([0][1]\) in equation 1 is almost equal to the predicted value in equation 2: \(\text{predicted}[0][1] \sim \text{input}[0][1]\) as \(4.07 \sim 4\). Consequently, we can assume that the predicted values for the unknown values are within an acceptable range of how the user might rate that movie if they

\(^2\)This implementation is adjusted from Albert Au Yeung’s example available at [6]

\(^3\)See [https://gist.github.com/1626941](https://gist.github.com/1626941) for the source code
Algorithm 1 Simplified pseudo-code for matrix factorisation, omitting the RMSE calculation for brevity.

\[
\begin{align*}
\text{upm} &\leftarrow \text{initRandomMatrix(InputMatrix.height, noLatentFeatures)} \\
\text{mcm} &\leftarrow \text{initRandomMatrix(InputMatrix.width, noLatentFeatures)} \\
\text{repeat} & \\
\text{for } u \rightarrow \text{InputMatrix.height} \text{ do} & \\
\text{for } m \rightarrow \text{InputMatrix.width} \text{ do} & \\
\text{if } \text{InputMatrix}[u][m] > 0 \text{ then} & \\
\text{est} &\leftarrow \text{InputMatrix}[u][m] - \text{upm}[u,:] \cdot \text{mcm}[:, m] \\
\text{for } f \rightarrow \text{noLatentFeatures} \text{ do} & \\
\text{upm}[u][f] &\leftarrow \text{upm}[u][f] + \lambda(2 \ast \text{est} \ast \text{mcm}[f][m] - \alpha \ast \text{upm}[u][f]) \\
\text{mcm}[f][m] &\leftarrow \text{mcm}[f][m] + \lambda(2 \ast \text{est} \ast \text{upm}[u][f] - \alpha \ast \text{mcm}[f][m]) \\
\text{end for} & \\
\text{end if} & \\
\text{end for} & \\
\text{end for} & \\
\text{until } \text{RMSE} \leq \text{threshold}
\end{align*}
\]

were to do so. By sorting the user’s predicted ratings and filtering out those that she has already seen we can recommend movies that the user may like in the future.

This example is obviously small and the rating data is dense, meaning there are a lot of ratings available. As the number of items increase data sparsity also increases, often necessitating smarter data structures which more efficiently handles sparse data. At Tuenti, for example, any given user only watches a small fraction of all the available videos leading to a large, and extremely sparse, input matrix. One challenge that has been studied by several authors as indicated by [4] is to scale matrix factorisation techniques to large number of users and items. Proposals to solve this issue include limiting the input data set, for example by removing old items, minimise the number of variables, use smarter data structures, or use a distributed computing framework such as Mahout [3] or GraphLab [2].

2.4.2 Context-aware matrix factorisation

Before describing our suggested approach using matrix factorisation, lets provide an introduction to Context-Aware Recommender Systems (CARS). CARS was introduced to counter the largest drawback of traditional recommender systems: namely, the lack of recommending with respect to the situation the user is currently in [26, Ch 7]. Context has different meanings in different areas of research and it is important to define what it means for recommender systems, and more particularly for the recommendation problem at hand. Examples of contexts include temporal, such as time of the day or season of the year; location, if the user is at work, at home, or on holiday; platform, is the user using a smartphone or desktop computer; mood, is the user happy, sad or sleepy. Each of these may be combined, making the problem of adjusting recommendations to context multi-dimensional. Needless to say some contextual information is hard to collect and the ubiquity of data collection also impacts the usefulness of a recommendation engine.

Recommendation models using context are typically classified in one of three categories depending on at which point the contextual dimension is included in the com-
putation: pre-filtering, post-filtering, and contextual modeling. In the first case, data that will form the prediction is selected according to the contextual variable. In the second case the contextual information is initially ignored and only applied to filter the resulting recommendation set. Finally, contextual information can be used as part of the prediction [26, Ch. 7, p. 232].

One example of contextual modeling is based on regression models, and to date, the so called Tensor Factorisation (TF) model is considered the most accurate. It is essentially an extension of the classical two-dimensional matrix factorisation technique to a n-dimensional one. The drawback with this technique, however, is that it introduces exponentially many parameters depending on the number of contextual factors. While it does increase accuracy for large training sets, its performance gains are hard to justify. In these cases, [7] argues that a simpler model may perform almost as well or even better under some conditions. Simpler models are required if we want to make use of context online as computation time is a scarce resource in those circumstances.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recommender purpose</td>
<td>To help users find good items which will increase the interaction making the users spend more time with the site. Especially we want people to (re)connect with the long-tail. The recommendations are an additional way to discover content alongside other features such as ”most viewed” and ”newest videos”.</td>
</tr>
<tr>
<td>Recommender type</td>
<td>Multiple items, a list of recommended videos to watch next.</td>
</tr>
<tr>
<td>Integration with navigation</td>
<td>Embedded with the video player such that the items are provided when a video is paused or a completed.</td>
</tr>
<tr>
<td>Performance criteria</td>
<td>Recommendations should keep the user engaged with the site.</td>
</tr>
<tr>
<td>Device support</td>
<td>Any desktop browser initially, may be relevant for mobile browsers or clients in the future.</td>
</tr>
<tr>
<td>Number of users</td>
<td>13 millions and more.</td>
</tr>
<tr>
<td>Application infrastructure</td>
<td>Browser-based, recommendations are requested when video starts playing.</td>
</tr>
<tr>
<td>Screen real-estate</td>
<td>Limited to video player, and in future possibly also a video portal.</td>
</tr>
</tbody>
</table>

Table 1: Describing the features expected by the recommender for the Tuenti use-case.

3 A real-time recommendation model

3.1 Understanding the domain

Before one defines an algorithm it is important to understand exactly what the purpose of the recommender system is. [26, Ch. 10] presents three templates that are useful for analysing the application, the users, and the data which the recommender system will analyse and serve. Below these templates are provided with respect to Tuenti. Outlining this provides us with an understanding of what users of the recommendation system is expecting and also helps outline the requirements before proposing an architecture.

The input is based on how Tuenti’s recommender system works today and the product requirements that have been developed by product managers and engineers at Tuenti.

3.1.1 The application

With the template in Table 1 we are positioning the recommender application as a component in the Tuenti ecosystem. The application template helps us to narrow down the use-cases of the recommender by specifying the goals that we are trying to achieve.

3.1.2 The users

Any software project begins, or should begin, with the user in mind. What are their needs and wants? Who are they? Why do they want to use feature X? And so on. The
Table 2 helps us understand who the Tuenti users are and for whom we are providing recommendations.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demographic information</td>
<td>Yes. It is available and may be used during the offline model processing.</td>
</tr>
<tr>
<td>Goal’s existence</td>
<td>Eventually we strive to let users engage with content they enjoy. The product goal, which is clearly distinct from the user’s perceived goal, is to have users spend more time on the site and interact with more content as that has shown to increase revenue over time.</td>
</tr>
<tr>
<td>Goal’s nature</td>
<td>Implicit since application may infer preferences based on the interaction from the user.</td>
</tr>
<tr>
<td>Level of expectation</td>
<td>We expect users to have a rather intrinsic motivation for using the recommendation system, such as ”just browsing” [14]. Therefore, a user’s expectation of the recommendations are low.</td>
</tr>
<tr>
<td>Change of expectation over time</td>
<td>Yes. As users engage with content it is important to regularly update their preferences as they will shift over time as new content arrives.</td>
</tr>
<tr>
<td>Importance of user situation</td>
<td>Understanding what situation the users are in when interacting with the recommendation system is incredibly hard. We do not aim at providing recommendations for a particular situation. Nevertheless, adapting to contextual variables such as location, device used, or time of day should be possible.</td>
</tr>
<tr>
<td>Social environment</td>
<td>Users are seen as browsing the content alone. In other words, most commonly not with other people.</td>
</tr>
</tbody>
</table>

Table 2: Describing the characteristics of the Tuenti users with respect to recommendations.

