Title	Coping with noisy search experiences
Authors(s)	Champin, Pierre-Antoine; Briggs, Peter; Coyle, Maurice; Smyth, Barry
Publication date	2010-05
Publication information	Knowledge-Based Systems, 23 (4): 287-294
Publisher	Elsevier
Link to online version	http://dx.doi.org/10.1016/j.knosys.2009.11.011
Item record/more information	http://hdl.handle.net/10197/1999
Publisher's version (DOI)	10.1016/j.knosys.2009.11.011

Coping with Noisy Search Experiences

Pierre-Antoine Champin[1,1], Peter Briggs[1], Maurice Coyle[1], Barry Smyth[1]

[a] *LIRIS, Université de Lyon, CNRS, UMR5205, Université Claude Bernard Lyon 1, F-69622, Villeurbanne, France,*
[b] *CLARITY: Centre for Sensor Web Technologies, School of Computer Science and Informatics, University College Dublin, Ireland*

Abstract

The so-called *Social Web* has helped to change the very nature of the Internet by emphasising the role of our online experiences as new forms of content and service knowledge. In this paper we describe an approach to improving mainstream Web search by harnessing the search experiences of groups of like-minded searchers. We focus on the HeyStaks system (*www.heystaks.com*) and look in particular at the experiential knowledge that drives its search recommendations. Specifically we describe how this knowledge can be noisy, and we describe and evaluate a recommendation technique for coping with this noise and discuss how it may be incorporated into HeyStaks as a useful feature.

Experience is the name everyone gives to their mistakes. —Oscar Wilde

Key words: experience, web search, recommender system

1. Introduction

The now familiar Social Web reflects an important change in the nature of the Web and its content. The development since 1999 of blogs, as a simple way for users to express their views and opinions, ushered in this new era of *user-generated content* (UGC) as many sites quickly began to offer a whole host of UGC alternatives including the ability to leave comments and write reviews, as well as the ability to rate or vote on the comments/opinions of others. The result has been an evolution of the Web from a repository of information to a repository of experiences, and an increased emphasis on people rather than content. In combination with social networking services, this has precipitated the growth of the Social Web as a platform for communication, sharing, recommendation, and collaboration.

Web search has continued to play a vital role in this evolving online world and there is no doubting the success of the mainstream Web search engines as

Based on works supported by Science Foundation Ireland, Grant No. 07/CE/I1147, the French National Center for Scientific Research (CNRS), and HeyStaks Technologies Ltd.

a key information tool for hundreds of millions of users everyday. Given the importance of Web search it is no surprise that researchers continue to look for new ways to improve upon the mainstream search engines. However, new tools are also needed to gather, harness, reuse and share, in the most efficient and enjoyable way, the experiences captured by UGC [**?** **?**]. One particular line of research has focused on using recommendation technologies in an effort to make Web search more personal: by learning about the preferences and interests of individual searchers, personalized Web search systems can influence search results in a manner that better suits the individual searcher [**?** **?**]. Recently, another complementary research direction has seen researchers explore the *collaborative* potential of Web search by proposing that the conventional *solitary* nature of Web search can be enhanced in many search scenarios by recognising and supporting the sharing of search experiences to facilitate synchronous or asynchronous collaboration among searchers [**?** **?**]. Indeed, the work of [**?** **?**] has shown that collaborative Web search can lead to a more personalized search experience by harnessing recommendations from the search experiences of communities of like-minded searchers.

Our recent work [**?**] has led to the development of a new system to support collaborative Web search. This system is called HeyStaks (*heystaks.com*) and it benefits from providing a collaborative search experience that is fully integrated with mainstream search engines such as Google. HeyStaks comes in the form of a browser toolbar and, as users search as normal, HeyStaks captures their search experiences and promotes results based on their past search experiences and the experiences of friends, colleagues, and other like-minded searchers. HeyStaks introduces the key concept of a *search stak* which serves as a repository for search experiences. Users can create search staks to represent their search interests and they can share their staks with others to create pools of focused search experiences.

