Liquid FM: Recommending Music through Viscous Democracy*

Paolo Boldi, Corrado Monti, Massimo Santini, and Sebastiano Vigna

Dipartimento di Informatica, Università degli Studi di Milano, Italy
corrado.monti@unimi.it

Abstract. Most modern recommendation systems use the approach of *collaborative filtering*: users that are believed to behave alike are used to produce recommendations. In this work we describe an application (Liquid FM) taking a completely different approach. Liquid FM is a music recommendation system that makes the user responsible for the recommended items. Suggestions are the result of a voting scheme, employing the idea of *viscous democracy* [3]. Liquid FM can also be thought of as the first testbed for this voting system. In this paper we outline the design and architecture of the application, both from the theoretical and from the implementation viewpoints.

1 Introduction

Most modern recommendation systems use the approach of *collaborative filtering* [15, 2]: users that are believed to behave alike are used to produce recommendations. The idea behind Liquid FM is to tip over this approach by making the user responsible for these matches: deciding who they want to resemble becomes a choice of the user, instead of being inferred algorithmically. This scenario can be cast as a voting scheme: each user has to select another one that is believed to be a good recommender. This idea allows us to use this task as a testbed for *viscous democracy* [3].

Viscous democracy is a kind of liquid democracy [13]. In liquid (or *delegative*) democracy, each member can take an active role—by participating directly and exercising their decision power—or a passive role—by *delegating* to other members their share of responsibility. It can be seen as a compromise between representative democracy, where voters are usually neglected any decision making and can only delegate others to do so, and direct democracy, where every voter is called to an active role, regardless of what their inclinations are.

In this sense, liquid voting systems try to take the best from both worlds. Every member's opinion, in a direct democracy, is directly relevant to a final decision, but the vote of each one can be (knowingly!) uninformed; instead, in a classical representative system, elected representatives are encouraged to be informed on the specific decision they are making, but on the other hand many people feel that their opinion on that matter is basically irrelevant. This phenomenon is called *political inefficacy* by social scientists—see for example [6]. Liquid democracy permits members to choose among expressing their opinion directly if they feel entitled to do so, or delegating their voting power if they believe others are more capable. Note that these two options are not necessarily exclusive—in our case, in fact, users will be able to do both, if they want to.

* The authors were supported by the EU-FET grant NADINE (GA 288956)

Viscous democracy was proposed by Boldi *et al.* in [3] as a particular way to compute the outcome of a liquid democracy voting scheme. It takes advantage of known techniques for measuring centrality in social networks, and in particular it resembles Katz's index [8]. It stems from the assumption that the delegating mechanism should transfer *a fraction* of the user voting power. I.e., if A delegates B and B delegates C, the trust that A puts in C should be less than if A voted C directly. This principle will be further detailed in the next section.

This framework can be used in a variety of settings. In our application, we show how it can be easily adapted to music recommendation. For a certain music genre, we ask users to express a short list of their favorite songs, or to delegate one of their Facebook friends they consider to be an expert on that genre. This builds a graph of delegations for each music genre. We wish to employ this data to create recommendations for each user.

In Section 2, we will give a panoramic overview of recommender systems, and of liquid democracy. Then, in Section 3 we will detail how we extract information from the delegation graph. In Section 4, we will describe how we have developed the system: how its algorithms were implemented, the architecture of its components, and the external resources we used; finally, in Section 5 we will sum up our work and present possible directions for future research.

2 Related works

Information overload, experienced everyday by internet users, has caused a growing research interest in recommendation systems—i.e., systems able to pro-actively suggest items of interest, in order to save users' time.

Recommender systems (RS) have been done in the past mainly with one of two different approaches, known as content-based filtering (CBF) and collaborative filtering (CF). Content-based filtering [12] guesses what a user will like in the future basing on the features of items they liked in the past. For example, if a person has expressed a preference for rock songs composed in the 90's by bands from Seattle, this will raise the chance they will like to listen to more similar music.

Collaborative filtering [5], instead, uses preferences from other users to infer what a certain user could like: if a group of users share a certain taste, we can suggest items liked by one of them to the others. Formally, we have a bipartite graph composed by users and items, and each user may be linked to a subset of items. First, CF identifies which users are close to a target user—that is, users that have a similar neighborhood on the bipartite graph. Then, we can recommend items linked by many of these users to the target.

