Coping with Noisy Search Experiences

Pierre-Antoine Champin[a,b]
Maurice Coyle[b]

Peter Briggs[b]
Barry Smyth[b]

[a] LIRIS, Université de Lyon, CNRS, UMR5205
Université Claude Bernard Lyon 1
F-69622, Villeurbanne, France
http://liris.cnrs.fr
pchampin@liris.cnrs.fr

[b] CLARITY: Centre for Sensor Web Technologies
School of Computer Science and Informatics
University College Dublin, Ireland
http://www.clarity-centre.org
{first.last}@ucd.ie

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. User-generated content, from blogs and wikis to ratings and comments, all add an important layer of experiential knowledge to our online interactions. 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

1. INTRODUCTION

The now familiar *social Web* reflects an important change in the nature of the Web and its content. The arrival of blogs in 1999, 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 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 engine. However, new tools are also needed to gather, harness, reuse and share, in the most efficient and enjoyable way, the experiences captured by UGC [4, 15]. 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 [3, 18]. 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 [12, 8]. Indeed, the work of [13, 1] 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 (www.heystaks.com) and it benefits from providing a collaborative search experi-

ence 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 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 search experiences. At search-time, recommendations are generated from the user's staks and presented alongside mainstream search results. In this way, HeyStaks harnesses the shared experiences of searchers to deliver an improved search experience by working with, rather than competing against, mainstream search engines. With HeyStaks, users search as normal but enjoy the benefits of being able to easily share their search experiences and the advantages of a new form of search collaboration.

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 that searchers must perform before they search: 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 — usually the searcher's default "My Searches" stak, which is not shared with other users. Ultimately this limits the effectiveness of HeyStaks and contributes significant experience noise to search staks.

In what follows 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. First however, we will briefly introduce the HeyStaks system.

2. HEYSTAKS: AN OVERVIEW

HeyStaks adds two important collaboration features to any mainstream search engine. First, it allows users to create *search staks* as a type of folder for their search experiences at search time. Staks can 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 and help the searcher to discover results that friends or colleagues have found interesting, results that may otherwise be buried deep within the engine's result-list.

In designing HeyStaks, our primary goal is to help improve upon the search experience offered by mainstream search engines, while at the same time allowing searchers to continue to use their favourite search engine. As such, a key component of the HeyStaks architecture is a browser toolbar that permits tight integration with search engines such as Google, allowing searchers to search as normal while providing a more collaborative search experience via targeted recommendations. 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 System Architecture

As per Fig. 1, 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. In the following sections we will outline the basic operation of HeyStaks and then focus on some of the detail behind the recommendation engine.

To make things more concrete, consider the following example. Steve, Bill and some friends were planning a European vacation and they knew that during the course of their research they would use Web search as their primary source of information about what to do and where to visit. Steve created a (private) search stak called "European Vacation 2008" and shared this with Bill and friends, encouraging them to use this stak for their vacation-related searches.

Fig. 2 shows Steve selecting this stak as he embarks on a new search for "Dublin hotels", and Fig. 3 shows the results of this search. The usual Google results are shown, but in addition HeyStaks has made two promotions. These were promoted because other members of the "European Vacation 2008" stak had re-

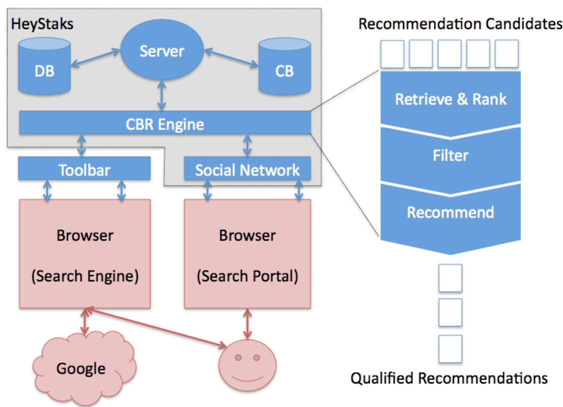

Figure 1: The HeyStaks system architecture and outline recommendation model.

cently 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 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 Steve and Bill benefit from promotions that are based on their previous similar searches. In addition, HeyStaks can recommend results from Steve and Bill'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. 4 shows the portal page for the "European Vacation 2008" 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.