The key contribution of this paper is to focus on an important challenge faced by HeyStaks and to propose a recommendation solution to meet this challenge. The challenge concerns the basic stak selection task: prior to a search, a HeyStaks user must select an *active* stak so that their search experiences can be correctly stored and so that they can receive appropriate recommendations. Many users have built this into their search workflow and HeyStaks does contain some simple techniques for automatically switching to the right search stak at search time. However, many users forget to choose a stak before they search and, as a result, search experiences are often mis-filed in an incorrect stak. Ultimately this limits the effectiveness of HeyStaks and contributes significant experience noise to search staks.

In what follows we will briefly introduce the HeyStaks system. Then we will describe the development of a stak recommendation technique as part of HeyStaks' *stak maintenance* features, which allow stak owners to review and edit stak content. In brief, our stak recommender is capable of highlighting potentially mis-filed experiences and offers the user a suggested target stak that is expected to provide a better fit. We will describe an evaluation on real-user search data to demonstrate the effectiveness of this technique.

2. HeyStaks: an overview

HeyStaks is a collarative search systems, similar to those presented in [**?** **?**]. Our primary goal in designing HeyStaks is to help improve upon the search experience offered by mainstream search engines, while at the same time allowing searchers to search as normal with their favourite engine. In this section we will outline the basic HeyStaks system architecture and summarize how result recommendations are made during search. In addition we will make this discussion more concrete by briefly summarizing a worked example of HeyStaks in action.

2.1. Concepts and Architecture

HeyStaks adds two important collaboration features to any mainstream search engine. First, it captures users' experiences in using the search engine, and store them in *search staks*. Staks are a type a of folder that users can create to store search experiences related to a given topic of interest. Staks can also be shared with others so that their own searches will also be added to the stak. Second, HeyStaks uses staks to generate recommendations that are added to the underlying search results that come from the mainstream search engine. These recommendations are results that stak members have previously found to be relevant for similar queries in the context of this stak, and help the searcher to discover results that friends or colleagues have found relevant, results that may otherwise be buried deep within the engine's result-list.

HeyStaks takes the form of two basic components: a client-side *browser toolbar* and a back-end *server*. The toolbar allows users to create and share staks and provides a range of ancillary services, such as the ability to tag or vote for pages. The toolbar also captures search result click-thrus and manages the integration of HeyStaks recommendations with the default result-list. The back-end server manages the individual stak indexes (indexing individual pages against query/tag terms and positive/negative votes), the stak database (stak titles, members, descriptions, status, etc.), the HeyStaks social networking service and the recommendation engine.

2.2. Running Example

To make things more concrete, consider the following example. Pierre, Maurice and some colleagues are using the LaTeX typesetting system on a regular basis, and Web search as a source of information about how to use it. Pierre created a search stak called "LaTeX" and shared this with Maurice and colleagues, encouraging them to use this stak for their LaTeX-related searches.

Fig. **??** shows Maurice selecting this stak as he embarks on a new search about the tabular environment, and Fig. **??** shows the results of this search. The usual Google results are shown, but in addition HeyStaks has made one promotion. This was promoted because other members of the "LaTeX" stak had recently found these results to be relevant; perhaps they selected them for *similar* queries, or voted for them, or tagged them with related terms. These recommendations may have been promoted from much deeper within the Google

Figure 1: Selecting a new active stak.

result-list, or they may not even be present in Google's default results. Other relevant results may also be highlighted by HeyStaks, but left in their default Google position. In this way Pierre and Maurice benefit from promotions that are based on their previous similar searches. In addition, HeyStaks can recommend results from Pierre and Maurice's other staks, helping them to benefit from the search knowledge that other groups and communities have created.