A note on the expectations; contrasting our recommendation engine with for example that of Amazon, where enabling the users to buy more products is the goal, the expectations are low. In the case of Amazon, if a relevant product to the user is not within the lets say 10 first items, this implies a potential loss for Amazon. Similar may hold for Tuenti, but the fact that a video is not within the 10 first items does not necessarily correlate to a direct loss of income. Since we are targetting the “just browsing” users they are more willing to actually scroll through to the next 10 recommendations, or click on one of the 10, even though they might not fit perfectly, and only use it as a step to discover 10 more new recommendations. This also emphasize the need to adapt for context where the most recently engaged videos can help form the future recommendations.

Another important aspect to keep in mind is that when a user watching a set of videos, they are more likely from the same genre. Someone who begins watching music videos are likely to continue with that. Contrast this to listening to music, where
### Feature | Value
--- | ---
Data type | Semi-structured. The metadata available for video items include for example the length of the video, category it belongs to, a title, and the user who uploaded it.
Quality of metadata | Good, with the exception of the title and the optional description which are entered by the user. For example, a word in the title may be misspelled reducing the quality of those metadata fields.
Description based on standards | No.
Volume and diversity of items | 40+ million videos in a variety of genres with a fraction of professional videos (e.g music videos and tv shows) provided by media publishers. Videos may be linked from YouTube or uploaded directly by the users themselves.
Distribution of items | See Figure 1
Stability and persistence of items | Thousands of items are added daily and most videos receive a majority of their views within the first two days. Naturally, popular content accumulate more views over time while not so popular content quickly lose pace. See Figure 2
User ratings | Implicit and binary, it is possible to also account for the percentage viewed of a video.

Table 3: Describing the properties of the data, the items, and the interactions between the users and the items.

switching between genres are more common. You may easily switch from rock to indie, but not necessarily from a music video to a video with a funny cat.

### 3.1.3 The data

Each recommender system must account for the data that is available about the items, users and interactions. An overview of this is presented in Table 3. It is also important to consider the rate at which new items are added, what the long-tail looks like, and when popular content normally reaches its peak. The latter tells us how many items of all available are important to the users and for how long.

Figure 1 shows the long-tail distribution. Around 10% of the views are generated by unpopular or obscure videos, and the most views are generated by the very popular videos. This differs from the behaviour and usage pattern seen in the music industry where small bands and obscure genres may generate as much as half of the interactions of the long-tail. This fact could be used to restrict the item catalogue if needed, for example, by removing the videos which have only received a few views from the possible recommendations.
3.2 The case for real-time

Generating recommendations at real-time for web applications means requests have to be answered within a relatively short time, normally no longer than 200 ms and the vast majority much below that. Studies suggest that requests taking longer than 100 ms is noticeable for the end-user and may lead to a notion of the application being slow or unresponsive [23, Ch 5]. Google, for example, found that increasing the search results by as little as 10 ms made users less likely to use their search service. It is also important to note that the 200 ms is not a hard real-time requirement for recommendations, and if the certain timeouts are crossed it is simpler and acceptable to return default values.

In order to achieve response times much less than 200 ms for the majority of requests it is unfeasible to compute the most relevant items from the entire dataset online. Irrespective of the technique used, millions of items cannot be processed reliably fast. There are ways to approach this. Most techniques as we have seen have focused on minimizing the dataset or improving the algorithms [4], or as in more recent developments, attempted to use stream processing [9, 12, 24].

Matrix factorisation techniques, as demonstrated by the success in the Netflix Prize competition4, yields significant improvements of the accuracy of the recommendations. The challenge with applying a matrix factorisation model for online computation is accounting for user context. Models based on Tensor factorisation are computationally heavy and includes the contextual dimensions during model generation. Assuming that the model is regenerated nightly, the users’s context would be assigned historical data only. For example, what video had the user watched previous to one she clicked on just now; or, at what time did the user view these types of videos compared to other types. Instead, we strive to enable some adaptation to context due to the nature of how users engage with video. This allows the recommendations to use the most recent context such as the last video item watched or the current time.

3.3 Approach

Overcoming context and dataset size requires two things. First, the model cannot be completed with all contextual information during the offline model generation phase. If it were then only old context would be incorporated. And second, we need to store an appropriate intermediate representation of the result of the model such that we can easily complete it in run-time together with the contextual dimensions. Fortunately matrix factorisation techniques provides a natural intermediate - the two matrices that contain the latent vectors for each user and item. These two matrices, called projections, and contain the user, or item, and a fixed number of hidden factors respectively. Combining the two using the inner-dot product gives us the estimated ratings for all users and all items.

Furthermore, the notion of storing the intermediate matrices require splitting the recommendation engine in two parts, much similar to the recommenders we have previously described in section 2.3.1. During the offline phase two datasets are created, one containing the user preferences and the other item characteristics. The items are thereafter grouped into something called itemsets. Preferably the itemsets contain an equal number of items, but they should be no larger than what the machine is capable of process online. In the second part, online, recommendations are put together, ranked, and served to the users.

While essentially any matrix factorisation model is applicable to this approach, we shall use the model explained in [15] as an example for this report. Any two-dimensional matrix factorisation model can be used as long as the projections can be stored to disk. The actual model proposed for Tuenti is a combination of several techniques and constitutes its secret sauce to recommendations.

Avid readers will have noted that the model described in [15] is not explicitly designed for contextual data. Even if the offline model does not account for context it does not mean we can use it to guide the recommendations. Context can be used to
guide the choice of itemset from which items will be recommended. The underlying model and, to some extent, the approach we are taking are agnostic to which way context is included. Be it in both the underlying model and for finding the most relevant itemset.

3.4 Generating partial ratings offline

During the offline processing two steps have to be performed, factorisation and clustering of items. For the sake of clarity we shall refer to a cluster of items as an itemset.

3.4.1 Storing the projections

Computing the projections from the sparse interaction data is a costly endeavour. As seen in example Algorithm 1, matrix factorisation continues until the error between the estimations and the original known values is small enough. Evidently the algorithm can be stopped earlier according to some fixed number of iterations if need be and possibly to avoid over-shooting the error. The exact computational and space costs depends on which algorithm for matrix factorisation that is used.

The last step of matrix factorisation is performing the inner dot product of two large and dense matrices. However, since we are not interested in getting the final rankings offline, in order to integrate context with the factors, this step can be avoided. Instead the last step to perform when the error is small enough is to simply write the two projections to stable storage such that our online system can retrieve them later.

When a request for recommendations for a particular user is made, the dot product for only that user is performed. Hence, in total we only compute recommendations for those users that are really active and request them.

3.4.2 Clustering the items

Clustering is a tricky topic as it depends heavily on the properties of the data being clustered. One of the properties that we have to ensure when creating the itemsets is that they are of appropriate sizes for online computation. This means that an itemset should not be bigger than the number of items that we can process within the time constraints given. This is in fact one of the primary reasons for clustering. Only by limiting, or parallelising, the computations of ratings can we serve requests in a timely manner.

Additionally, each of the itemsets have to be described by a vector. A so called centroid. The centroid is used to determine which itemset to use when a request arrives.

Working out the exact details of the clustering function is a thesis in itself. Therefore, for experimental reasons, we simply base clustering on the item factors of the resulting matrix factorisation. Irrespective of which variables are used to form the clustering of the items, it is essential that the itemsets contain the hidden factors generated by the model as these are used to determine the ratings.

3.5 Complete recommendation online

Intuitively, when a request for recommendation arrives we determine which itemset is most relevant for that request. We call this request routing. The factorisation model infers the ratings of items that are not yet known and assuming that this process is correct, the online challenge becomes traversing the millions of items and filtering out
the top most ones. That, as we have already concluded, is not realistic within the timeframe provided, hence, grouping items into itemsets.