2.2 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

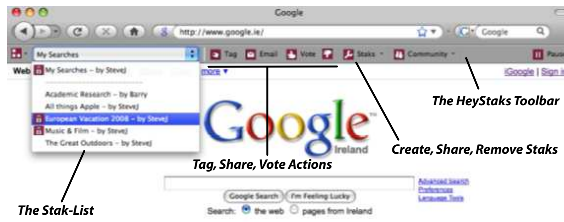

Figure 2: Selecting a new active stak.

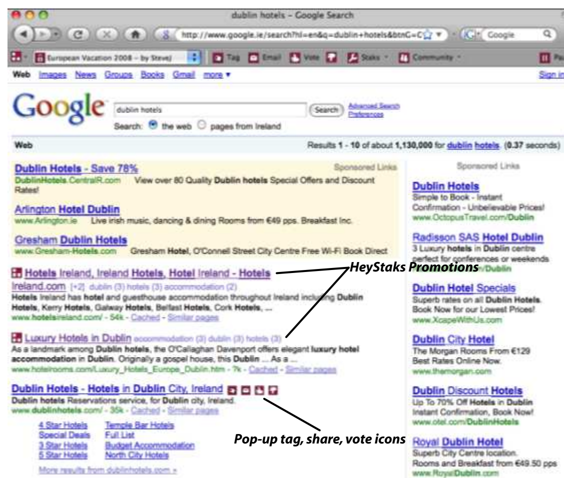

Figure 3: Google search results with HeyStaks promotions.

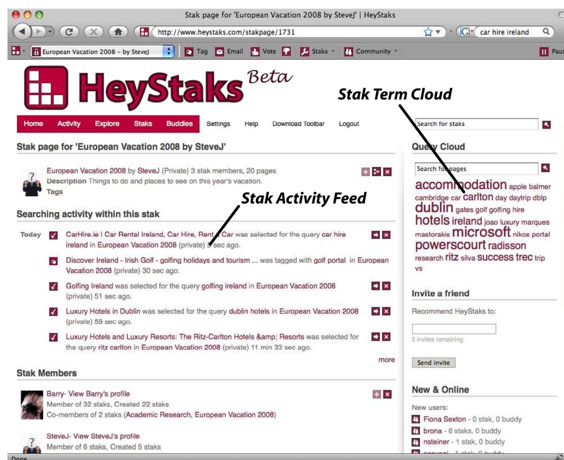

Figure 4: The HeyStaks search portal provides direct access to staks and past searches.

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 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 noisy 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 [14] for more details.

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, 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 the above problem would be for HeyStaks to automatically select, or at least suggest, the appropriate stak when the user starts a query. This is a recommendation problem – recommending a stak – but one that is different from the core recommendation focus, namely the recommendation of search results. We therefore face two challenges: using a repository of recommendation knowledge (search experiences) for another purpose than the one it was designed for, and using it despite the significant amount of noise it contains. Should we succeed, the quality of the collected experiences will increase thanks to the stak recommendation, which in turn will itself be improved: the vicious circle will be reversed to a virtuous circle.

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 intrusive, we believe that stak owners will have an incentive to improve the quality of their staks, especially after experiencing the benefit of relevant recommendations from HeyStaks. 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 that classifier despite

the noise in the available data.

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 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^S_i of the hit matrix, the *hit vector* of page p_i, is how that page will be represented in our classifier.

4.1 Assessing Relevance with Popularity

An immediate approach to assess 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:

$$pop(p_i) \doteq \sum_j h^S_{ij} \qquad pop(t_j) \doteq \sum_i h^S_{ij}$$

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:

$$pop_2(p_i) \doteq \sum_j pop(t_j) \times h^S_{ij}$$

This is illustrated by Fig. 5, which shows that a page like $p1$ with a high number of hits will always be popular, but a set of pages sharing the same terms will also be popular, even if each one of them has a low number of hits (see $p2, p3, p4$).

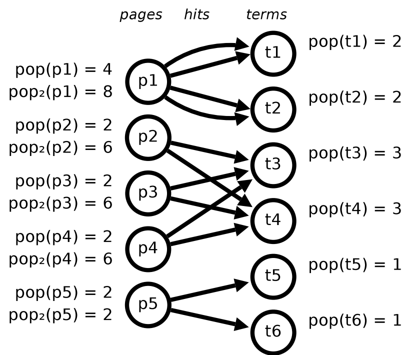

Figure 5: Popularity measure pop_2 illustrated.