Separately from the toolbar, HeyStaks users also benefit from the HeyStaks *search portal*, which provides a social networking service built around people's search histories. For example, Fig. **??** shows the portal page for the "LaTeX" stak. It presents an activity feed of recent search history and a query cloud that makes it easy for the user to find out about what others have been searching for. The search portal also provides users with a wide range of features such as stak maintenance (e.g., editing, moving, copying results in staks and between staks), various search and filtering tools, and a variety of features to manage their own search profiles and find new search partners.

2.3. Generating Recommendations

In HeyStaks each search stak (S) serves as a profile of the search activities of the stak members. Each stak is made up of a set of result pages ($S = \{p_1, ..., p_k\}$) and each page is anonymously associated with a number of implicit and explicit interest indicators, including the total number of times a result has been selected (sel), the query terms ($q_1, ..., q_n$) that led to its selection, the number of times a result has been tagged (tag), the terms used to tag it ($t_1, ..., t_m$), the votes it has received (v^+, v^-), and the number of people it has been shared with ($share$).

In this way, each page is associated with a set of *term data* (query terms and/or tag terms) and a set of *usage data* (the selection, tag, share, and voting counts). The term data is stored as a Lucene (*lucene.apache.org*) index, with each page indexed under its associated query and tag terms, and provides the basis for retrieving and ranking *promotion candidates*. The usage data provides

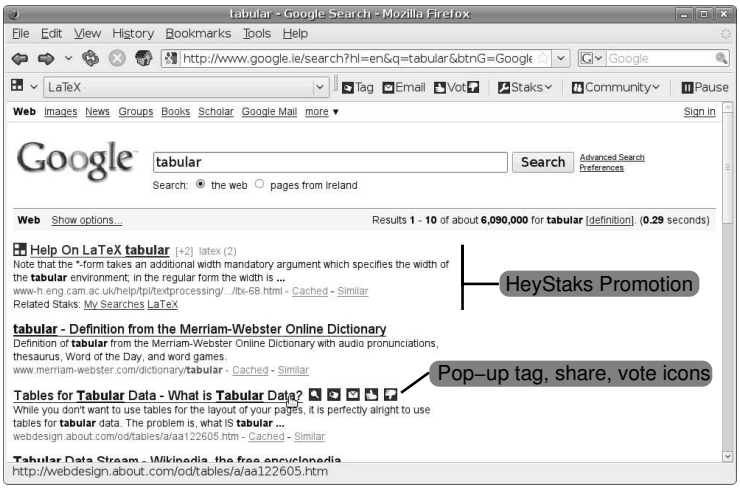

Figure 2: Google search results with HeyStaks promotions.

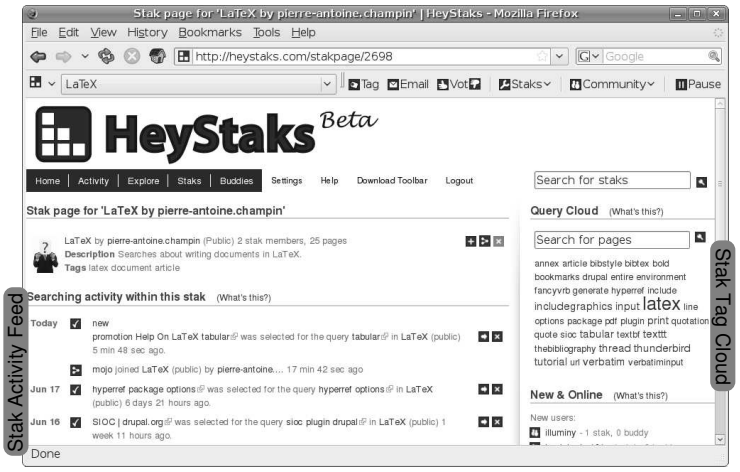

Figure 3: The HeyStaks search portal provides direct access to staks and past searches.

an additional source of evidence that can be used to filter results and to generate a final set of recommendations. At search time, recommendations are produced in a number of stages: first, relevant results are retrieved and ranked from the stak index; next, these promotion candidates are filtered based on the usage evidence to eliminate spurious recommendations; and, finally, the remaining results are added to the Google result-list according to a set of presentation rules.