CF does not need any specific knowledge about items: while CBF requires a phase of feature representation that can be costly for some domains, CF does not and can be applied without much effort to movies, news stories, websites, *et cetera* [11]. Furthermore, it is a scalable technique, that fits well to the size required by modern web services. For these reasons, CF is widely used in practice (for example, [2]).

A well-known problem of CF is *cold start* [16]: new users will have only a few ratings, and therefore recommendations are very inaccurate, since the system is not able to find close users. Another problem is *malicious rating*: by creating fake users with appropriate rating data, one can divert the CF mechanism [19].

We propose an application sharing traits with CF—we take a bipartite graph of user ratings of items as input, and employ user ratings to produce recommendations; however, we take in suggestions about the importance of trust in recommendations and from the study of liquid and viscous democracy. In fact, [17] found that people prefer recommendation from friends, over those by RS. Therefore, we make the choice of "close users"—used to produce recommendations—a decision of the target user, over his network of personal acquaintances. This simple strategy turns the system basically in a vote-casting setting, permitting us to employ *liquid democracy* to compute recommendations, and in particular to use this system as a proving ground of *viscous democracy* [3]. This could help in the above mentioned problems of CF: a new user is only required to indicate an existing user to be able to see recommendations, and fake users are unlikely to be trusted by real ones.

The idea of propagating trust to predict ratings has been previously analyzed theoretically, with the help of simulations made on real data. In [18], authors use a model where relationship between users are symmetric. A framework more similar to ours has been studied by [7]: they extensively tested the propagation of trust with a leave-one-out cross-validation approach, suggesting that this mechanism of building a *web of trust* could be useful in predicting ratings. The interplay between trusted social connections and RS has been studied by [9]; they employ trusted friends to make recommendations, but do not use mechanism for the diffusion of trust. A work with a similar inspiration to ours is [10], where the authors use a trust metrics derived from PageRank; their findings supports the idea that these methods can outperform CF, helping especially cold-start users.

3 Viscous democracy and recommender systems

From now on, we will denote with $d_G(x, y)$ the distance from node x to node y in the graph G, and with $o_G(x)$ the outdegree of node x in G; we may omit reference to G if it is obvious from the context.

Let us define U as the set of users and S as the set of items—they will be songs in our application[1]. $D = (U, A_D)$, with $A_D \subseteq U \times U$, is the directed graph of delegations; an arc from user u to u' means the former delegates the latter as an expert on the topic. $V = (U, S, A_V)$, with $A_V \subseteq U \times S$, is the bipartite graph of votes, where an arc from $u \in U$ to $s \in S$ means that the user u recommends item s.

We are going to put some restrictions on these graphs: first of all, we are going to assume that there is an underlying, undirected friendship graph $F = (U, E_F)$, with $E_F \subseteq U \times U$, where an edge $(u, u') \in E_F$ expresses a personal acquaintance of u and u'. We impose that $A_D \subseteq E_F$: this permits us to ensure that the trust expressed through a delegation is a result of personal knowledge, as suggested in [3].

Further, we are going to impose that $\forall u \in U$, we have $0 \leq o_D(u) \leq 1$, meaning that a user can delegate only one person, and $0 \leq o_V(u) \leq 3$, meaning that every user can vote up to 3 items[2]. We are going to consider only nodes $u \in U$ having $o_D(u) > 0 \vee o_V(u) > 0$. An example of such a setting is pictured in Figure 1.

[1] As we will explain in Section 4, we are going to consider different sets of songs and votes, one for each music genre treated. For the rest of this section, we are going to consider the music genre as fixed.

[2] We will explain this assumption in Section 3.2.