3.5.1 Finding the most relevant itemset

The simplest approach to determining the most relevant itemset is by using the user’s preferences to guide the selection. Using a similarity function, we can compute the relative distance between each itemset’s centroid and user’s preference. There are a number of common ways of calculating similarity in a Euclidean space, including Jaccard similarity, Pearson similarity, and cosine similarity. Each function has its merits and respective shortcomings. Cosine similarity is used for two reasons: the calculation is cheap and the dimensionality is small. We are only dealing with at most 50, normally 10-20, latent factors.

Assuming that we can use user preferences to guide the selection of the itemset then we replace the context with the preferences. This is a bit contrary to the initial approach, but it will enable us to evaluate the online prototype against an offline implementation using the same model.

Contextual variables that can be used is is not within the scope of this report to address, instead refer to [26, Ch. 7] for an overview of the subject. For simplicity, however, the easiest may be to use the last video watched as context.

3.5.2 Retrieving top-K recommendations

Determining the actual recommendations is straightforward once the itemset is located. By calculating the dot-product between the user’s preferences (the user latent factors) and each item’s characteristics (the item latent factors) we get the predicted rating for the respective items. In one pass, using a priority queue, we can determine the top-K most relevant items and return these to the user. The cost of computing the recommendations on an itemset is linearly proportional to the number of items it contains. Here, the importance of appropriately sized clusters therefore becomes apparent.

In the next section (4) we shall outline the requirements and the architecture of a system that supports the approach described above. First, however, we shall explain how such system remains independent of the underlying model.

3.6 Model applicability

A recommendation model is content-agnostic if it depends solely on the interactions between users and items and not the content of the items. It is important that an online system component does not enforce any restrictions making the model content-dependent.

The major dependency of the online component is the two latent matrices, the user- and item latent factors. Inevitably this constrains the algorithms to generate the model to matrix factorisation techniques. The algorithm used to generate the model should not depend on any data that is only available at runtime since the offline component is separate from the online one.

Finally, since several models can be used to support the online system, the online component is content-agnostic. As long as the underlying model fulfills the requirements outlined earlier, primarily creating two projections of which one can be clustered, then the approach is also model-agnostic. This implies that the online system
is not specifically built for video content although that is the primary example used throughout this report.
4 Software design

In this section we will provide an overview of the recommendation engine’s architecture and what requirements that drove the proposed design. The requirements are based on the scenario outlined in section 3.1.

4.1 Requirements

The system should:

- **Generate N number of recommendations** based on users’ preferences.
- **Always return some N recommendations**.
- **Be able to account for users’ context**. First and foremost the last video watched.
- **Be able to support a multitude of content sources**. Content sources other than video include photos, albums, games and ads.
- **Handle at least 500 requests per second at peak times**, which is taken from the current peak load.
- **Handle popular content**, meaning some content is likely to be recommended much more due to popularity.
- **Be developed as a self-contained component** and expose a simple REST interface.

It is important to note that while availability is an important aspect to deal with when building large-scale systems, it is not paramount for recommendations. We shall see in the following sections how load-balancing and replication is used to handle load, but not strictly for providing higher availability although these methods certainly also improves it.

4.2 Architecture

In order to capture the entirety of the recommendation engine we employ a software architecture template first proposed by [19]. He proposed to capture the architecture in four different views, complemented by one or more optional scenarios. The default views suggested, and used here, are: logical, focusing on the system component various responsibilities; process, illustrating the typical program execution flow; development, commonly illustrating the separation of concerns of code entities; and lastly, physical which outlines the actual separation between users and system components. The physical view, in our case, is used to illustrate how the system may scale.

4.2.1 Logical view

To put the recommendation engine in perspective Figure 3 shows the separation between the offline and the online components. Nightly generated and clustered projections are stored in a stable storage. Input data involved in the model generation first has to be normalised. This includes, for example, extracting the relevant items from the log files. For example, each item’s type such as sport, comedy, or music video, must be converted to a numerical value. The user preference factors are stored in a
Figure 3: Illustrates the data sources and necessary offline steps before the online components load the resulting itemsets.

The database such that they can be retrieved and attached to the recommendation request. The centroids, also called itemset signatures, and the itemsets are stored as plain text files. Each line of the text files represents an item and each line is composed of the hidden factors of the projection.

When the offline computation is completed, the online component is signalled such that it reloads the new model. This signal marks the boundary between the two components.

The online component consists of a configurable number of nodes which reads and stores the item preferences in memory and serves recommendations from these. Its responsibility is solely to serve recommendations by retrieving the top-N items that match a user’s context and preferences. A request may be served by any available node. The node receiving the request will be responsible for ensuring that a reply to the frontend is sent. However, in the course of doing so it may ask one or more nodes in the system to compute the top-N items. In case the node cannot generate a recommendation set within a specified timeout a default set of pre-computed recommendations will be returned. These are provided auxiliary to the process described above for the itemsets. For example, a default recommendation set can be the most viewed or most recently added videos.

4.2.2 Process view

Not all pages on Tuenti would trigger a request for recommendations. One example is the video portal and when a user finishes watching a video. When a user browses one of these pages the a request is sent to one of many frontend servers (1) which will gather
and generate the necessary data to render the site. As seen in Figure 4 the frontend will format a request for recommendations such that it includes the user’s current context as well as the user’s preferences. The preferences will be retrieved from a database (2). At the online component, the only one with which the frontend interacts (3), the request is routed (4) to the itemset which is most relevant for the user and its current context. Using this itemset the top K items are calculated (5). Finally, the request is completed by marshalling the list of recommendations as a JSON structure (6). The frontend may use the JSON reply as it wishes, for example via a callback (7).

Although the online system has been designed to sit as a service behind a set of front-end servers, this is not strictly necessary due to its separation of the interface as outlined in the Physical view (4.2.4). Nevertheless, it is assumed that all requests arriving to a recommendation node are well-formed and well-behaved.

Figure 4: The user request is processed by one node in the online component. The request contains user’s preferences and possibly context. Once the recommendations have been computed they are sent as a JSON object.

4.2.3 Development view

Each node stores available itemsets in memory. Irrespective of which node that hosts the itemset the incoming request will be forwarded to the most relevant. It means that if the request arrives at node A and the most relevant itemset resides at node B, node A will forward the request to node B. This minimises the need for data transfer between the nodes as requests are many times smaller than the itemset that recommendations are computed from. Needless to say, machines with larger memory or, for that matter, a smaller item catalogue may fit all in memory. If this is the case the node will use
round-robin to balance load.

To handle popular content, itemsets can be replicated across multiple nodes. For example, with a configuration setting of three replicas, each itemset will be loaded three times spread across the available nodes. More details about the loading of itemsets will be covered in section 5.5.

It is also worth mentioning that all nodes have a strictly local view of the itemsets available between the nodes. The registry over the available itemsets are local to each node. This means that the registry maintained at each node may at times differ from each other (Figure 5). However, given that all failures are eventually detected, the registries will also be eventually consistent across the nodes. This will be further elaborated upon in section 5.7. Each itemset is managed by a so called worker. A worker may never handle more than one itemset, but several workers can manage duplicates of the same itemset.

Communication between nodes is managed through message passing. Hence, the references to workers maintained in the registry are merely addresses to specific workers. The workers’s location are, however, transparent to the registry. In other words, the registry does not know whether the worker is local or not.

![Diagram showing the internal components of the recommendation engine and how they relate to one another. Note that workers may reside both locally and remotely.](image)

### 4.2.4 Physical view

The recommendation engine depends on three major components: a specialised interface, recommendation node(s), and a nameserver (Figure 6). None of these are required to run on the same machine, however, it is assumed that they are located within a secured network. The only potentially public facing component is the interface with which other services communicate to request recommendations.

Each recommendation node depends on a configuration file which is used to bootstrap the node before being able to serve recommendations. To maintain group membership a nameserver provides a simple look-up facility. It is could be provided by Zookeeper which is both highly-available and consistent. Nevertheless, for our experimental purposes the nameserver was implemented as a simple application without stronger guarantees. Communication between the recommendation nodes and
the nameserver is not on the critical path for recommendation requests, and hence, response times are allowed to be significantly longer than between recommendation nodes. The nameserver is only contacted periodically and for managerial tasks.

