DataSift: An Expressive and Accurate Crowd-Powered Search Toolkit

Aditya Parameswaran Ming Han Teh Hector Garcia-Molina Jennifer Widom
{adityagp, minghan, hector, widom}@cs.stanford.edu
Stanford University

Abstract

Traditional information retrieval systems have limited functionality. For instance, they are not able to adequately support queries containing non-textual fragments such as images or videos, queries that are very long or ambiguous, or semantically-rich queries over non-textual corpora. In this paper, we present DataSift, an expressive and accurate crowd-powered search toolkit that can connect to any corpus. We provide a number of alternative configurations for DataSift using crowdsourced and automated components, and demonstrate gains of 2–3x on precision over traditional retrieval schemes using experiments on real corpora. We also present our results on determining suitable values for parameters in those configurations, along with a number of interesting insights learned along the way.

1 Introduction

While information retrieval systems have come a long way in the last two decades, modern search engines still have quite limited functionality. For example, they have difficulty with:

1. Non-textual queries or queries containing both text and non-textual fragments: For instance, a query *"cables that plug into <IMAGE>"*, where *<IMAGE>* is a photo of a socket, cannot be handled by any search engine.
2. Queries over non-textual corpora: For instance, a query *"funny pictures of cats wearing hats, with captions"* cannot be handled adequately by any image search engine. Search engines cannot accurately identify if a given image satisfies the query; typically, image search engines perform keyword search over image tags, which may not be sufficient to identify if the image satisfies the query.
3. Long queries: For instance, a query *"find noise canceling headsets where the battery life is more than 24 hours"* cannot be handled adequately by a product search engine. Search results are often very noisy for queries containing more than 3-4 keywords. Most search engines require users to employ tricks or heuristics to obtain meaningful results (MOOC on Searching the Web 2013).
4. Queries involving human judgment: For instance, a query *"apartments that are in a nice area near Somerville"* cannot be handled adequately by an apartment search engine.
5. Ambiguous queries: For instance, a query *"fast jaguar images"* cannot be handled adequately by an image search engine. Search engines cannot tease apart queries which have multiple or ambiguous interpretations, e.g., the car vs. the animal.

For all of these types of queries, currently the burden is on the user to attempt to express the query using the search interfaces provided. Typically, the user will try to express his or her query in as few textual keywords as possible, try out many possible reformulations of the query, and pore over hundreds or thousands of search results for each reformulation. For some queries, e.g., *"buildings that look like <IMAGE>"*, identifying a formulation based solely on text is next to impossible.

Additionally, there are cases where the user does not possess the necessary knowledge to come up with query reformulations. For instance, for the query *"cables that plug into <IMAGE>"*, a particular user may not be able to identify that the socket is indeed a USB 2.0 socket.

To reduce the burden on the user, both in terms of labor (e.g., in finding reformulations and going through results) and in terms of knowledge (e.g., in identifying that a socket is indeed a USB 2.0 socket), we turn to humans (i.e., the crowd) for assistance. In the past few years, crowdsourcing has been incorporated as a component of data processing, gathering, and extraction systems (Park et al. 2012; Franklin et al. 2011; Parameswaran et al. 2012; Bernstein et al. 2010; Zhang et al. 2012). Inspired by these systems, in this paper, we present DataSift, a powerful general-purpose search toolkit that uses humans (i.e., crowd workers) to assist in the retrieval process. Our toolkit can be connected to any traditional corpus with a basic keyword search API. DataSift then automatically enables rich queries over that corpus. Additionally, DataSift produces better results by harnessing human computation to filter answers from the corpus.

Figure 1 shows a high-level overview of DataSift: The user provides a rich search query Q of any length, that may include textual and/or non-textual fragments. DataSift uses an internal pipeline that makes repeated calls to a crowdsourcing marketplace—specifically, Mechanical Turk (Mechanical Turk 2013)—as well as to the keyword search interface to the corpus. When finished, a ranked list of results are presented back to the user, like in a traditional search engine. As an example, Figure 2 depicts the ranked list of results for the query Q = *"type of cable that connects to <IMAGE: USB B-Female socket of a printer>"* over the Amazon products corpus (Amazon Inc. 2013). The ranked results provide relevant USB 2.0 cables with a B-Male connector.