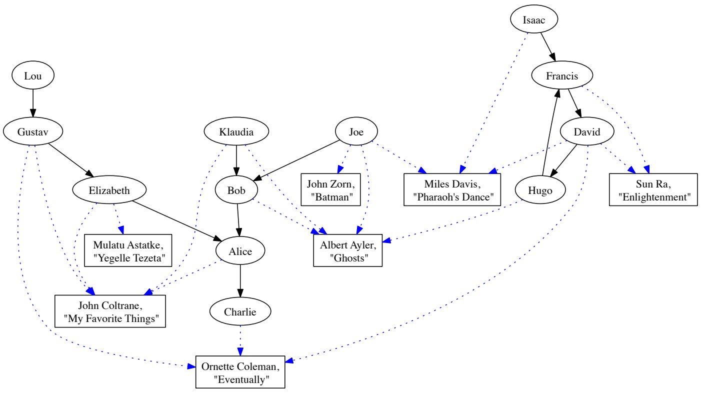


Fig. 1. An example of delegation and voting graphs. Users $u \in U$ are represented with a circle; songs $s \in S$ with a box; the delegation graph D is drawn with black solid arrows, while the bipartite voting graph V with blue dotted arrows.

3.1 Liquid voting

A voting system is a function $v_D : U \to \mathbb{R}$ assigning a score to each user, depending on the delegation graph. Such a function will be the basic building block of our recommendations.

Usually, in liquid vote this function is just the size of the tree with root in $u \in U$:

$$l_D(u) = \left| \{u' \in U | d_D(u', u) < \infty\} \right|$$

This function is used, e.g., by the well-known `LiquidFeedback`[3] platform. Nonetheless, it assumes that "trust" transferred from a to b is the same whether a delegated b directly, or whether they are connected by a long chain of delegations—and they may not even know each other.

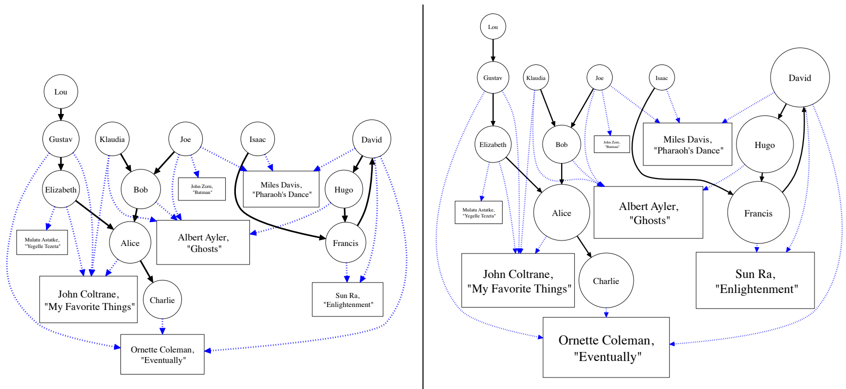
Let us assume that we wish, instead, that the amount of trust passed on from a to b is greater if $(a, b) \in A_D$, and lesser if there are many steps connecting them. To do so, we introduce a *damping factor* $\alpha \in (0, 1]$, defining how much of the voting power of a is transferred to b when a delegates b. Therefore, the scoring function characterizing *viscous* democracy will be:

$$v_D(u) = \sum_{u' \in U} \alpha^{d(u',u)} \tag{1}$$

Authors [3] have noted how, depending on the value of α, the behavior of the voting function greatly differs. For higher values of α, the fraction of trust "lost" in each delegation step becomes smaller and smaller; in fact, for $\alpha > 1$, we have that $v_D \to l_D$: all the nodes in the tree of u contribute with all their voting power to u, exactly as in pure liquid democracy. Note that if we allow $\alpha \geq 1$, we must explicitly avoid cycles in D—exactly as with pure liquid democracy; this constraint is not needed with viscous democracy with $\alpha \in (0, 1)$.

With α approaching 0, instead, the voting power becomes nontransferable: all users become equal, regardless of the delegations they received; in other words,

[3] `http://liquidfeedback.org/`



$\alpha = 0.25$ $\alpha = 0.75$

Fig. 2. The same graphs pictured in Figure 1 are here displayed with node size proportional to their viscous score, with two different values for α. Note how a higher α gives higher importance to users delegated by important users. Lowering its value get us closer to a simple vote count. For example, Ornette Coleman's song is ranked higher than Coltrane's only for higher α: this is because it is voted by fewer users, but those are recognized by the community as experts.

the model becomes a direct democracy, without any proxy vote. These differences are presented graphically in Figure 2, making use of the item-scoring function we will show in the next section.