Finally, the interface is separated from the internal interface for two reasons: first, a separate interface makes it easier to add several implementations. For example, the proposed solution uses a HTTP-REST interface, but one could easily build an interface which supports SOAP or XMPP, or implements a tighter security and sanity check on the incoming requests. Moving on, the second reason for decoupling the interface is to make each individual component horizontally scalable. Much in the same way as recommendation nodes can be added to distribute load, multiple interfaces can be added to support higher number of requests. From a code perspective, separating the recommendation node and its external interface keeps responsibilities clear.
5 Implementation details

5.1 Introducing the toolbox

The following sections will make occasional references to tools used to build the prototype. For some aspects, especially while describing the failure scenarios, it is important to be aware of the libraries’ underlying properties.

- **Scala** is a relatively modern programming language which runs on the Java Virtual Machine (JVM). It combines both functional and object-oriented properties. The majority of the prototype is implemented in Scala.

- **Akka** is an actor framework for Java and Scala. Actors are independent processes that can only communicate with one another using message passing. Akka does not guarantee message delivery between actors, however, it does provide useful abstractions for detecting and handling actor and node failures. Furthermore, all failures within an actor are self-contained and if an actor fails, for any reason, it will be restarted according to a supervisor strategy specified by the actor who started it.

- **jBlas** is the math library used for performing the matrix operations. It contains optimised data structures and algorithms which performs incredibly well on the JVM. The performance of jBlas and the isolation of workers that Akka provides were two main reasons why this technology was chosen.

- **Scipy** is a Python library geared towards scientific usage. It provides an implementation the k-means clustering algorithm used to group the items into itemsets.

5.2 Creating clusters of items

In order to get the online recommendation engine operational a number of steps are required. One of them after the partial matrix factorisation model has been generated and stored in persistent storage is to create the clusters of items.

Clusters are formed by applying the k-means algorithm as provided by Scipy. While the Scipy implementation does not scale easily to millions of items, it is sufficient for the sample set used.

5.2.1 Creating balanced clusters

The k-means algorithm does not guarantee that clusters formed will be of equal sizes. This is almost entirely dependent on the data that is being clustered. The resulting matrices of the model, the item and user factors respectively, has not been designed to be clustered. It is therefore highly likely that many items end up belonging to the same itemset causing an uneven distribution.

There may be other clustering algorithms that are more suitable for the data we are trying to cluster. Investigating which one provides the optimal results has not been the focus of this report, and despite the possible short-comings of k-means we have continued to use this algorithm for this dataset.

For samples using synthetic data this is less of an issue. The approach used to generate synthetic data was to first create all the item vectors and afterwards apply clustering to them to determine their centroids. For item catalogues up to one million
items with 1000 itemsets this approach yielded almost perfectly even clusters. The synthetic datasets were used to test the performance of the recommendation engine. While exactly balanced itemsets are not a requirement it helps maintain a balanced load on the system as well as better recommendation quality. For example, lets say a request is routed to an itemset with few items. The items are reranked and the top-K ones are sent to the user. However, since the itemset is small only few relevant items in that itemset exists. In the end the user may perceive the recommendation engine as performing badly and stop interacting with the results completely. We provide a further explanation and discussion about this in section 7.1.2.

5.2.2 Advantages of clustering

By using itemsets we restrict the number of items for which recommendations have to be generated. This reduces coverage [26, Ch. 8] which will impact the result of the recommendations given. However, there is also a strong computational advantage of splitting items into itemsets: it makes it possible to compute recommendations from many itemsets in parallel. Instead of determining the most relevant itemset during the routing phase we can compute recommendations from the top-N most relevant itemsets. The results, collected asynchronously within a given timelimit, merged and eventually the top-K most relevant items are returned to the user. The merge function could be as simple as concatenating the results from each cluster, sorting the items, and selecting the first K elements. Optionally, one may introduce a merge function that accounts for certain business rules.

Beyond parallelisation, recommender systems exploiting a collaborative filtering model suffers from something called the cold-start problem. New items can form a itemset on its own, and if needed, we can mix recommendations from this itemset with other recommendations according to some probability. On the other hand, new users are trickier as they do not have any interaction history. Hypothetically it would be possible to only use the user’s context to select the itemset that is most relevant. Once the model is updated the user’s interaction history will be incorporated in the user preferences. Using only context, however, will be less accurate than those integrating the user’s preferences.

5.3 Determining the most relevant itemset

As explained in Section 3.5 routing to the most relevant itemset is done using distance function. This Euclidean distance is computed using cosine similarity. It is a simple and fast metric to implement and is regarded as one of the more accurate ones according to a survey examining several different distance functions applied to social networks [29].

\[
\text{itemset relevance} = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}
\]

(3)

When a recommendation request arrives the first thing to do is to determine which itemset is most suitable to handle the request. Most suitable in this case is the itemset with the centroid closest to the incoming request. Both being described as vectors. Naturally the similarity has to be calculated for all clusters and for every request.

Finding the most relevant cluster has linear complexity to the number of clusters. The calculation is done for all signatures with respect to the provided context and cannot easily be cached.
5.4 Client API

The client API is simple and straightforward. It is uses REST over HTTP and all responses are structured using JSON. A recommendation request takes two arguments: the context vector and the number of recommendations to generate. The second can be used to minimise the number of requests made to the recommendation cluster. See below for an example.

Listing 1: Sample request and JSON response

```
http://a.cool.url/recos?context=[vars]&quantity=N

{
  "recommendations" : 
  [
    {
      "id" : 100,
      "rating" : 2.5
    },
    {
      "id" : 150,
      "rating" : 2.0
    }
  ],
  "generated" : <timestamp>,
  "timetogenerate" : <time_in_millis>
}
```

5.5 Bootstrapping the system

When a node of the recommendation engine starts it registers with the group membership service. For this report the service is implemented as a simple nameserver accepting three commands:

- **join** - adds a node hostname and ip to a list
- **part** - removes a node
- **get** - responds with a list of all currently connected nodes

It is, however, easily replaceable by a more fault-tolerant alternative such as Zookeeper\(^5\) and Dozzerd\(^6\), or implemented using jGroups\(^7\).

As a confirmation on a node’s registration the nameserver replies with a list containing all other nodes that are also registered, including the node which is performing the registration. From here on there are two different paths. If the node is first to join the cluster it will become responsible for populating the nodes with data. Hence, once the second node joins it will before alleviating load from the first node, ask if there is data that still hasn’t been loaded and prioritise that. The number of itemsets a node can load depends on its memory capacity. If there are no new itemsets to load when a node joins it will automatically alleviate load from the existing nodes. Any node in the cluster can serve recommendation requests as its routing mechanism will ensure each request is processed by a node with the most relevant itemset.

Let’s look at the each of the two join scenarios separately.

\(^5\)http://zookeeper.apache.org/, last accessed 2012-05-09
\(^6\)https://github.com/ha/doozerd, last accessed 2012-05-09
\(^7\)http://www.jgroups.org/overview.html, last accessed 2012-05-09
5.5.1 First node join

The first node to join automatically becomes responsible for coordinating the itemset load procedure. From the nameserver’s reply the joining node can determine whether it is the first or not to join. The responsibility as the first node includes three parts:

1. **Reading signatures** - as described in Section 5.2 itemsets are stored in files. Each itemset is identified by a signature and these are stored in a separate file. The first node to join reads the signature file and assigns ids to each signature. This id is used internally by the routing mechanism to lookup itemsets. The id remains constant throughout the validity of the itemsets.

2. **Creating a registry** - using the signatures and their id’s the node creates a registry with all the itemsets that eventually will become available. At this point the registry is not able to route to any itemset but maintains a list of itemsets to expect. These can be initiated locally on the node or on a remote node.