We now want to normalize this popularity measure: first, by bringing it between 0 and 1, then by centering the mean popularity to 0.5. This gives us the normalized popularity *np*, which is comparable across staks, regardless of the span or skew of the popularity distribution. The normalized popularity is computed as follows:

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

In order to evaluate the validity of our popularity measure as a predictor of page relevance, we performed a small user evaluation. For each of the 20 staks of our test set, 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.

The results of this evaluations are shown in Fig. 6. 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 can, however, safely assume that the noisy experience is located in the unpopular part of our experience repositories.

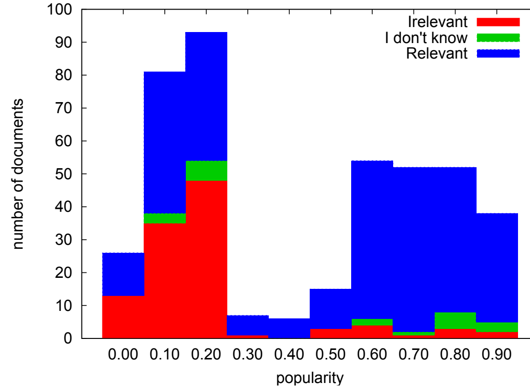

Figure 6: Poll results.

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 instances, 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 [11] 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. 7.

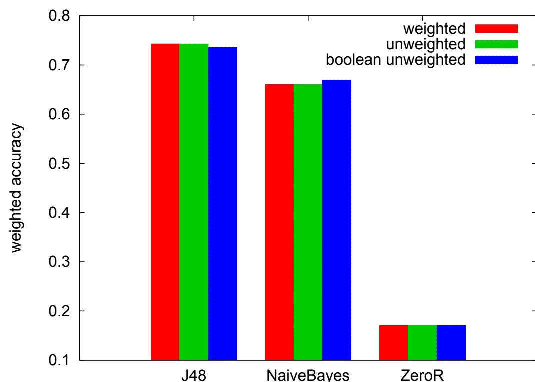

Figure 7: The weighted accuracy is only slightly influenced by the use of popularity in the training set.

This surprising result may be explained by considering how the accuracy varies for pages with different weights. This is shown in Fig. 8, 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.

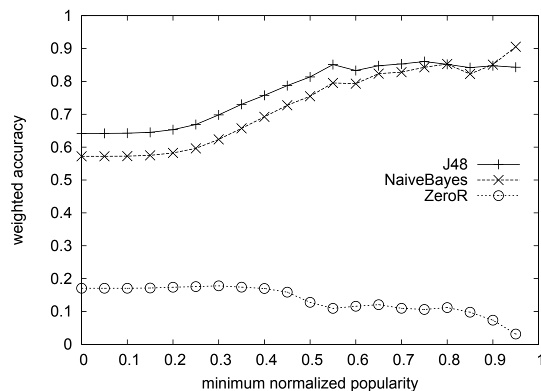

Figure 8: J48 and NaiveBayes classifiers are better at classifying popular pages.

4.3 Stak Kernels

Another interesting thing that Fig. 8 teaches us is that the evolution of the accuracy is not linear. It first stagnates until around 0.3, then increases steadily until around 0.6, then stabilizes again. This seems to indicate that 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 with the less informed (*i.e.* using unweighted boolean vectors) and the most informed one (*i.e.* using weighted hit vectors). Note that we considered the accuracy on pages with *np* > 0.6, even for the whole-trained classifier, for the comparison to be fair.

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

[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".

The results are reported in Fig. 9. We see that NaiveBayes is significantly 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.