Briefly, HeyStaks uses a number of different recommendation rules to determine how and where a promotion should be added. Space restrictions prevent a detailed account of this component but, for example, up to 3 *primary* promotions are added to the top of the Google result-list and labelled using the HeyStaks promotion icons. If a remaining promotion is also in the default Google result-list then this is labeled in place. If there are still remaining promotions then these are added to the *secondary* promotion list, which is sorted according to TF*IDF scores. These recommendations are then added to the Google result-list as an optional, expandable list of recommendations. The interested reader can refer to [**?**] for more details.

It is worth noting that, unlike many other recommender systems, HeyStaks does not filter information from a search stak in order to personalize recommendation results. Personalization does however occurs in HeyStaks (unlike other systems such as Google's SearchWiki) by *chosing the active stak* from which recommendation knowledge will be used. The focus is therefore on the task or context of the user, rather than an ever-valid user profile that would not account for the diversity of their search activity.

3. Stak recommendation

With the current version of HeyStaks the focus is very much on the recommendation of results during search. However, in this section we will argue the need for a second type of recommendation – the recommendation of staks to users at search time – as a way to help ensure that the right search experiences are stored in the right staks.

3.1. The Problem of Stak Noise

One problem faced by HeyStaks, and many other systems relying on users' experiential knowledge, is that of reliably collecting that knowledge. Explicitly requesting information from the user is often considered too intrusive, and discourages many users from using the system in the first place. On the other hand, implicitly collecting this information is error prone because in order to interpret users' actions in terms of reusable knowledge, the collection process must be based on some idealized expectation of user behaviour.

For example, HeyStaks relies on users selecting an appropriate stak for their search experiences, prior to selecting, tagging or voting for pages. Those actions are therefore considered as evidence that the page is relevant to the stak currently active in the HeyStaks toolbar, and to the query, in the case of a selection.

The relevance to the query is not guaranteed, though, since the page may prove less interesting than its title suggested. More important for HeyStaks, the relevance to the selected stak is not guaranteed either, for it is common occurence that users forget to select a stak those actions. Many pages are then filed by default in the users "My Searches" stak (which is not shared with other users), or even in an unrelated stak. The point is that this limits the quality of search knowledge contained within the staks, hence the quality of the recommendations made by the system.

3.2. Coping with Stak Noise

A solution to break that vicious circle would be for HeyStaks to automatically select, or at least suggest, the appropriate stak when the user starts a query. This is a meta-recommendation problem [**?**] (selecting the stak from which search results will be recommended). We therefore face two challenges: using a repository of recommendation knowledge (search experiences) for another purpose than the one it was designed for (meta-recommendation), and using it despite the significant amount of noise it contains. Should we succeed, the quality of the collected experiences would increase thanks to the stak recommendation, which in turn would itself be improved.

We envision two different uses of the stak recommender system. The first one, the *on-line phase*, has already been described above: at query time, in order to ensure that the selected stak corresponds to the focus of the user's search. The second use is an *off-line phase*: whenever they want, the owner of a stak can visit a maintenance page, where the system will present them with *a)* pages in that stak which could be moved or copied to one or several other staks, and *b)* pages from other staks which could be moved or copied to this stak. Though the off-line phase is more demanding, we believe that some stak owners will have an incentive to improve the quality of their staks (or "curate" them); e.g. for staks shared by a community of knoweldge workers with a strong need for accurate experience sharing. In the rest of this paper, we will focus on the off-line phase.

4. Noise-Robust Classifier

We consider the off-line stak recommendation problem as a *classification* problem: our goal is to train a classifier to find the "correct" stak for each page stored in the HeyStaks repository. More precisely, the recommender system will use this classifier to find the three most likely staks for each page, and submit them to the stak owner for validation.