3. **Loading itemsets** - all nodes will load as many itemsets as possible to memory. Each itemset is owned by a worker, and they are distinct from each other. A worker will never be responsible for more than one itemset, but there can be several workers with the same itemset. Failures are contained within a worker and does not propagate to the client requesting recommendations. It is possible to specify using a configuration value how many itemsets a node should be allowed to load. When a worker has been started with its respective itemset it signals all available nodes that it is available to handle incoming requests.

Assuming that the first node is not able to maintain all itemsets in memory on its own it will ensure that those not loaded are given priority when the next node joins. Even if all itemsets may not have been loaded with the first node, it is now possible to serve recommendation requests.

5.5.2 Consecutive node join

All nodes following the first will use a slightly different approach to bootstrapping. Consequently, when the second node registers with the nameserver it will determine that another node is responsible for ensuring that all itemsets are loaded in the cluster. The node will not read the signature file since the first node already have assigned id’s to all itemsets. Instead the node follows these three steps:

1. **Replicate registry** - to learn which itemsets are already loaded the node asks for a copy of all other nodes’s registries. If it receives more than one it will merge them, explained later, and afterwards instantiate its router. At this point the node is ready to route recommendation requests to the existing itemsets.

2. **Prioritise itemsets** - together with the copy of the registries the joining node receives a list from each node with the itemsets that it believes should be loaded.

   (a) If a node sends ids for itemsets that have not yet been loaded by the recommendation cluster the node will prioritise these before any other itemset. It will load as many as it can according to its memory capacity or setting. The remaining itemsets will be dealt with by the following node joining.
(b) If all itemsets are already loaded, the new node can alleviate load from existing nodes instead. Each node keep tracks of the number of requests processed by each worker. This number determines the priorities of which itemsets to duplicate. Therefore, after sorting the itemsets received by the other nodes it will load as many of them as possible.

3. **Load itemsets** - this procedure is exactly the same as the third step for the first node. It is important to point out that all workers starting on the node joining will register all other nodes. This ensures that all nodes are able to route to it given that no messages are lost. We shall discuss failures and different failure scenarios later.

The merge-procedure of registries is straightforward. A node always starts out with an empty registry. For any registry that it is asked to merge it will traverse all entries and add them to its own registry. If the entry does not exist in its own registry an entry is added and all the worker references for that itemset is copied. On the other hand, if an identical entry already exists it will copy only new worker references.

### 5.6 Popular itemsets

By natural distribution some itemsets will be more popular than others due to the items they contain. For example, a U2 video may be recommended more often compared to the less known Stockholm band Winston. An itemset containing many popular items risks suffering from contention. Meaning more requests are destined for the same itemset than one worker can process in time. As mentioned in during the bootstrapping procedure, many workers can be responsible for the same itemset. When a node routes the request to the particular itemset, it will select one of the available workers to handle the request. This is done by a simple round-robin algorithm which provides uniform distribution of load. Keeping track of which node receives the request, however, is strictly local to the node performing the routing. Hence, it remains to be explored whether a shared global view could improve load-balancing amongst popular itemsets further.

### 5.7 Failure scenarios

In this section we shall discuss the implications of specific system failures and how they are handled. At the end of the section an outline of failures not handled, but recognised is provided. The recommendation engine has not been designed to handle Byzantine behaviour explicitly although we are aware of them. A recommendation engine is not that mission critical component and therefore error handling has been designed to be contained within to individual recommendation requests as far as possible.

#### 5.7.1 Worker crashes

A single worker may crash independently and isolated from all other workers. For example, if the data becomes corrupt leading to an arithmetic exception only that request will fail. A failure does not impact any workers or components in the system. Each worker is automatically assigned a supervisor when it starts. The supervisor monitors the worker’s life-cycle and in case of a crash restarts the worker. Moreover, the worker is re-initialised with the same state, all pending requests delegated to that workers will continue to be handled. Moreover, the system is designed to handle failures in such a way that the impact of a single failure is minimal and the system remains robust.
be processed once the worker is restarted. However, the request that the worker was processing when it crashed will be lost unless it had sent the reply first.

As each node’s registry keeps references to workers these may have to be removed if the workers dies and does not come back. Supervisors ensure that a crashed worker reuses the same reference as before. However, to counter permanent worker disappearances the registry sets up watches for all workers it knows about. The watches are added when the worker registers with the nodes. If a termination signal is received from the failure detector the node removes the worker reference from the registry.

The drawback with the watch, which relies on a timeout, is that incoming requests may be routed to a dead worker during the timeout period. If no worker exists to replace it, it will not be possible to route requests to the respective itemset at all until a new one is added. This, inevitably, will result in failed recommendation requests and a default set of recommendations will be provided instead.

5.7.2 Node crashes

There are two scenarios under which the first node may crash: 1) during coordination of the bootstrap procedure, and 2) during normal operation, in other words, after the bootstrap is complete.

The first case is not handled and therefore if this happens the bootstrap procedure needs to restart from the beginning. This is due to the fact that the registry maintained by the node is not persistent and, hence, once ids to signatures are assigned it is not possible to recover from where it crashed. The registry is only kept available once more nodes have joined the cluster as it is then replicated across the nodes.

A node failure is seen by the other nodes as several workers shutting down at the same time. The watches will see termination signals coming from all workers that ran on the node crashing and remove them from the registry. It may happen that all nodes do not do this at the same time, however, the registry eventually becomes consistent on all nodes again. When a large number of workers fail at the same time performance may be negatively impacted.

One may rightly wonder what happens if all workers supplying a specific itemset die and do not recover. This may happen when a node fails and all the workers are lost. Unless the workers have been distributed across multiple nodes the recommendation engine will no longer be able to serve recommendation requests using that itemset.

Two things will happen: first the node determines that all available worker references are lost and marks the itemset as unloaded. If the node can host the itemset itself it will load it. If it cannot, the itemset will be loaded again when a new node joins the system.

5.7.3 Failures mitigated by timeouts

All recommendation requests have an upper bound on the time it may take to complete them. A request delegate, which is spawned to coordinate the request processing, also enforces a strict timeout returning default values in case it is met. While the timeout is configurable it is by default set to 500 ms. There may be a number of reasons for timeouts to occur such as Byzantine behaviour. Most commonly, as discovered during performance testing and further explained in section 6.3, it will be due to overloaded workers. In particular, a high number of requests dedicated to the same worker may cause the worker to queue up messages. Not being able to process them in time the delegate will ignore the worker’s potential future reply and instead reply with a default set of recommendations.
5.8 New users and items

Neither new items nor new users will be able to benefit from personalised recommendations until there is some interaction between a new item and existing user or new user and existing item. During the offline model regeneration the new user and item will be included and thus, the following day we can serve recommendations including those.

A possible extension to the architecture and prototype to being able to integrate new items faster may be to form a separate itemsets with only new items arriving during the day. Items from this itemset could be incorporated, merged, with the existing recommendations according to some rules. This solution would, however, not be beneficial for new users which depends on the interaction with items. Further work is required to determine how to handle new users until they have been incorporated by the model.
6 Results

6.1 Prototype

In order to evaluate the concept of splitting the matrix factorisation model in two components we developed a functional prototype. The prototype was built in Scala and the actor-framework Akka running on the JVM and using the jBlas library for all matrix computations. It was deployed and evaluated on three machines in a data center with high-bandwidth connections. The recommendation engine and the HTTP interface is packaged individually and a supplied configuration file to each enables customisation. In addition to the prototype, several scripts for parsing log files, clustering items, and performing performance and accuracy measures were also developed. The majority of these in Python. The implementation of the matrix factorisation model [15], implemented in Java, was kindly provided by Telefonica R&D in Barcelona, Spain.

<table>
<thead>
<tr>
<th>Language</th>
<th># Files</th>
<th># Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala</td>
<td>35</td>
<td>1708 (25% tests)</td>
</tr>
<tr>
<td>Python</td>
<td>9</td>
<td>532</td>
</tr>
</tbody>
</table>

Table 4: Lines of code excluding comments and whitespace.