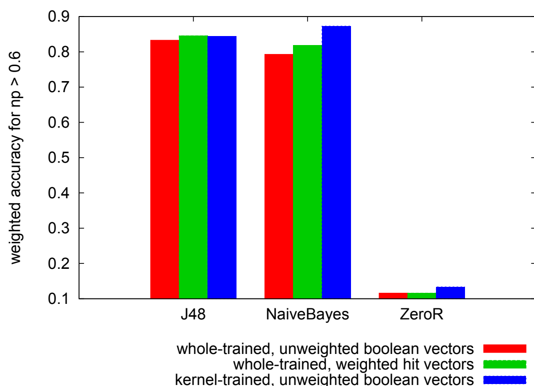

Figure 9: Comparing kernel- and whole-trained classifiers

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 presented to the stak owner along with the three most likely staks according to the classifier. 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. DISCUSSION

In this paper, we have focused on one kind of noise that we call *mis-interpretation*: 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 [16, 17], 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 [10], 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. [9], 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 [6]. 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 [5, 7]. 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.

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). By training our classifier with boolean vectors rather than hit vectors, we solved the first part of the problem. However, we need to perform more tests to determine how the classifier deals with sparsity, a common problem for recommender systems [2].

6. CONCLUSION

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 new form of recommender system which is designed to recommend search staks instead of search pages 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 are now considering using one such classifier to implement the actual recommending system in HeyStaks, in order to improve its stak selection mechanism.

Acknowledgement

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.

7. REFERENCES

[1] P. Briggs and B. Smyth. Provenance, trust, and sharing in peer-to-peer case-based web search. In *ECCBR*, pages 89–103, 2008.

[2] R. Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, 2002.

[3] P. A. Chirita, W. Nejdl, R. Paiu, and C. Kohlschütter. Using odp metadata to personalize search. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 178–185, New York, NY, USA, 2005. ACM.

[4] A. Cordier, B. Mascret, and A. Mille. Extending Case-Based reasoning with traces. In *Grand Challenges for reasoning from experiences, Workshop at IJCAI'09*, July 2009.

[5] I. Koychev and I. Schwab. Adaptation to drifting user's interests. In *Proceedings of ECML2000 Workshop: Machine Learning in New Information Age*, pages 39–46, 2000.

[6] S. K. Lam and J. Riedl. Shilling recommender systems for fun and profit. In *Proceedings of the 13th international conference on World Wide Web*, pages 393–402, New York, NY, USA, 2004. ACM.

[7] S. Ma, X. Li, Y. Ding, M. E. Orlowska, B. Benatallah, F. Casati, D. Georgakopoulos, C. Bartolini, W. Sadiq, and C. Godart. A recommender system with Interest-Drifting. *LECTURE NOTES IN COMPUTER SCIENCE*, 4831:633, 2007.

[8] M. R. Morris. A survey of collaborative web search practices. In *CHI*, pages 1657–1660, 2008.

[9] J. O'Donovan and B. Smyth. Trust in recommender systems. In *Proceedings of the 10th international conference on Intelligent user interfaces*, pages 167–174, San Diego, California, USA, 2005. ACM.

[10] M. P. O'Mahony, N. J. Hurley, and G. C. Silvestre. Detecting noise in recommender system databases. In *Proceedings of the 11th international conference on Intelligent user interfaces*, pages 109–115, Sydney, Australia, 2006. ACM.

[11] J. R. Quinlan. *C4. 5: programs for machine learning*. Morgan Kaufmann, 1993.

[12] M. C. Reddy and P. R. Spence. Collaborative information seeking: A field study of a multidisciplinary patient care team. *Inf. Process. Manage.*, 44(1):242–255, 2008.

[13] B. Smyth. A community-based approach to personalizing web search. *IEEE Computer*, 40(8):42–50, 2007.

[14] B. Smyth, P. Briggs, M. Coyle, and M. O'Mahony. Google? shared! a case-study in social web search. In *Proceedings of the 1st and 17th International Conference on User Modeling, Adaptation and Personalization (UMAP '09)*, Trento, Italy, 2009. Springer.

[15] B. Smyth and P. Champin. The experience web: A Case-Based reasoning perspective. In *Grand Challenges for reasoning from experiences, Workshop at IJCAI'09*, July 2009.

[16] B. Smyth and M. T. Keane. Remembering to forget: A Competence-Preserving case deletion policy for Case-Based reasoning systems. In *IJCAI*, pages 377–383, 1995. Best paper award.

[17] B. Smyth and E. McKenna. Competence models and the maintenance problem. *Computational Intelligence*, 17(2):235–249, 2001.

[18] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen. Cubesvd: a novel approach to personalized web search. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 382–390, New York, NY, USA, 2005. ACM Press.