The problem is of course to correctly train and evaluate that classifier despite the noise in the available data. Since manually tagging a significant set of pages as relevant or noisy is not feasible, we first propose a measure that we will use as a proxy for relevance. Then we will show how this measure can be used to evaluate and improve our classifier.

In the following, we will represent the search experience stored in each stak S as a *hit matrix* h^S where h^S_{ij} is the number of times that term t_j has been

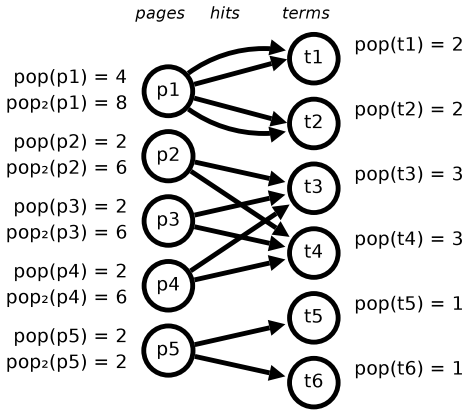

Figure 4: Popularity measure pop_2 illustrated.

related to page p_i, either as a query term or a tag. Since we use pseudo-terms to represent votes, this matrix captures in a synthetic way all the term and usage data used by HeyStaks. Each line h_i^S of the hit matrix, the *hit vector* of page p_i, is how that page will be represented in our classifier.

4.1. Predicting Relevance with Popularity

An immediate approach to predict the relevance of a page or a term to a particular stak is to consider its popularity, *pop*, measured as the total number of hits accounted for by this page or term in the stak's hit matrix h^S:

$$pop(p_i) \doteq \sum_j h_{ij}^S \qquad pop(t_j) \doteq \sum_i h_{ij}^S \tag{1}$$

The rationale is that a page or a term may be added to a stak once or twice by accident, but if it has been repeatedly selected for that stak, it is probably relevant to it. The problem with these two measures, though, is that they are independent of each other. We would also like to take into account the fact that a page may benefit from the popularity of the terms for which it was selected: hence, we propose a second measure of popularity, pop_2, for pages, defined as follows:

$$pop_2(p_i) \doteq \sum_j pop(t_j) \times h_{ij}^S \tag{2}$$

This is illustrated by Fig. **??**, which shows that a page like $p1$ with a high number of hits will still be popular, but a set of pages sharing the same terms will "inherit" the popularity of that term, even if each one of them has a low number of hits (see $p2, p3, p4$).

We now want to normalize this popularity measure, in order to make it comparable across staks, regardless of the span and skew of their popularity

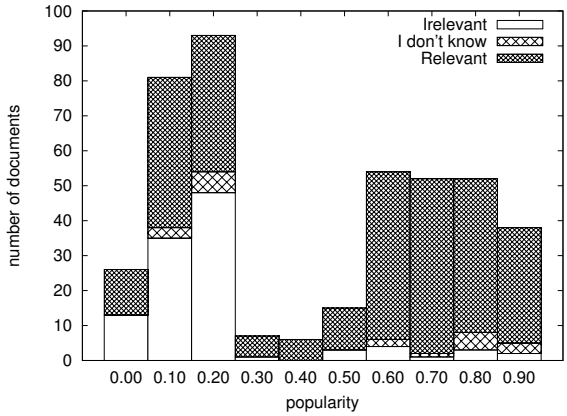

Figure 5: Poll results. For each stak, the 15 most popular pages and the 15 least popular pages have been manually evaluated by the stack owner.

distributions. First, we bring it between 0 and 1 (addressing the span problem), then we center the mean popularity to 0.5 (addressing the skew problem). The normalized popularity np is computed as follows:

$$np(p_i) \doteq \left(\frac{pop_2(p_i)}{\max_j pop_2(p_j)}\right)^{\frac{\log 0.5}{\log \text{mean}_k pop_2(p_k)}} \quad (3)$$

In order to evaluate the validity of our popularity measure as a predictor of page relevance, we performed a small user evaluation. We limited our study to the 20 biggest shared staks (smaller staks having potentially not reached a critical mass for efficient recommendation). For each of those 20 staks, we picked the 15 most popular pages and the 15 least popular pages. We presented them to the stak owner in a random order, and asked them if each page was relevant or not to that stak. Since other stak users are supposed to join a stak because they share the stak owner's interest in its topic, we assume that the owner's opinion is representative of the user community of that stak.