3.2 Global recommendations

Having a score for each user, we can easily score each item $s \in S$. Indeed, we can define a function $r : S \to \mathbb{R}$ as

$$r(s) = \sum_{u \in U \,|\, (u,s) \in A_V} v_D(u) \tag{2}$$

This function will get us a score for an item proportional to the importance of who voted it, according to v_D. The score is completely defined by the graphs V and D. We can then proceed to rank each item with r, and present them to the users accordingly. As in many standard information retrieval tasks, a user looking for results (about a certain music genre, as we will explain in Section 4) will be presented with all possible items—all items in S—ranked from higher to lower r. Users of our application will be therefore more likely to listen to songs ranked higher in this list.

Let us call the *influence* of $u \in U$ the difference the votes of user u make in the final rankings—that is, $\sum_{s \in S} r(s) - r_{V \setminus \{u\}}(s)$. Please note that, since we have not normalized r, users giving more votes have a larger influence in the final rankings, serving the purpose of encouraging them to give more recommendations. However, it also explains why we had to put a limit on $o_V(u)$: if we had not, a single user u could have an arbitrary influence on the score r, resulting in the possibility of spam. To drop this constraint on o_V, we could require a costly effort from users to sustain a large number of votes—for example, a user could vote a song only if they have fully listened to it.

In the end, the influence of a user on item scores is determined by the number of recommendations they give—limited, but under their control—and by the delegations they received—unlimited, but not under their direct control.

As mentioned before, an example of how r behaves is pictured in Figure 2.

3.3 Personalized recommendations

The item-scoring function we presented gives the same ranks to whoever is their observer. This behavior is unusual in recommender systems, where the goal is to give the right recommendation to the right person. In our case, a user may be more interested in listening to what their delegate suggested, rather than other—possibly more popular—items. Looking at our example in Figure 2, Francis may be more interested in listening to "*Pharoah's Dance*", even if it is not globally highly-ranked, because it is the recommendation of his delegate David. Similarly, Hugo may be interested in it, because he has, in turn, delegated Francis.

This goal can be easily expressed as a personalized item-scoring function. Let us define a function $p : S, U \to \mathbb{R}$ as

$$p(s,u) = \sum_{u' \in U | (u',s) \in A_V} \alpha^{d(u,u')} \tag{3}$$

Such a function permits the user u to get a positive score only for the items recommended by users belonging to the chain of delegations starting in u. For the purpose of maintaining this intention, but at the same time avoiding to completely discard all the items highly ranked by the original r, we can define a linear combination of the two functions, normalized to 1:

$$c(s,u) = \delta \frac{p(s,u)}{\max\limits_{s' \in S} p(s',u)} + (1-\delta) \frac{r(s)}{\max\limits_{s' \in S} r(s')} \tag{4}$$

where $\delta \in [0,1]$ regulates the amount of personalization of c.

An example is pictured in Figure 3.

3.4 Insights for users

In addition to the presented ways to compute recommendations, the setting here described also permits to compute other information that may be of interest to the users. Particularly, it allows them to know how authoritative (i.e., trustable) their taste is in a particular music genre. The function v_D, in fact, can be normalized into a percentile-based scoring, obtaining an easy-to-read assessment in the form "u is better than $\widehat{v_D}(u)$ people out of 100" (for a specific genre), with $\widehat{v_D}(u) = 100 \frac{|\{u' \in U | v_D(u') < v_D(u)\}|}{|U|}$. It can then be used to provide useful information from two different perspectives:

1. Showing to the user a fair evaluation about which music genres they are believed to be more expert about.
2. Presenting to a user interested in learning more about a specific genre which of their friends is considered an expert—making use of the direct knowledge graph defined on page 3.

