Vibes: A Platform-Centric Approach to Building Recommender Systems

Biswadeep Nag
Monetisation and Targeting Group
Strategic Data Solutions
Yahoo! Inc
biswadeepnag@yahoo.com

Abstract

Recommender systems have gained a lot of popularity as effective means of drawing repeat business, improving the navigability of web sites and generally in helping users and customers quickly locate items that are likely to be of interest. The rich literature of recommendation algorithms presents both opportunities and challenges. Clearly there are a wide variety of algorithmic tools available, but there are only a few that are suited for application to a broad variety of problem domains and even fewer that can scalably deal with very large data sets. In this paper we describe the architecture of the Vibes platform that is used to power recommendations across a wide range of Yahoo! properties including Shopping, Travel, Autos, Real Estate and Small Business.

The design principles of Vibes stress flexibility, re-usability, repeatability and scalability. The system can be broadly divided into the modeling component ("the brains"), the data processing component ("the torso") and the serving component ("the arms"). Vibes can accommodate a number of techniques including affinity based, attribute similarity based and collaborative filtering based models. The data processing component enables the aggregation of data from users' browse and purchase history logs after any required filtering and joining with other data sources such as categorizer outputs and unitized search terms. We are currently working on moving the modeling and data processing components to the Hadoop grid computing platform to enable Vibes to take advantage of even larger data sets. Finally the serving infrastructure uses REST based web services APIs to provide quick and easy integration with other Yahoo! properties. The whole Vibes platform is designed to make it easy to extend and deploy new recommendation models (in most cases without having to write any custom code). We illustrate this point by using a case study of how Vibes was used to build recommendation systems for Yahoo! Shopping.

1 Introduction

Research into recommendation systems goes back more than a decade with several important classes of algorithms proposed[1]. Lately they have achieved quite a bit of commercial success as well[5, 4], culminating in 2007 with the award of the first Netflix prize[7]. Though recommender systems clearly add value to the commercial proposition and user experience of a web site, they suffer from the drawback of being somewhat fragile and

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

sensitive with regard to the matchup between the data and the algorithms.In other words, recommender systems usually need a fair bit of tweaking to work well in a particular application setting. Added to this are the problems in gathering enough data in a common format about user behavior, item metadata and also in presenting the resulting recommendations in a form that can be easily integrated into target web pages.

The Vibes recommendation platform was built as a scalable and generic solution for Yahoo!'s recommendation needs. The platform has the capability of housing a variety of recommendation models along with all the machinery needed (data collection, data processing, model building, recommendation serving and reporting) to deploy recommendation modules for internal customers.

2 A Typical Vibes Use Case

Unlike the well-known problem of trying to construct a *(user, item)* $\rightarrow$ *rating* function given a set of numerical user ratings, Vibes is usually deployed in the following cases:

1. Provide an inter-item similarity or relatedness function: *(item1, item2)* $\rightarrow$ *similarity* $\in [0, 1]$. There are a couple of ways to do this as described in Section 4.

2. Provide a user-to-item recommendation function *(user, item)* $\rightarrow$ *recommended* $\in \{0, 1\}$. The output of this function is a boolean which decides whether to suggest *item* to *user* or not.

As an example of case 1, take a look at a product detail page in Yahoo! Shopping, for example an Apple iPod (`http://shopping.yahoo.com/p:Apple%20iPod%20touch%208GB%20MP3%20Player:1994935518`). Vibes item-to-item recommendations are shown in the section titled: **Yahoo! Shoppers Who Viewed This Item Also Viewed:**. All the data we need to supply these recommendations can be obtained from a web log of all the product pages viewed by visitors to Y! Shopping. If enough users visit a common set of items within, say a 90-day time period, those items can be the basis of generating recommendation rules.

Providing recommendations for case 2 is harder, primarily because of the sparsity of the data. Getting users to explicitly rate an item can present a barrier, but we can collect implicit binary rating data either from users' browse or buy history. It is fairly straight-forward to generate recommendations for users who *have* a history in Y! Shopping, but it is more challenging to cater to users who land-up directly from another search engine for example. The model then has to augmented with other behavioral data from the Yahoo! network (such as a user's search history) if available. This of course may raise some privacy issues in case we tap into a user's declared age, gender or any other personally identifiably information. Vibes customers may decide on a case-by-case basis to explicitly ask permission from users before showing recommendations that rely on their behavioral history.