The following evaluation shows that the prototype is able to serve the required number of recommendations as defined by the peek-traffic at Tuenti. At the point where the system got saturated possible areas for improvement have been discovered and are discussed in more detail in section 7.1.1.

Moreover, the accuracy, using a small subset of real data provided by Tuenti, shows that routing to the most relevant itemset is imperative to maintain good recommendation accuracy. Computing recommendations from multiple itemsets in parallel improves the accuracy, but determining the most relevant ones are still important.

The evaluation is composed of two types of tests: recommendation accuracy and performance tests. While performance is critical due to the large number of recommendation requests and data size, it is equally important to demonstrate how accuracy of the recommendations is affected with the proposed changes. Specifically with respect to the extra layer of re-direction introduced with the routing.

For scalability and performance we measure the prototype’s capacity with one to three nodes to obtain the maximum throughput and best latency during full load. Those tests are performed with synthetic data.

6.2 Accuracy of recommendations

Evaluating recommendations have been studied and discussed before in [14, 26]. For our purposes we have opted for Mean Average Precision (MAP), which gives a global estimation of the recommendations’s usefulness. It is most appropriate for data where binary ratings are used. That is, when the an item is not worth more or less than another, simply that it either is relevant or it is not [14]. Such is our case since either a video is watched, or it is not. There is no way of providing a gradual rating to it.

MAP is a measure of the model’s accuracy. Have two different models work through the same data and you can use MAP to compare which of them provides the better result. In our case, however, we are not interested in comparing two different models. Rather, we want to investigate how clustering of items affect the results of
Table 5: Measuring Mean Average Precision over a varying number of itemsets.

<table>
<thead>
<tr>
<th>Itemsets (#)</th>
<th>Queried (#)</th>
<th>Non-zero (#)</th>
<th>MAP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>196</td>
<td>74.49</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>41</td>
<td>23.71</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>87</td>
<td>24.99</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>116</td>
<td>36.46</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>165</td>
<td>58.08</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>196</td>
<td>74.49</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>29</td>
<td>37.82</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>58</td>
<td>34.99</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>74</td>
<td>34.60</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>110</td>
<td>44.67</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>123</td>
<td>37.34</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>147</td>
<td>51.10</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>163</td>
<td>58.02</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>176</td>
<td>64.83</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>196</td>
<td>74.49</td>
</tr>
</tbody>
</table>

Hypothetically, since the proposed recommendation engine splits the item catalogue in multiple itemsets the accuracy will decrease because all relevant items are not available. The MAP on the entire item catalogue will not be affected. Thus, if we can query the entire itemset we can establish a baseline measurement from which we can compare consecutive results using a part of the item catalogue. The quality of the recommendations will depend on how well we can locate the relevant itemsets. For example, let say the item catalogue is divided into three itemsets and three relevant items for user A is stored in the first itemset. Assume that as long as we can find the right itemset, all relevant items will be recommended on top. If the request is routed to the first itemset our MAP value will be 100% since all relevant items occupy the first three positions of the recommendation set. If we query the wrong itemset, the MAP value will be 0% as no relevant items are found. Equally, imagine now that the three items are distributed so that there is one relevant item in each itemset and it will come out on top in a recommendation request. In this case MAP will also be 100%, irrespective of which itemset we query. Naturally, all relevant items may not end up being the most
relevant according to our model. It is an approximation after all. Therefore MAP will normally be less, or much less, than 100%.

Essentially, when a request is processed by only one itemset the item coverage is reduced. The probability that the itemset contains relevant items for the particular user depends on the size and the appropriateness of the itemset. Increasing the number of itemsets used to compute recommendations from improves the probability of discovering relevant items. However, with more items available the rank of the relevant ones may be lower.

Using a small set of data from the Tuenti video play logs we measured how MAP varies with the number of itemsets queried for each request. Our results are presented in Table 5. The baseline MAP value for the entire sample item catalogue is 74.49%. As can be seen from querying one itemset, using five and ten itemsets respectively, MAP is significantly lower. In the ten itemset-case, MAP is in fact 0. This is due to the fact that the most relevant itemset does not contain any relevant items. Consequently, the ability to route correctly is imperative for recommendation quality.

Moreover, the measurements on ten itemsets illustrates how MAP can reduce with increased coverage. Between five and six itemsets there is a measured drop. As mentioned, this is due to the fact that more non-relevant items rank higher than the relevant ones. In general, however, the more itemsets we can cover in a computation the better becomes the quality of the recommendations.

A more appropriate model to generate the projections may yield a better baseline MAP value for the Tuenti dataset. Such study is beyond the scope of this report.

### 6.3 Scalability and Performance

Performance of the recommendation engine is primarily determined by two things: the router and the number of workers. Even though the router performs fast computations on a small set of data, every request have to pass through this component. It risks becoming a bottleneck although it can be alleviated if load-balancing between multiple nodes is performed. The workers computes recommendations on a larger set of data compared to the router and may become a bottleneck. This is the reason why they are easily replicated, such that computations for the same itemset can take place in parallel.

Performance was measured in three stages. First to get a baseline measurement only one node was used. The node had all itemsets loaded in memory. Secondly, configurations with up to three nodes were tested. In all cases could the nodes load all itemsets to memory. Requests were evenly distributed across the nodes due to the round-robin load-balancer. In the third case, more interfaces were added. To simulate a hardware load-balancer the load-testing tool was modified to issue requests evenly amongst the available nodes.

Jmeter, a tool to measure performance and load test system behaviour, was used in all tests. A plug-in in Java was developed to create randomized queries on the data. The load was generated from a fourth machine so as not to interfere with the recommendation engine.

#### 6.3.1 Single node

To determine the maximum throughput of the recommendation engine we designed a test where one node was put to under an increasing workload. Over time the number of requests per second was gradually increased until the point where the node began to
either 1) drop incoming requests, or 2) was not able to serve requests within the defined time constraints of 100 ms.

As Figure 7 shows how load is gradually increased from 0 to 1200 requests per second. As seen the throughput levels around 450 requests per second, and at the same time the latency levels at approximately 22 ms. The system is able to perform at a lower latency if throughput is lower. The optimal latency is achieved when less than 300 requests per second are issued.

As expected when running on the JVM, due to Just-in-Time compilation, there is an initial period when code optimisations occur. This period, albeit short, affects latency and throughput negatively as seen in Figure 8. Once most, or all, optimisations are in place the latency and throughput stabilises at a steady state which also is the maximum capacity for one node with a synthetic data with 100 itemsets of approximately 10000 items each. The test ran for five minutes.
Figure 7: Gradually increasing load tester from 0 to 1200 requests per second. The system stagnates at 450 requests per second.
Figure 8: Throughput and latency during the first five minutes with one node. After the ramp-up period throughput levels at 425 req/s (blue) and latency at 23ms.
6.3.2 Multiple nodes

When adding more nodes there are two things that we should be concerned about: increasing the availability of itemsets, and increasing or at least maintaining the throughput the nodes handle. Continuing from Figure 8 we thus add one and two more nodes as measured in Figure 9 and 10 respectively. With two nodes the throughput increases slightly, about 50 requests/second, and remains stable throughout the test. Latency remains at an average of 20 ms.

The sudden spike in latency and drop in throughput seen in both Figure 9 and 10 may be caused by garbage collection or other external OS process performing some short CPU intensive task. However, the most notable discovery from the two graphs are the performance drop when a third node is added.