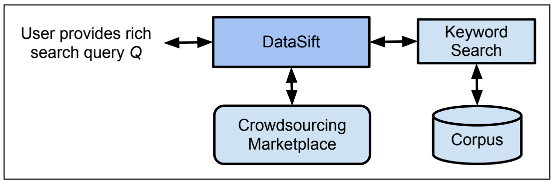


Figure 1: DataSift Overview

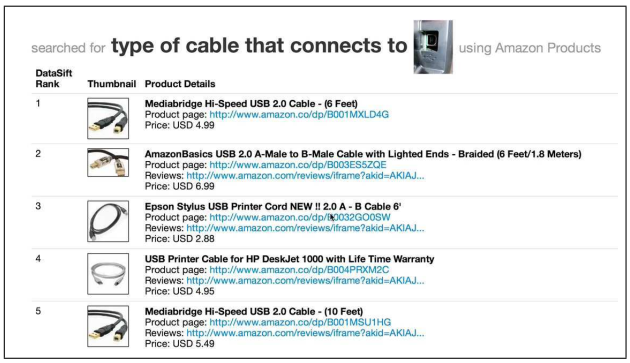


Figure 2: DataSift Example

A disadvantage of our approach is that the response time will be substantially larger than with a traditional search engine. Thus, our approach is only applicable when the user is willing to wait for higher quality results, or when he is not willing or capable of putting in the effort to find items that satisfy his query. Our experience so far is that the wait times are of the order of 20 minutes to an hour. (Note that users case see partial results as they come in.)

We now present some challenges in building DataSift, using our earlier example: Q = *"type of cable that connects to <IMAGE>"*. We assume that we have a product corpus (e.g., Amazon products) with a keyword search API. Consider the following three (among others) possible configurations for DataSift:

- **Gather:** Provide Q to a number of human workers and ask them for one or more reformulated textual keyword search queries, e.g., *"USB 2.0 Cable"* or *"printer cable"*. Then retrieve products using the keyword search API for the reformulated keyword search queries and present the results to the user.
- **Gather-Filter:** The same configuration as **Gather**, but in addition ask human workers to filter the retrieved products for relevance to the query Q, e.g., are they cables that plug into the desired socket, before presenting the results to the user.
- **Iterative Gather-Filter:** The same configuration as **Gather**, but in addition first ask human workers to filter a small sample of retrieved products from each reformulated textual query for relevance to Q, allowing us to identify which reformulations produce better results. Then, tetrieve more from the better reformulations, e.g., more from *"USB 2.0 printer cable"* instead of *"electronic cable"*. Finally, ask human workers to filter the retrieved results before presenting the results to the user.

In addition to determining which configuration we want to use, each of the configurations above has many parameters that need to be tuned. For instance, for the last configuration, DataSift needs to make a number of decisions, including:

- How many human workers should be asked for reformulated keyword search queries? How many keyword search queries should each worker provide?
- How many items should be retrieved initially for each reformulation? How many should be retrieved later (once we identify which reformulations produce better results)?
- How do we decide if a reformulation is better than a different one?
- How many human workers should be used to filter each product for relevance to Q?
- How should the cost be divided between the steps?

Our current implementation of DataSift is powerful enough to be configured to match all of the configurations we have described, plus others. We achieve this flexibility by structuring the toolkit as six plug-and-play components that can be assembled in various ways, described in detail in the next section. In this paper, we present and evaluate a number of alternative configurations for DataSift, and identify good choices for the parameters in each configuration.

To the best of our knowledge, we are the first in addressing the problem of designing a rich general-purpose search toolkit augmented with the power of human computation. By themselves, traditional information retrieval techniques are insufficient for our human-assisted retrieval task. On the other hand, existing crowd-powered systems, including Soylent (Bernstein et al. 2010), Clowder (Dai, Mausam, and Weld 2010), and Turkit (Little et al. 2009) do not address the problem of improving information retrieval. However, we do reuse algorithms from CrowdScreen (Parameswaran et al. 2012) for filtering items using human workers. Unlike social or collaborative search, e.g., (Horowitz and Kamvar 2010; Adamic et al. 2008; Morris and Teevan 2009; Morris, Teevan, and Panovich 2010), we do not leverage the social network, and the system moderates the interaction using reformulations and filtering to ensure high quality results.

Here are the main contributions of the paper:

1. We describe a number of plug-and-play components — automated and crowdsourced — that form the core of DataSift (Section 2).
2. We identify a number of configurations for DataSift using the plug-and-play components (Section 3).
3. We present the current implementation of DataSift, which supports all the described configurations (Section 4).
4. We perform a performance evaluation of these configurations. We show that configurations that use the crowd can

yield 100% more precision than traditional retrieval approaches, and those that ask the crowd for reformulations can improve precision by an additional 100% (Section 5).