Various Vibes customers have reported substantial benefits from adding a recommendation capability to their web site. For example Yahoo! Shopping has recorded a 16% increase in revenue after deploying Vibes. Similarly Yahoo! Small Business has measured a 30% increase in per-order revenue over manually generated cross-sell rules. The big advantage in Yahoo! is that it is possible to leverage users' network-wide behavior, including terms typed into general search to segment users into clusters and further personalize the recommendations. Currently we are in the process of developing such a model.

3 Platform Requirements

Yahoo! is in a unique position as one of the most popular destinations on the internet. Not only does Yahoo! have one of the largest user bases, a number of Yahoo! properties (such as Autos, Games, Shopping, Sports etc.) are ranked in the top 2 web sites of their respective categories. Yahoo! also has a major advantage in the fact that its users spend a significant amount of time on the web site, accumulating a large number of views and clicks, which translate into significant user behavioral history. These User Link Tracking (ULT) data ultimately find its way into data warehouses from which vertical-specific aggregated data can be queried. To achieve deployment of scale of the Vibes recommendation system across a broad range of Yahoo! properties, we started with the following set of requirements:

Loose coupling The recommender system stands apart from its customers. Changes in the recommendation platform, its algorithms and its infrastructure should be transparent to the consumers of the recommendations. This means that Vibes would run on a different set of systems running possibly different operating systems and language runtimes. A standard way of doing this is by using Web Services, in particular using REST principles.

Configurability It should be possible to easily tailor recommendations for each customer in terms of model parameters, data sources, and APIs. This could be done via a level of meta-programming where each instance of Vibes has a set of configuration parameters in the form of XML files that specify the methods of data generation, model building and the signatures of the RESTful web service.

Extensibility The Vibes platform should be able to easily accommodate new modeling methodologies and particular requirements for each customer deployment. The modeling block should be able to incorporate new code (written in any programming language) while the customer interface should have logic to activate business rules to merge, filter, compare and combine the recommendation results as required.

Easy integration The customer should have to take minimum effort both to provide data that feeds into the modeling engine as well as to consume the recommendation outputs. The data input could happen through standardized channels for instrumenting view and click events that flow into the warehouse. The recommendation output would be served through a web service that will be easy to parse and consume.

Quick deployment The platform needs to minimize the man power and incremental effort required for each new deployment. This could be done by having a standard configuration template that could be tailored to each customer's requirement by making localized changes for the input data source and output data format. No new code should have to be written in the common case, thereby alleviating the need for a long QA test cycle. The scheduling of model refreshes should happen automatically.

Quality checks We should anticipate operational issues such as missing or truncated input data, or perhaps changes in data distribution. Models should be evaluated based on metrics such as *precision*, *recall* and *coverage*. Every time a new model is built, it should be compared with a set of historical models and pushed into production only if the deviation is within tolerance limits.

Scalability The recommendation serving infrastructure hosting the web service should scale horizontally. That means that the total number of requests that can be served per second should be linearly proportional to the number of serving systems. In addition, 99.8% of responses need to be within 20 ms. On the model building front, the platform also needs to exploit data parallelism and should be able to take advantage of multi-CPU machines and grid clusters.

Reporting A recommender system is only as good as the visibility it provides into the effectiveness of its recommendations. Instrumentation needs to be embedded into the recommendations so that we can track the number of views and clicks made by users. The effectiveness of a recommendation module is measured using the click-through-rate (CTR) on the recommended items.

The architecture of Vibes takes into account the above requirements and is graphically shown in Fig.1. It is useful to think of the system as having three main components: the modeling engines, the data gathering and processing framework and the recommendation serving infrastructure. The data processing and modeling

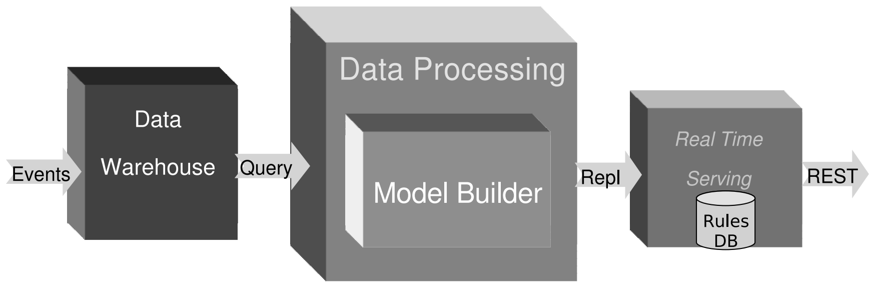

Figure 1: Vibes Platform Architecture

code runs in a central location in close proximity with the data warehouse while the model output is replicated to various data centers where the front-end systems serve it at real-time to co-located customer web servers. Before we delve into the details of each component it is worth noting that the components themselves are loosely coupled together, making it relatively easy to swap in new implementations.