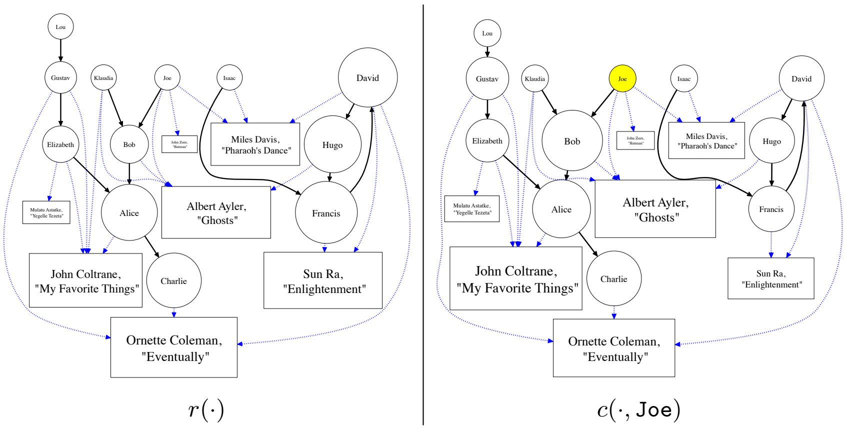


Fig. 3. The same graphs pictured in Figure 2 are here displayed with global song-scoring function r on the left and, on the right, with the personalized function c from the view point of user Joe (in yellow) and $\delta = 0.9$. In the latter, recommendations suggested by the delegate of Joe acquire more importance; those suggested by indirect delegates (namely, Alice and Charlie) increase as well, but by a minor amount.

4 Development

We will now discuss how the presented techniques have been implemented in practice. The final result is Liquid FM: a Facebook application that enable its users to vote one of their friends as an expert on a music genre, and (by means of the described formulas) recommends them some piece of music to listen to, by identifying the best experts.

Firstly, we will present a general overview of the architecture of Liquid FM, explaining the role of its main components; then, we will give a more detailed look at the implementations of the formulas presented in Section 3; finally, we will discuss the external components we employed.

Categories As anticipated in the previous section, we applied our scoring algorithms to 9 music genres, called *categories* from now on, and their set will be denoted as C. In this way, we will have different votes and different recommendations for each category. Such a behavior is closer to reality: an expert in HipHop is not assumed to be qualified to give, say, classical music suggestions. However, it also permits to have different graphs for the same users—an interesting fact for future analysis.

The selected categories were Classical, Electronic, Folk, HipHop, Indie, Jazz, Metal, Pop, Rock. They were chosen by inspecting LastFM top 20 tags[4] and discarding those not expressing a musical genre (such as "seen live") and sub-genres (having included Indie and Rock, we discarded "Indie Rock"). We decided to add Classical (only ranked 36th on Last Fm), since it is a different and interesting community, under-represented in services such as the one we referred to.

4.1 Overview

As pictured in Figure 4, Liquid FM features two main components:

[4] `http://www.last.fm/charts/toptags`

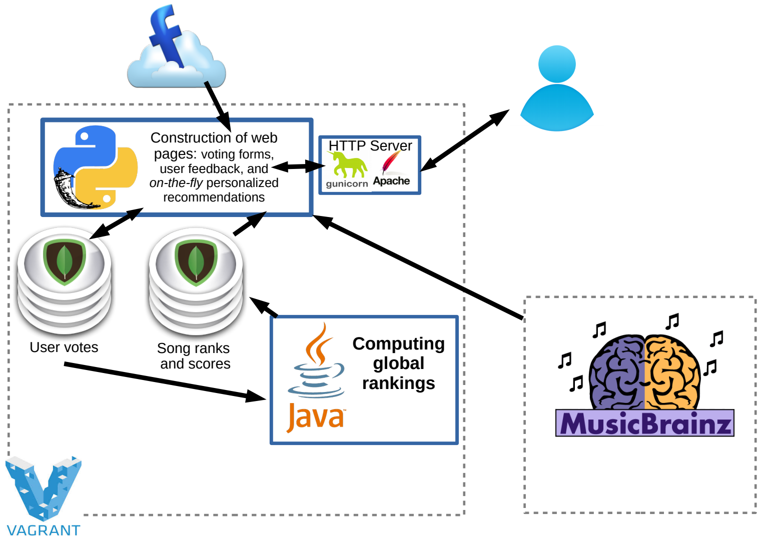


Fig. 4. A schematic representation of the architecture of Liquid FM. An arrow going from A to B indicates data flowing from A to B.