The results of this evaluations are shown in Fig. **??**. We see that pages with a high popularity are almost always considered relevant by users. Unpopular pages, on the other hand, are uncertain: about half of them are relevant, while the other half are not. This is not a big surprise since our popularity measure relies on the number of times a page has been selected; an unpopular page may be relevant but too recent to have become popular yet. We are however encouraged to consider that the noisy experience is located in the unpopular part of our experience repositories.

4.2. A Weighted Stak Classifier

Confident in our popularity measure, we have decided to use it for training our classifier. The popularity measure can be used to *weight* the training in-

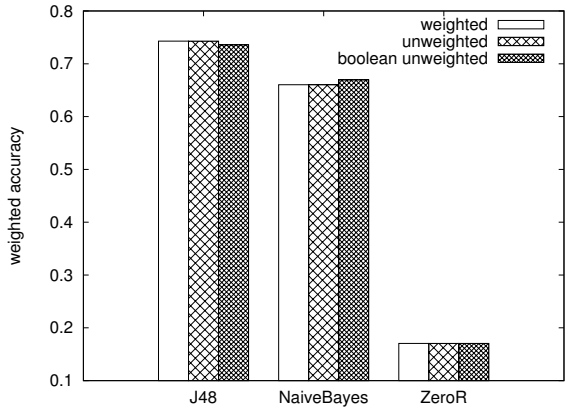

Figure 6: The weighted accuracy is only slightly influenced by the use of popularity in the training set.

stances, so that the classifier learns more from popular pages (more likely to be reliable) than from less popular ones. This weighting is also used to compute the accuracy of the classifier: indeed, the fact that the classifier disagrees with the experience repository for a page with a low weight (*i.e.* considered unreliable) should not have the same importance as a disagreement on a highly weighted (hence reliable) page. The *weighted accuracy* that we use is then computed by dividing the sum of the weights of the "correctly" classified[2] pages by the sum of the weights of all the pages.

Our first training set comprises all pages from the 20 largest shared staks in HeyStaks. Each instance represents a page p_i from a stak S by its hit vector h_i^S, its class is the stak identifier, and its weight is $np(p_i)$. We use three classifiers: a ZeroR random classifier (always predicting the more frequent class), a J48 decision tree [**?**] and a naive bayesian classifier. We tested those three classifiers with a standard 10-fold cross-validation. The resulting weighted accuracies are 17%, 74% and 66% respectively.

These first results were encouraging. However, we wanted to measure the benefit of weighting the training instance with our popularity measure. We did the same test, but with unweighted instances. The results are only marginally worse: 17%, 73% and 64% respectively. We then trained the classifiers with boolean vectors instead of hit vectors (i.e. replacing any non-null number of hits by 1), thus removing even more information about the popularity (np is computed using the number of hits). The results are still very similar (and even slightly *better* for the NaiveBayes), as shown on Fig.. **??**

This surprising result may be explained by considering how the accuracy

[2]Where "correctly" actually means "in agreement with the available data", which is known to be partially inaccurate.