4 Data Modeling

4.1 The Vibes Affinity Engine

The workhorse of Vibes data modeling is the **Affinity Engine** that is used to build item-to-item affinity models. Items could be almost anything in the Yahoo! universe, such as, product pages, auto makes, real estate listings, travel destinations, RSS feeds, computer games, search keywords and so on. User interaction with items is discretized into groups. A *group* (also sometimes called a session, transaction or market basket) is a set of events relating items. For example, a group could be page views by a single user, all the searches in a session, all the RSS subscriptions for a userid, products bought in a single checkout or pretty much any association of items. Item-to-item affinity is nothing but a set of association rules[3]. An association rule $A \rightarrow B$ relating two items A and B implies that we will recommend B when given A as input. To qualify as a rule, the pair (A, B) must co-occur frequently in a group i.e. they must pass certain thresholds of *support* and *confidence*. Support or minimum pair count is the least number of co-occurences of A and B for them to generate a rule. The choice of the support threshold depends on the characteristics of the data, particularly the ratio of the number of items to the number of groups. A higher item to group ratio (i.e. sparser data) may require lowering the support threshold to ensure sufficient number of rules (and item coverage). Typically we choose support thresholds of 9 or above to minimize noise. In our implementation, confidence or the affinity value is not a threshold, but instead an ordering metric. Confidence for a rule $A \rightarrow B$ is the conditional probability: $P(B|A) = P(A \cap B)/P(A)$. We generate all item pairs satisfying the support threshold and then for item A find the top n items X which have the highest $P(X|A)$. This gives rise to n recommendation rules (n being a config parameter for a particular deployment).

The most computationally intensive tasks involve finding the item pairs that have at least the minimum pair-count. At the scale of Yahoo!'s data (3 million items, 100 million groups), this is reasonably hard to do. This is where classic algorithms like Apriori[3] fail to scale. To make the problem tractable, we only consider binary rules, i.e., we only count item *pair* frequency (experiments have shown that the gain from having rules involving more than 2 items is not signficant). At a high level, this involves creating aggregate hash tables in memory mapping item-pairs to current counts and then flushing these tables to disk when memory overflows. Finally a second pass is made to merge and sum up all the item pair counts. There are several optimizations geared toward large data set processing in the actual implementation. Chief among these are encoding all the itemid strings to integers (as well as all itemid pairs to integers). Processing and comparing strings is the biggest consumer of cpu time and we have found this integer encoding method to be the biggest contributor to scalability. There are also optimizations involved in dropping items that fall below the occurrence threshold and dropping item pairs which do not make the cut of the top n affinity rules.

The affinity engine uses the well-known technique of association rule mining. Though this technology is quite mature, we have found it to be remarkably effective in real-life situations. Given enough data (i.e. a low item to group ratio) it is remarkably effective in tracking user behavior and often beats other more sophisticated models in predictive performance. Additionally it can consume large data sets with ease, even when used on a single system. In the near future, we plan to signficantly enhance the data processing capability of Vibes by deploying a version of the affinity engine that runs on a Hadoop[8] grid cluster where it would be able to utilize hundreds of compute nodes.

4.2 The Vibes Attribute Similarity Engine

Affinity based modeling seems to have only two weaknesses. It may not be able to produce an accurate model under these conditions:

- There is not enough data, i.e. the number of items to number of groups is high. This can happen if a particular web store-front does not have enough visitors or transactions. This reduces the probability of two items co-occurring enough number of times to overcome the support threshold.
- There are new items that have not accumulated enough history (in views, buys etc.). This is also known as the *cold start* problem. In general, it is very difficult to apply a behavioral model to such items.