- a Java part, with the role of analyzing the whole graphs and computing global scores through v_D and r (equations 1 and 2): it is meant to be fast, and executed periodically;
- a Python part, with the role of glueing the different parts together and providing all the other functions: from the construction of web pages to the implementation of personal scores (functions p and c, equations 3 and 4).

These two parts interact with each other through a shared database, that persistently stores every information. We chose MongoDB, an open-source document-oriented NoSQL database, for efficiency, scalability, and flexibility. Furthermore, we do not often need complex operations, involving more than one collection, and we would like to control what is happening at application-level.

On this database, we have two main collections gathering user-submitted data: one for D and one for V. These collections are both represented in Figure 4 as "User votes". A document in the collection for D looks like this:

$$\{ \texttt{category} : c, \texttt{from} : u, \texttt{to} : u' \}$$

While a document in the collection for V has this structure:

$$\{ \texttt{category} : c, \texttt{user} : u, \texttt{advice} : s \}$$

The advice s is a dictionary containing author and title of the song, as well as a YouTube video id. In fact, we associate with each song selected by a user a YouTube video, in order to be able to play it as a recommendation. YouTube is in fact one of the largest and most used music streaming platforms, and it can be included in third-party services (with small limitations). A screenshot of the voting phase in displayed in Figure 5.

Please note that the structure of an advice, as well as the category c, is well-incapsulated: therefore, the schemas of these collections can be easily extended in the future in order to support different (i.e., not music-related) scopes.

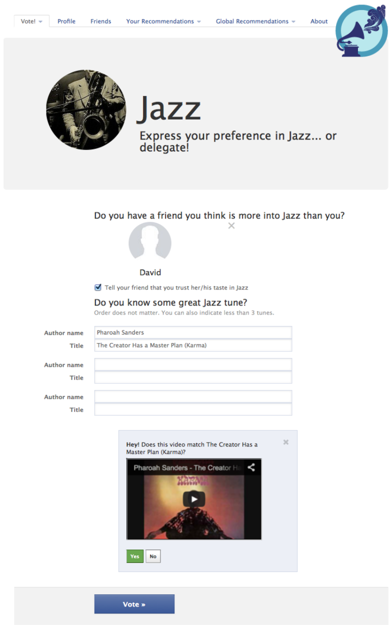


Fig. 5. A screenshot of the voting phase of a user.

4.2 Recommendations

The division of global and personalized recommendations into two separate components originates from efficiency reasons. Having to compute and store all the personalized scores for each user would be impracticable, as they would be $|C| \cdot |S| \cdot |U|$ scores. Therefore, they are computed with a lazy approach: when a user u asks for her personal recommendations, we compute all of them *on-the-fly* and cache them. Global recommendations, instead, are the main result of the system, and every user depends on them—even to see the personalized scores, since we use the function c (equation 4). For this reason, we compute them periodically with a fast Java component, and save them to a dedicated MongoDB collection.

Global recommendations in Java The global recommendation component was carried out in Java, since for this task it is faster than Python and since efficient open-source libraries to deal with graphs are available; in particular, we employed extensively the WebGraph framework [4] and the `fastutil` library.

This component is run periodically. It takes as input the graphs D and V, memorized in their MongoDB collections, and it results in a new collection for each category, composed of documents of this form:

$$\{ \quad \texttt{advice} \ : \ s, \ \texttt{rank} \ : \ r(s) \ \}$$

and in another collection ranking users, where each document has this form:

$$\begin{array}{l} \{ \quad \texttt{\_id} : \ u \, , \\ \quad\quad \textit{category}_1 : \ \{ \ \texttt{score} \ : \ v_D(u) \, , \ \texttt{perc} \ : \ \widehat{v_D}(u) \ \} \, , \ \ldots \\ \} \end{array}$$

We can schematize the process, for each category c, in these steps:

1. Read the graph D from MongoDB and convert it to WebGraph format.
2. Use a parallel implementation of the Gauss-Seidel method (from WebGraph) to compute v_D for each user u. We decided to choose 0.75 as the value of α.
3. Compute the percentile-based normalization $\widehat{v_D}$, and save the user-ranking collection to MongoDB.
4. Read the graph V from MongoDB, identifying the set S of songs. In this step, we also find which YouTube video is the most frequently associated with a certain song, using author and title as identifiers. While doing this, we compute $r(s)$ for each song s.
5. Save $r(s)$ in their collection, indexing documents by decreasing scores. Also save $\max_{s \in S} r(s)$, for normalization purposes.