The throughput and latency graphs does not yield the full story with respect to its distribution. To fully understand it we plot the latency distribution in Figure 11. The 95th percentile is 29 ms and the 99th percentile is 38 ms.
Figure 9: Two nodes with throughput around 500 req/s (blue) and latency (red) 20 ms. Note that latency is increased 10x to fit the graph.
Figure 10: Three nodes with throughput around 480 req/s (blue) and latency (red) 22 ms. Note that latency is increased 10x to fit the graph.
Figure 11: Latency distribution with the 95th percentile below 28 ms and 99th percentile below 38 ms.
6.3.3 Scaling the interface

The system was designed to be able to scale components individually and the previous tests used only one HTTP interface. In the following tests we add recommendation nodes each with its own HTTP interface. This will reduce the load on the router of each node as the HTTP interface by default binds to the local node’s router.

In Figure 12 we use one HTTP interface for each recommendation node. The graph shows that throughput increases and latency decreases compared to tests with only one HTTP interface. The improvement is attributed to the fact that the available recommendation routers are better utilised. Hence, this results show that the router is in fact becoming a bottleneck that limits the scalability of the system. This is despite of the computation time for one request being as low as half a millisecond.

Figure 13 adds yet another node and interface. With three nodes and three HTTP interfaces we see that throughput increases to roughly 700 req/s. However, as also clearly seen both latency and throughput have significant spikes and drops. Correlated with the logs we find that at the times of a latency spike we have a number of timeouts happening. This is related to the requests piling up in the router’s queues and not processed fast enough. We measured the queue time for messages and found that messages could stay in the queue for more than 500 ms before being processed by the worker. Even if the worker computes the recommendations the delegate at the interface waiting for the result would have timed out and returned a default set of values (and hence count as failed request causing the drop in throughput). Essentially we have reached the maximum sustained throughput, and while it does improve with each node added, the improvements are relatively small. Giving some headroom to avoid the majority of timeouts we can conclude that with three nodes a maximum sustained throughput is 600 req/s.
Figure 12: Two nodes with individual HTTP interfaces. Throughput around 550 req/s (blue) and latency (red) around 17 ms. Note that latency is increased 10x to fit the graph.
Figure 13: Three nodes with individual HTTP interfaces. Throughput around 700 req/s (blue) and latency (red) around 15 ms. The spike in latency is attributed to requests being queued up at the router.
7 Discussion

7.1 Main findings

7.1.1 Scalability

From the performance results on throughput and latency we know that the system safely handles 600 requests per second with the 95th percentile latency at a 28 ms using three nodes. The results surpass the performance requirements specified in section 4.1, but performance could be improved further. Especially we found that the router on each node more easily than thought also becomes a potential bottleneck. It is in other words not enough to replicate workers. While that resolves one problem, balancing requests between the three nodes does not provide significant improvements to the number of requests handled per second.

In this section we discuss some alternatives to the current setup. The findings are based on the scalability experiments presented in section 6.3 as well as discoveries made during the prototype implementation phase.

- **Pool of routers:** Despite not initially being considered a bottleneck, the router shown to cause significant overhead as the number of requests increased beyond its capacity. Similar to the worker queues, this lead to increased queue times for the messages and eventually causing several requests to timeout. Better parallelising the requests to the router, for example by instantiating a pool of actors which can route the requests and load-balancing between them, queue times can be reduced. This would, however, require that more references to workers are kept up to date as actors do not share any memory. A possible approach for tackling this would be to serve the registry form a separate actor and have the routers cache a version of it. In case a request fails it can invalidate its entry and perform a new lookup next time. It is likely that this approach would impact latency negatively due to the increased overhead of lookups and load-balancing between routers, but that may be acceptable if scalability improves.

- **No remote routing:** Instead of letting all routers have references to all workers, only maintain references to workers on the same node. Requests that are destined for an itemset only available at a remote node would have to perform a two-stage lookup instead of the current direct reference lookup. Hence, a router which wants to route a request to a worker located at a remote node would look up which node is responsible for the targeted itemset and forward the request to that remote node. At the remote node, the node’s router would perform a local lookup and forwarding the request to the responsible worker. This would minimise the number of worker references that a node needs to keep in memory.

- **Tree-like organisation:** If the number of clusters grow too large, it could be possible to organise the routing of requests in two or more hierarchies, much like a tree. The root node would perform a lookup amongst its children which itemset is most suitable to serve the request (using context and other variables). All intermediate nodes in the tree would also be routers, gradually moving towards the most suitable worker, which would be the leaf-nodes of the tree. This approach could also facilitate merges from, for example, a branch of workers before answering the request. Needless to say, the complexity of management increases with an approach like this, but it would be interesting to explore its advantages with respect to item coverage and scalability.
• **Automatic rebalancing:** With respect to rebalancing workers, when a worker is receiving more messages than it can handle and its queue is growing load can be handled by creating replicas of that worker. However, in the prototype’s current form this only happens when a new node joins. It would be beneficial to do this continuously such that when the load increases more workers are dynamically added or removed. Elastic load-balancing has been subject to study before and best-practices from this research can be incorporated here.

  Additionally, the prototype stores all itemsets in memory for one reason: computational performance. We could envisage a situation where the itemsets are stored in a key-value store and retrieved, possibly cached, when necessary instead of storing everything in memory all the time. However, there are good reasons not to. First, jBlas which is used for the matrix computations comes with optimised data structures. Implementing these, and the computations, are error prone and likely not as fast as those already supplemented with the library. Second, retrieving the itemsets, which are of significant sizes, would considerably worsen latency. Since each itemset contains thousands of items, each described by a float-vector, the amount of data needed to the transferred for every request outweighs the benefits of a third-party storage.

### 7.1.2 Recommendation quality

The assumption made in section 3.3, being able to group items into itemsets, requires a little bit of elaboration. It is assumed that itemsets can be formed in such a way that: one, they are of approximately equal sizes, and two, they are formed according to the needs of the user. Section 3.1 describes the requirements needed to fulfill quality recommendations for the Tuenti use-case. While this information is important when choosing or designing the model in itself, we stress the fact that this information is now equally important when creating the clusters. For instance, we know that people watching videos are likely to watch the same type of videos within the user’s session. Begin watching a sports video about car races and you are likely to look for a similar one next, perhaps from a different race. On the other hand, the pattern when listening to music is different. When recommending music, novelty is important. Recommending The Beatles over and over again may not score any points with the user. Therefore, the problem of creating itemsets must be taken into account when analysing the data and the use-case. The only requirement put forward by the suggested system is that itemsets are of approximately equal size. In other words, the system is independent to the clustering itself.

Nevertheless, by querying more itemsets per request we can increase the probability that more relevant items are discovered. This comes natural since the item coverage increases and more items are made available. Querying more itemsets, however, also has a performance penalty. The computations are done in parallel, but more resources are required to handle the requests, and latency increases as one has to await results from several itemsets.

### 7.1.3 Summary

Building a recommender is a multi-faceted challenge. It includes analysing data, defining a model, and constructing a system to support the model and the data. In this report we have focused on the latter: building a system capable of serving recommendations from a large item catalogue. We found that:
• Scaling to millions of items can be achieved by clustering them and querying only the most important itemset(s).

• Clustering is an important but orthogonal problem that should be addressed when defining the model.

• Quality of recommendations increases by querying more itemsets in parallel.

7.2 Positioning

7.2.1 Comparison to existing systems

Few existing systems that we have covered in section 2 uses matrix factorisation as their method for generating recommendations. [8] suggests that model-based methods tend to involve complexities which require great attention to details. More, matrix factorisation’s subjectivity to changes in variables requires careful tuning for the particular problem at hand. Perhaps because of these reasons, joint with the scalability challenges, matrix factorisation techniques have not made it into more large-scale recommenders. Given the recent developments in distributed computing frameworks more suited for iterative style computation, such as Mahout [3] and Graphlab [2], this is becoming less of a problem. Providing engineers with the tools necessary to implement these algorithms and using them with large datasets are paving the way for their usage in more recommender systems. With the developments, serving context-aware recommendations online is becoming easier and our prototype shows how it can be accomplished.