One particular case where these conditions occur is for Yahoo! Real Estate and Autos which have high volume house or car listings. These listings have a finite lifetime and may not accumulate enough views within a 30-90 day window to make affinity based recommendations. In these cases, we plan to leverage structured metadata that is available with the listing. The idea is somewhat similar to the approach proposed in [6], but we use structured metadata instead of textual descriptions. In the real estate case, these would include attributes such as the price, zip, number of bedrooms and number of bathrooms for a house. Using just these metadata we can calculate a similarity score between any two listings based on a linear combination of attribute distances. The exact formulation of the similarity score between two items i and j each having n attributes $(a_{i1}, a_{i2}, \ldots a_{in})$ and $(a_{j1}, a_{j2}, \ldots a_{jn})$ respectively is:

$$Similarity(i,j) = 1 - \sum_{k=1}^{n} w_k * \delta_k(a_{ik}, a_{jk}) \tag{1}$$

where δ_k is the distance function and w_k is the weight applied to the kth attribute. It is possible to choose from a wide variety of distance functions such as Euclidean, Manhattan, Hamming distance etc. The distance function chosen is normalized to produce an output in the range $[0, 1]$.

Now comes the problem of finding weights. Our current approach is to learn these attribute weights from the data points where affinity model data exists. Based on user click behavior, we can model the similarity function to approximate the affinity (or confidence) of the recommendation $i \rightarrow j$. Given *Affinity(i, j)* and the attribute metadata for items i and j we can construct a set of linear equations of the form given in Eq.2.

$$Affinity(i,j) \simeq 1 - \sum_{k=1}^{n} w_k * \delta_k(a_{ik}, a_{jk}) \tag{2}$$

Since $\delta_k(a_{ik}, a_{jk})$ can be easily calculated, it is relatively straight-forward to do a linear regression to derive the weights w_k. A similar idea applies to constructing distance functions for categorical attributes. We intend to leverage user behavior to calculate affinities between categorical attribute labels and thereby deduce the relative distances between those attribute values.

4.3 User-to-Item Recommendations

This is usually considered the classic collaborative filtering (CF) problem, but the primary problem here is one of scale. Yahoo! has of the order of 500M users, and if we consider 1M items, we are faced with a 500 trillion cell matrix. Most of the algorithms described in the literature tend to break down when faced with data sets of this size. Furthermore, the data tend to be rather sparse, partly because of the rank of the user-item matrix and partly also because of practical problems of cookie churn which tends to inflate the number of users presented to the CF algorithm. The goal of the Vibes platform is to identify a small set of core algorithms that can be applied to a wide domain of user-to-item recommendation problems. We are currently evaluating several promising algorithms like those in [2].

In the meantime, it is possible to simplify the problem somewhat by noting that most of our use cases do not require a numerical rating prediction. We simply need to output a binary prediction of whether to suggest an item to a particular user or not. Of course we want to optimize the click response to our recommendations as well. One simple way of doing this would be to look at the click history of a user in a given vertical (say Yahoo! Shopping), i.e. for a user u, we have a set of items *Viewed(u)*. Then it is possible to expand this set using affinity based item-to-item recommendations:

$$Recommended(u) = \bigcup_{\forall i \in Viewed(u)} AffinityRecos(i)$$

where *AffinityRecos(i)* refers to the set of items recommended for item i by an affinity based model. When faced with new items with no behavioral data, it is possible to bring in a content dimension to the above by substituting (or augmenting) affinity based recommendations with attribute similarity based recommendations.

5 Model Building

The power of the Vibes framework stems not from the sophistication of the modeling engines but rather the ease and rapidity with which they can be deployed in large number of divergent use cases. The model building framework (the Data Processing block in Fig.1) has the job of aggregating, filtering and processing data to feed into the modeling engine. The following are some of the main operators that have been implemented in the Vibes Model Builder.

query Queries the data warehouse using an SQL like query language to extract user behavioral data.

mergesort Takes a set of compressed files as input, merges them and sorts them on the grouping key.

model Encapsulates the modeling engine and is further specialized into affinity, attribute similarity etc.

script Vehicle for plugging in ad-hoc scripts or executables written in any language for data processing.

evaluate Calculates metrics such as *coverage, precision, recall* for each model generated.

compare Compares current model with a set of historical models.

dbload Loads model result into serving database.

dbreplicate Replicates the model database across data centers.

In addition to specific operator properties, each operator has a set of common attributes (derived from a parent operator class in an inheritance hierarchy) such as name, scheduling frequency, data duration, input data source, output file and output schema. The individual operators are orchestrated into a workflow (specified as an

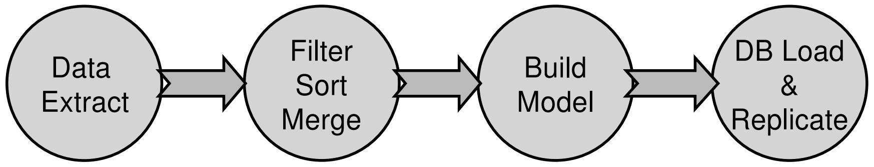

Figure 2: Vibes Data Modeling Pipeline

XML file) for each customer deployment. Fig.2 shows an example linear workflow, but in general this can be a dependency graph in the form of a DAG.