Personalized on-the-fly recommendations As discussed above, personalized recommendations are computed on-the-fly by a Python component. Python was in fact chosen as the main language of the application, due to its versatility and its fast production times; also, we decided to use `Flask`[5], an open-source web development micro-framework particularly suited for our task.

Personalized recommendations are computed only when users ask for them, since they require to see only a very small part of the graphs, and because storing all of them would be unfeasible. The score we will use to rank personalized recommendations is the function $c(s, u)$ (eq. 4); in order to compute it we must, in the first place, compute p (eq. 3).

To compute $p(s, u)$ for all songs $s \in S$ and a fixed user u we walk through the chain of delegations on graph D, starting from u. Since $\forall u \; o_D(u) \leq 1$, this path on D is unique (although it may end in a cycle). Therefore, we simply proceed as follows (for a suitable stopping threshold ϵ):

---

1. Let `p` be a map with 0 as default value for missing keys.
2. **while** $t > \epsilon$ **and** $\exists u'$ s.t. $(u, u') \in A_D$
   (a) $u \leftarrow u'$ s.t. $(u, u') \in A_D$
   (b) For each s s.t. $(u, s) \in A_V$:
   $\mathtt{p}[s] \leftarrow \mathtt{p}[s] + t$
   (c) $t \leftarrow t \cdot \alpha$

Now we have all the ingredients for function c, and we can proceed to compute the ranking order according to it.

First of all, consider that the ranking order induced by $c(s)$ is equivalent to

$$\bar{c}(s) = k \cdot p(s, u) + r(s) \qquad \text{where} \qquad k = \frac{\delta \cdot \max_{s' \in S} r(s')}{(1 - \delta) \cdot \max_{s' \in S} p(s', u)}$$

Therefore, for each element s of the map `p`, we multiply its value by k and add the value of $r(s)$. Then, we retrieve all the other items s s.t. $r(s) \geq \min_{s'} \mathtt{p}[s']$, and insert them in the map `p`. Finally, we can build the iterator of personal recommendations by chaining two iterators:

1. the iterator of all elements in `p`, sorted by their values;

[5] `http://flask.pocoo.org/`

2. the iterator of all other elements $s \in S$ having $r(s) < \min_{s'} \mathtt{p}[s']$, sorted by their values; remember that they are already indexed in this order in the database.

The final iterator can be implemented in a lazy fashion, allowing us to retrieve the elements of the second iterator only when necessary. The first iterator, instead, will be computed eagerly upon its request, and then cached. To cache these (and other) values, we employed `redis`[6], an open-source in-memory key-value cache.

4.3 External services

To conclude this section, we would like to briefly describe the main external software components we used in developing Liquid FM.

Facebook As mentioned earlier, Liquid FM is a Facebook application. The reason for it is that we used the Facebook friendship graph as the graph F defined on page 3. In fact, Facebook is at the moment the largest existing social network (with 1.4 billion users), and it has been previously used as a good approximation of an acquaintance graph [1]. Therefore, we require users to have a Facebook account, in order to limit their choice of delegate to their acquaintances. Accordingly, in the collections described earlier, we used a Facebook-provided id to identify a user u.

Musicbrainz To ensure the validity of the set S of songs chosen by users, we check them against the Musicbrainz database. Musicbrainz is an open music database that anyone can edit. At the time of writing, it contained information about more than 900 000 artists and 14 000 000 recorded songs. Since it follows the open-content paradigm, a user who does not find its favorite song in the database is in principle free to add it; however, the community-review process acts as a filter. Furthermore, Musicbrainz provides a disk image to set up a virtual machine with a fully-functioning, self-updating Musicbrainz server; we used this approach to be able to access the database fast, without network delays and minimizing the impact on their hosts.