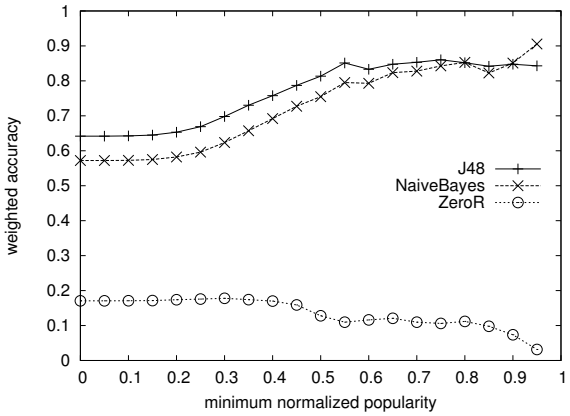

Figure 7: J48 and NaiveBayes classifiers are better at classifying popular pages.

varies for pages with different weights. This is shown in Fig. **??**, where each point represents the accuracy of the classifiers (trained with unweighted boolean vectors) when considering only their results for pages with a minimum *np*. We see that both the J48 and NaiveBayes are better at classifying popular pages, and that this is not a bias in the data, since the random classifier does not share this property[3]. We suggest that there is a correlation between popularity and purely structural similarity, which may account for the fact that weighting the instances does not significantly improve the accuracy – since this is information that the classifiers learn anyway.

4.3. Stak Kernels

Another interesting thing that Fig. **??** teaches us is that the evolution of the accuracy against popularity is not linear. Accuracy first stagnates when considering pages with $np \geq 0.3$, then increases steadily until around $np \geq 0.6$, then stabilizes again. This seems to indicate that $np = 0.6$ is a threshold below which pages are harder to predict, hence presumably also harder to learn. Since we already know from our user evaluation that pages above this threshold are highly reliable, we might expect to improve the accuracy of the classifier by training it only with them. We call this subset of reliable pages in each stak the *stak kernel*.

We compared the accuracy (computed with 10-fold cross-validation) of kernel-trained classifiers (using unweighted boolean vectors) with some of our previous classifiers, trained with the whole set of pages. More precisely, we compared it

[3]As a matter of fact, the random classifier performs worse when considering only popular pages. This indicates that the popular pages are not distributed within staks like other pages, or conversely, that the distribution of popularity is not the same in all staks. This should be investigated as an indicator of stak "maturity".

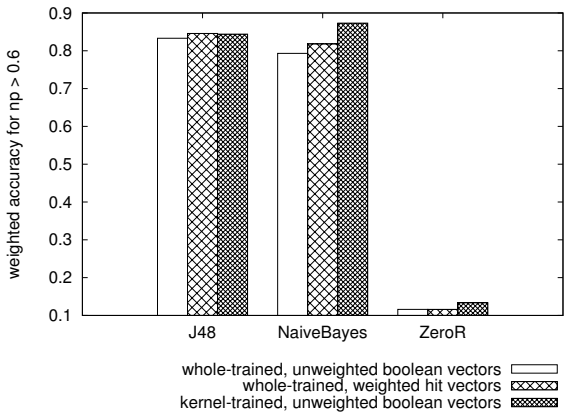

Figure 8: Comparing kernel- and whole-trained classifiers.

with the less informed (*i.e.* using unweighted boolean vectors) and the most informed one (*i.e.* using weighted hit vectors).

The results are reported in Fig. **??**. We see that NaiveBayes becomes better when kernel-trained. The outcome is not as definitive with J48, where the kernel-trained classifier is essentially equivalent to the most informed whole-trained classifier. Our intuition here is that J48 manages to learn from the unpopular relevant pages. The loss of those pages, in kernel-training, is not compensated by the lowering of the noise. This is not the case for NaiveBayes, however. Although this difference needs to be investigated, the fact that NaiveBayes outperforms, when kernel-trained, all of our previous classifiers (including J48) makes us confident in the value of kernel training.

4.4. Off-line Stak Recommendation

The accuracy of the kernel-trained NaiveBayes classifier makes it the best candidate for implementing the off-line phase of our stak recommendation system, as described in Section 3. In this phase, unpopular pages will be candidates for moving or copying to the the three most likely staks according to the classifier. Those pages will also appear as candidates for inclusion on the maintenance pages of those three staks. It is worth noting that in the cross-validation test, the correct stak is present in the top three guesses in 97% of the cases, which makes us very confident in the relevance of this phase for stak owners.