Model refreshes happen in an automated fashion depending on the desired refresh frequency (which in turn depends on the velocity of the data). Checks are built into each stage of the modeling pipeline so that downstream processing is halted in the event of any failure. For example if a signficant difference in the input data causes a substantial change in the number of rules, or the precision and recall of a model, then the model comparison operator fails and stops the dbload and dbreplicate operator preventing the deployment of the (possibly) faulty model into production. The model outputs are currently stored in a MySQL database table having the following schema: `recos: (src_item, dest_item, score)`. The `score` column refers to the degree of relatedness of the source item and the destination item and can either be the affinity or the attribute similarity score. The `recos` table is indexed on the `src_item` column, making it very easy to look up the set of recommended items using the SQL query: `select dest_item from recos where src_item='given item' order by score desc`.

A recommendation platform is only as good as the visibility it provides into its performance. From the outset, Vibes has ensured that appropriate metadata is sent along with the model output so that customers can record the number of views and clicks made by users on the recommendations. These data flow into the data warehouse from which we can report daily performance metrics such as views, clicks, click-through-rate and the number of unique end users targeted. This allows us to have ongoing quality checks tracking model performance.

6 Recommendation Serving

As we have mentioned before, the Vibes serving infrastructure (which is located in various data centers) is loosely coupled with the model building apparatus that is co-located with the central data warehouse. The glue is provided by MySQL replication. After the models are built, they are loaded into a master database that is then replicated asynchronously to slave serving databases in data centers across the country. The slave databases are read-only (except for batch model updates via replication) and this allows us to use the MyISAM storage engine optimized for batch rather than OLTP. The loose coupling betwen the data processing backend and the model serving frontend has several advantages namely better online performance and failure isolation. The Vibes service has to be up and serving recommendations 24 X 7 because of the nature of the traffic coming to our customers, the Yahoo! properties. There is no downtime due to model refreshes because the replication pushes the new model data into a staging area and then switches over in less than a second. Equally importantly, a failure in the model building process would stop the model refresh and replication, but the Vibes frontend would still satisfy recommendation requests using the older model data in the database. Fig.3 shows the detailed architecture of the Vibes recommendation web service.

The Vibes front-end is designed to scale horizontally and to serve 99.8% of requests within 20 ms. This is required to have an acceptable end-user experience. When a user browses a product page, say: `http://shopping.yahoo.com/p:Apple%20iPod%20touch%208GB%20MP3%20Player:1994935518` a call is made from the Y! Shopping web server to the Vibes web service in the form: `http://shopping.vibes.yahoo.com/vibes?method=shopping.recos.getVibes&itemid=1994935518`. After

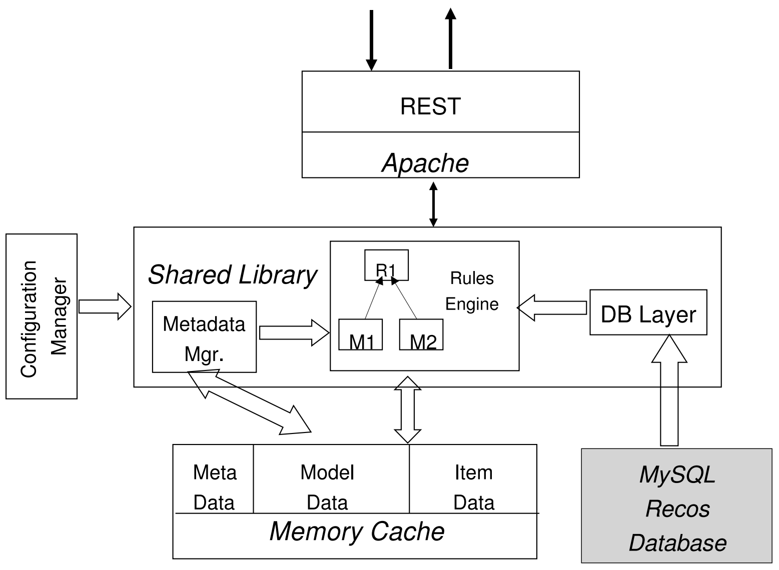

Figure 3: Vibes Serving Architecture

Vibes returns the recommendations in the form of an XML document, the Y! Shopping server parses the response and renders the HTML and graphics for the recommended items before sending them to user's browser. The browser has to load the complete page in 1-2s. One of the critical components in achieving this low latency is to use a large (currently 1GB) in memory cache (Fig.3) that caches responses from the database. The cache contains both positive and negative (no recommendations) results and uses technology similar to the popular *memcached* even though it is not partitioned across machines, (and so does not incur the penalty of an extra hop). It is possible to further optimize this process by using AJAX to make asynchronous calls to Vibes such that the main part of the page can load first while the recommendations are rendered in the background.