5 Discussion, conclusions and future work

In this work, we presented a Facebook application aimed at putting the viscous democracy framework [3] to the test. This is at the same time a proof-of-concept of how that voting system can be practically implemented in a real-world social network, and a way to collect data corroborating (or disproving) the supposed advantages of viscous democracy when compared to other, more standard, ways of performing elections in a social setting. An interesting point, here, is that the usage of viscous democracy for recommendation seems to avoid the filter bubble [14], at least in its more algorithmic sense, because this kind of recommendation does not rely on collaborative filtering but is based on a conscious choice. Whether this choice (delegation) can itself induce a similar kind of bubble will be subject of future analysis.

The discussed application is currently active on `http://bit.ly/liquidfm`, and we have so far collected some small datasets; currently, the delegation graphs consist of few tens of delegations, so it is impossible to draw any conclusion from

[6] `http://redis.io/`

them. In order to be able to collect larger amount of information it is crucial that we find a way to make the application *viral*: this is a matter of social engineering that needs to be taken into careful consideration.

References

1. Backstrom, L., Boldi, P., Rosa, M., Ugander, J., Vigna, S.: Four degrees of separation. In: Proceedings of the 4th Annual ACM Web Science Conference. pp. 33–42. ACM (2012)
2. Bernhardsson, E.: Collaborative filtering at spotify. New York Machine Learning meet-up (jan 2013), `http://www.slideshare.net/erikbern/collaborative-filtering-at-spotify-16182818`
3. Boldi, P., Bonchi, F., Castillo, C., Vigna, S.: Viscous democracy for social networks. Communications of the ACM 54(6), 129–137 (2011)
4. Boldi, P., Vigna, S.: The webgraph framework i: compression techniques. In: Proceedings of the 13th international conference on World Wide Web. pp. 595–602. ACM (2004)
5. Breese, J.S., Heckerman, D., Kadie, C.: Empirical analysis of predictive algorithms for collaborative filtering. In: Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence. pp. 43–52. Morgan Kaufmann Publishers Inc. (1998)
6. Finkel, S.E.: Reciprocal effects of participation and political efficacy: A panel analysis. American Journal of political science pp. 891–913 (1985)
7. Guha, R., Kumar, R., Raghavan, P., Tomkins, A.: Propagation of trust and distrust. In: Proceedings of the 13th international conference on World Wide Web. pp. 403–412. ACM (2004)
8. Katz, L.: A new status index derived from sociometric analysis. Psychometrika 18(1), 39–43 (1953)
9. Ma, H., King, I., Lyu, M.R.: Learning to recommend with social trust ensemble. In: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval. pp. 203–210. ACM (2009)
10. Massa, P., Avesani, P.: Trust-aware recommender systems. In: Proceedings of the 2007 ACM conference on Recommender systems, pp. 17–24 (2007)
11. Massa, P., Bhattacharjee, B.: Using trust in recommender systems: an experimental analysis. In: Trust Management, pp. 221–235. Springer (2004)
12. Mooney, R.J., Roy, L.: Content-based book recommending using learning for text categorization. In: Proceedings of the fifth ACM conference on Digital libraries. pp. 195–204. ACM (2000)
13. O'Donell, G.A.: Delegative democracy. Journal of democracy 5(1), 55–69 (1994)
14. Pariser, E.: The filter bubble: What the Internet is hiding from you. Penguin UK (2011)
15. Ricci, F., Rokach, L., Shapira, B.: Introduction to recommender systems handbook. In: Recommender systems handbook, pp. 1–35. Springer (2011)
16. Schein, A.I., Popescul, A., Ungar, L.H., Pennock, D.M.: Methods and metrics for cold-start recommendations. In: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval. pp. 253–260. ACM (2002)
17. Sinha, R.R., Swearingen, K.: Comparing recommendations made by online systems and friends. In: DELOS workshop: personalisation and recommender systems in digital libraries. vol. 1 (2001)
18. Yu, B., Singh, M.P.: A social mechanism of reputation management in electronic communities. In: Cooperative Information Agents IV-The Future of Information Agents in Cyberspace, pp. 154–165. Springer (2000)
19. Zhou, X., Xu, Y., Li, Y., Josang, A., Cox, C.: The state-of-the-art in personalized recommender systems for social networking. Artificial Intelligence Review 37(2), 119–132 (2012)