Furthermore, assigning a page to a stak during the maintenance phase is an *explicit* indication from the user that this page is relevant to the stak, unlike the implicit actions mainly used by HeyStaks to fill its experience repository. Such pages can then be considered as part of the stak kernel, regardless of their popularity – recall that unpopular pages are still relevant in 50% of the cases, according to our user evaluation. This may in turn improve the quality of the classifier, reversing the vicious circle introduced in Section 3.

5. Related works

In this paper, we have focused on one kind of noise that we call *misinterpretation*: experience is incorrectly filed, mostly because it is implicitly collected and because the user's behaviour is not always consistent with the idealized behaviour on which the collection process is based. This problem has long been studied in the case-based reasoning literature [**?** **?**], where experience is also collected in a more or less implicit way. With case-base maintenance, however, existing techniques are usually designed to manage case bases with relatively low amounts of noise and work best when relying on an objective measure of when a case can be used to correctly *solve* some target problem. The same kind of approach, applied to recommender systems, is used by [**?**], using the predictions of the system itself as a measure of likelihood. Hence, it relies on a "pristine" system, not *yet* polluted by noisy data, to train the initial classifier. [**?**], on the other hand, introduce a notion of trust to cope with noisy data (associated in this case with untrustworthy users).

Another kind of noise is *malicious noise*: unscrupulous users try to lure the system into recommending items for their own benefit [**?**]. Our notion of popularity is vulnerable to this kind of attack because hits in HeyStaks are anonymous: the popularity of a page can not be traced back to the (potentially malicious) users who selected it. It would seem safer to limit the influence of an individual user on the popularity of each page (even more in the standard workflow of HeyStaks where pages, not staks, are recommended).

A third kind of noise is *opinion drift*. Over time, people may change their mind about their experiences [**?** **?**]. Furthermore, in HeyStaks, once-relevant pages may become outdated, or be modified in a way that makes them less relevant. The problem with our popularity measure is that, once it has become popular, a page will be considered relevant for ever. This can easily be changed though, by applying ageing to our measure: the popularity of a page fades out as its last selection becomes older.

6. Conclusion and discussion

As the Web evolves to accommodate *experiences* as well as pure *content* it will become increasingly important to develop systems and services that help users to manage and harness their online experiences and those of others. In this paper we have focused on experience management in Web search by describing a case-study using the HeyStaks social search engine. HeyStaks is a browser toolbar that works with mainstream search engines such as Google and that allows users to create and share repositories of search experiences (search staks) which then act as a source of search result recommendation.

The main contribution of this work has focused on the nature of the search experiences that HeyStaks harnesses. We have argued that these experiences

can be noisy and that this limits the effectiveness of its search recommendations. As a solution we have argued the need for a meta-recommender system which is designed to recommend search staks, and we have argued that such a recommender can play a key role in supporting stak maintenance and selection. We have described a technique for identifying so-called stak kernels, as the non-noisy essence of stak knowledge – and described and evaluated a classification-based approach to stak recommendation that harnesses these kernels to make accurate stak recommendations.

We have shown that our kernel-trained classifier can be used to implement the off-line stak recommender system described in Section 3. The problem in the case of the online phase, on the other hand, is that we have to deal with *queries* rather than full term vectors. A query is similar to a term vector describing a page, but is a boolean vector (no number of hits, each term is either present or absent), and much sparser (vectors describing pages in HeyStaks combine all the queries used to select the page). Our classifier is already trained with boolean vectors, but we need to perform more tests to determine how well it deals with sparsity, a common problem for recommender systems [**?**].

We believe that our approach, although quite specific to HeyStaks, can be generalized to other recommender systems facing the problem of a noisy knowledge base. Transposing the notion of kernel may not be trivial, since many such systems have a knowledge base reflecting diverse points of view —while in HeyStaks, each stak is relevant to a given topic. However, as each stak has a single kernel, other recommender systems could have, for their single knowledge base, one kernel per user or group of users, allowing them to classify noise in a personnalized way.