The prototype we have developed is not intended to replace systems like Mahout and Graphlab. These systems are solving an orthogonal problem, that of creating the recommendation model. However, these systems are not suitable for serving recommendations at large scale, it is within this area our prototype connects and resolves the missing link for getting the recommendations to the users. Our system is in fact dependent on frameworks like Mahout, or specifically built solutions, to create the model.

In contrast to [1], which provides a reference architecture for recommender system, our prototype was built for large-scale datasets serving millions of users. A centralised recommender and data collection point is in such cases are insufficient and distributed measures must be applied. An advantage of their proposed architecture is the ability to adapt as data is recorded. In other words, improving the recommendations over time as more and more log data becomes available.

The personalised Google News [10] might be the system most similar to our prototype in terms of concepts. Remember their system is solving a larger problem-set which includes that of collecting and updating metrics in real-time. This is necessary for the type of items, news, which they are recommending as it changes frequently. Their model, however, is based on entirely different techniques and consists of a combination of three algorithms. Part of the model is, similar to our concept, required to be computed offline. We are, nevertheless, attempting to use a, by previous research acknowledged, more accurate model.

Often the exact methods used in production recommender systems are not revealed and it is therefore hard to make any exact comparisons. However, we know that Netflix uses a recommendation method based on matrix factorisation as well [5]. And as the authors acknowledge, there is a need to adapt these techniques for production, especially adapting to new users and the scale of the data. We provide one approach to tackling the scale problem.
7.2.2 Our prototype vs stream processing

While the literature is full of traditional methods, some which have made it into production systems [10, 11, 20], there seem to be little research on using stream processing to address the same problem. Stream processing has not been applied to the recommendation research earlier. With the exception of [24] we know of no other production system using stream processing for recommendations.

There are a number of challenges with the split model in general, one which we are primarily trying to tackle in this report. Stream processing systems are designed to scale to large amounts of data. They operate on the premise that data is moved through the system, at which time a number of signals are registered. Scaling is inherent in their design. However, not all collaborative filtering methods can be adjusted to the stream processing paradigm and thus innovative systems to support these algorithms are required. Our prototype does this.

Using a model-based method often requires intensive offline processing. The model has to be recomputed regularly and the variables used to tune the model also evaluated periodically. Due to the periodicity the model will only ever deliver results based on the interactions it had available at the time of generation. Compare this to stream processing which, in minutes and sometime seconds, is able to account for recent interactions[9]. In [10] the authors show that it is possible to circumvent this problem by using algorithms which supports incremental updates. In addition, there is some work on incremental updates for matrix factorisation [25].

Lastly, but applicable to both models, is the issue of new users. When there exists no previous information about the user recommendations either has to come from default values, or it should use item-to-item recommendation. The latter is not dependent on knowing any previous information about the user. New items, however, has to be handled differently. New items which has no previous interactions recorded will never be recommended. To tackle this all new items may be featured on a separate list, such that users can find it separately, or they may be randomly mixed in with the recommendations in order to attract some initial interactions.

7.3 Future work

In this section we highlight some specific and general future directions based on the work conducted. In particular this section addresses those areas not previously discussed.

7.3.1 System specific improvements

In particular, this list highlights the primary short-comings of the current prototype before it would be production ready.

- **Replacing the nameserver with Zookeeper.** The current nameserver was intended as a short-cut to handle group membership of the recommendation nodes. A fault-tolerant alternative is required and consistent alternative is a precondition for deployment.

- **Read itemsets from distributed storage.** In its current form the nodes read itemsets from the local filesystem. As these are more likely to be stored in a DFS or database certain adaptations need to be made.
• **Scaling the router.** The router is a potential bottleneck under heavy load. While not an immediate requirement, revising the routing is necessary to scale to more requests.

• **Autonomic handling of popular itemsets.** As we have discussed previously the itemsets are only rebalanced when a new node joins. Using the same primitives we could implement a function to periodically re-evaluate the load of specific itemsets. Incorporating ideas from autonomic computing and self-managing systems would be beneficial [16].

• **Improving instrumentation.** With any system in production it is crucial to understand what is happening right now.

• **Long running tests.** The performance tests shown in section 6 are mostly short-lived. Longer-running tests are needed to see how the performance is affected over time.

This list is probably incomplete but provides some initial steps that is required before deploying the recommendation engine.

### 7.3.2 Recommendation freshness

As for research, there are several interesting directions to explore. Integrating online updates to the matrix factorisation model [25] would minimise, partly remove, the need for an offline regeneration. Since both the user and item sets are growing by the day it would add major improvements to the readiness of the recommendations. This work, while specific, is not trivial. As the model is specific to a particular domain [26, Ch. 10] the online component becomes increasingly coupled with it to provide the incremental updates. It is unclear how such approach, if doable, would affect the two matrices with latent factors that the proposed system depends on. However, the result of being able to update the projections in real-time with new interactions would greatly improve the freshness of the recommendations.

### 7.3.3 Research directions for recommender systems

In the larger picture, and from an academic viewpoint, the recommendation research community would benefit from an updated survey on recommender algorithms and systems. One of the better surveys [4] dates back to 2005 and while it covers a lot of the basics, the recommender’s area has changed a lot since. In 2009 a survey on collaborative filtering [30] was published. This survey too focuses on the algorithms. As a complement to these studies it would be beneficial to perform an analysis of what techniques have progressed to production systems. Perhaps more importantly, covering and outlining the necessary developments for scaling the proposed algorithms to large-scale datasets is needed. Knowing which methods work and which ones do not would provide a clear direction as to where future research efforts should be spent. Research in recommenders have for a long time been focused on the algorithms, it is time to for systems researchers to pick up this work and build upon it such that it scales to our current data requirements.
8 Conclusions

Recommender systems are becoming an important component in installations where the amount of data exceed that of which a user can browse through with ease. Highlighting and personalising content to the user leads to increased user-satisfaction and a greater user retention at the service. However, due to the scale of the data in places like Tuenti, making sense of the information and using it in the right context is complex. A variety of algorithms have been proposed to address the problem of recommending relevant content. While most of the previous research has focused on the algorithms themselves, little exploration on how to use these algorithms at large scale has been provided.

In this study we proposed an approach to online collaborative filtering using one of the most accurate methods available: matrix factorisation. By capturing intermediate components of the matrix factorisation process we split the recommendation procedure in two parts. One offline component which does the most computationally heavy part, and one online part completes the recommendation and serves users with suggestions. Since the item collection is in the ranges of millions we divide it into smaller itemsets using clustering. Recommendations are drawn from the itemset which is most relevant to the current user using a cosine-similarity function.

As such, our content-agnostic prototype can serve recommendations from millions of items online. Experimental results show that latency for the 95th percentile is below 28 ms with a maximum sustained throughput of approximately 600 requests per second. These results were taken from a three-node deployment. Evaluation of the quality of the recommendations was done by sampling real data from the Tuenti users’ activity logs. Quality depends on item coverage, which in turn depends on response times required, and that being able to determine which itemset is most relevant for the current user is crucial to maintain good system performance.

In conclusion, this report shows promising results to scaling recommendations to web-scale deployments using matrix factorisation. With personalised recommendations in real-time, interesting videos will soon be no further than a click away.
Acknowledgements

There are a lot of people who should be attributed a huge dose of gratitude for assisting me with this thesis. My office neighbour and math champion, the generally awesome Toni Cebrian, as well as his colleagues and recommender-experts Dr Linas Baltrunas and Dr Alexandros Karatzoglou from Telefonica R&D. My supervisor Professor José Luis Balcázar who has had to endure my rants and kept me from taking over the world by thoughtful guiding throughout this work. All my colleagues at Tuenti for their inspiration and geeky humor, but especially my mentor Albert Fernández Marsal, my team leader Einar Persson, and Tuenti’s lead engineer Tomasz Matuszczyk. Finally, a big thank you to all my EMDC classmates. The constant flow of amazing jokes and brilliant insights through the IRC channel always kept the ideas coming.

Thank you.
References


[31] L. Xiang. Hulu’s recommendation system. 