In the near future, we will be deploying Vibes also on the home pages of Yahoo! properties where recommendations could be provided without an item context. In that situation we would use information about a user's interests as identified by the Yahoo! cookie sent by the browser. This could be used to *score* the user, i.e. retrieve behavioral and demographic information about the user which are then used to place the user into one or more clusters (with certain probabilities). Finally these user scores are used to suggest items that are considered to be the most popular among members of the selected clusters. This dynamic user scoring for personalized recommendations is also going to be the responsibility of the Vibes front-end.

Vibes uses the Apache web server as the base for its web service. All the logic to parse the request, look up the recommendations from the serving database and formulate the XML response is compiled into a shared library that is loaded by Apache. In addition, there is the flexibility to tailor the recommendations in real-time using a series of rules that can include or exclude items or combine the results of a number of models. For example, in the cases where we have both affinity based and attribute similarity based models, it is possible to make run-time decisions about which kind of recommendations to serve based on either item coverage or the relative magnitudes of the affinity and similarity scores. In the near future we plan to implement a model testing capability that can be configured to partition the requests among a set of available models whose performance would then be evaluated by the Vibes reporting mechanism.

As is required for a system in production, the Vibes serving architecture has several layers of fault tolerance built-in. We are located in several geographically distributed data centers and within each there is redundancy in the web serving layer as well as in the database tier to enable tolerance of single system failures. Just as for the data processing backend, the Vibes front-end meticulously preserves a series of statistics related to service up-time, system load, number of requests coming in, number of recommendations served and the number of cases where there were no recommendations. These data flow into a reporting portal that graphically displays these metrics over time as well as sends alerts to operations teams if they deviate over tolerance bounds.

7 Future Direction and Conclusion

There are significant enhancements planned for each component of Vibes. For the modeling component, we would like to push deeper into personalized recommendations in a way that will scale to millions of items and hundreds of millions of users. These models are going to be extremely computationally intensive to build, so we have started moving our backend infrastructure to a Hadoop[8] grid environment. We have already done experiments for generating affinity rules by mining search query logs which supply this data at a scale that can only be processed on the grid. Having larger models will also put greater stress on the serving infrastructure, which probably will have to handle models that are at least 100X larger - requiring a different caching solution and possibly a different storage architecture.

Longer term, we would like to expose the power of the Vibes recommendation platform to a wider audience. The algorithms and infrastructure should be generic enough to be able to tackle data from a wide variety of domains and satisfy a large number of use cases. Moving towards a self-service capability that allows the customers themselves to configure and deploy the recommender system by defining API parameters and pointing the system to the data source would be a big plus. Our vision for Vibes is that it would be deployed by a simple drag-and-drop into a web-enabled application framework. That is when taking a platform-centric approach to building recommender systems would really pay off.

References

[1] Adomavicius G. and Tuzhilin A., *Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions*, IEEE Trans.on Knowledge and Data Engg., Vol 17, No 6, 2005.

[2] Agarwal D. and Merugu S., *Predictive discrete latent factor models for large scale dyadic data*, Knowledge Discovery and Data Mining (KDD) 2007.

[3] Agrawal R. and Srikant R., *Fast algorithms for Mining Association Rules in Large Databases*, VLDB 1994.

[4] Das A., Datar M, Garg, A, Rajaram S., *Google News Personalization: Scalable Online Collaborative Filtering* World Wide Web Conference (WWW) 2007.

[5] Linden G, Smith B., York, J., *Amazon.com Recommendations: Item-to-Item Collaborative Filtering*, IEEE Internet Computing, Vol 7, No 1, 2003.

[6] Melville P., Mooney R., Nagarajan R., *Content-Boosted Collaborative Filtering for Improved Recommendations*, Proc.of 18th Conf. on Artificial Intelligence, AAAI 2002, Edmonton Canada, July 2002.

[7] The Netflix Prize, `http://www.netflixprize.com`.

[8] Hadoop, `http://hadoop.apache.org`.