5. We optimize the selected configurations, identifying good values for individual parameters (Section 6).

2 Preliminaries and Components

A user enters a query Q into DataSift, which could contain textual and non-textual fragments. Fully textual queries or fully textual reformulations are denoted with the upper case letter T (denoting text). The corpus of items I (products, images, or videos) over which DataSift is implemented has a keyword search API: it accepts a textual keyword search query and a number k, and returns the top k items (products, images, or videos) along with their ranks. The crowdsourcing marketplace M has the following interface: it accepts a task and a number h. It asks h distinct human workers to attempt the task independently, and then returns the h answers to DataSift. DataSift makes repeated calls to both I and M, and then eventually provides the user with n items in ranked order. (In fact, DataSift is flexible enough to provide the user with a dynamically updated ranking of items that is kept up-to-date as DataSift evaluates the items.)

Next, we describe the components internal to DataSift. Components are categorized into: (1) Crowdsourced Components: components that interact with the crowdsourcing marketplace, and (2) Automated Components: components that function independent of the crowdsourcing marketplace. The function signatures of the components are provided in Table 1. Note that the query Q and the corpus of items I are implicit input arguments in the signatures for all these components.

2.1 Crowdsourced Components

- (G) **Gather Component**: $h, s \rightarrow \{T\}$

  The Gather Component G asks human workers for fully textual reformulations for Q, providing DataSift a mechanism to retrieve items (using the keyword search API) for Q (recall that Q may contain non-textual fragments).

  Given a query Q, G uses the marketplace M to ask h human workers for s distinct textual reformulations of Q each, giving a total of $h \times s$ textual queries. Specifically, human workers are asked to respond to the following task: "Please provide s reformulated keyword search queries for the following query:" Q. The human workers are also able to run the reformulated query on the corpus I to see if the results they get are desirable.

- (F) **Filter Component**:

  $\{(I, T, rank)\}, t \rightarrow \{(I, T, p, n)\}$

  The input to the Filter Component F is a set of items. We will ignore T and $rank$ for now, these parameters are not used by F. For each item i in the set of items, F determines whether the item satisfies the query Q or not. The component does this by asking human workers to respond to the following task: "Does the item i satisfy query Q: (Yes/No)". Since human workers may be unreliable, multiple workers may be asked to respond to the same task on the same item i. The number of humans asked is determined by designing a filtering strategy (Parameswaran et al. 2012) using the overall accuracy threshold t. The number of positive responses for each item is denoted p, while the number of negative responses is denoted n.

  In the input, each item i is annotated with T and $rank$: T is the textual query T whose keyword search result set item i is a part of. $rank$ is the rank of i in the keyword search results for T. Both these annotations are provided as part of the output of the keyword search API call – see component R). $T, rank$ are part of the input for compatibility with the calling component, and T is part of the output for compatibility with the called component.

2.2 Automated Components

- (R) **Retrieve Component**:

  $\{T\} \mid \{(T, w)\},\ k \ \rightarrow\ \{(I, T, rank)\}$

  The Retrieve Component R uses the keyword search API to retrieve items for multiple textual queries T from the corpus. For each textual query T, items are retrieved along with their keyword search result ranks for T (as assigned in the output of the keyword search API call).

  Specifically, given a set of textual queries T_i along with weights w_i, R retrieves k items in total matching the set of queries in proportion to their weights, using the keyword search API. In other words, for query T_i, the top $k \times \frac{w_i}{\sum_j w_j}$ items are retrieved along with their ranks for query T_i. If the weights are not provided, they are all assumed to be 1. We ignore for now the issue of duplicate items arising from different textual queries; if duplicate items arise, we simply retrieve additional items from each T_i in proportion to w_i to make up for the duplicate items.

- (S) **Sort Component**:

  The Sort Component S has two implementations, depending on which component it is preceded by. Overall, S merges rankings, providing a rank for every item based on how well it addresses Q.

  $\{(I, T, p, n)\} \rightarrow \{(I, rank)\}$

  If preceded by the F component, then S receives as input items along with their textual query T, as well as p and n, the number of Yes and No votes for the item. Component S returns a rank for every item based on the difference between p and n (higher $(p - n)$ gets a higher rank); ties are broken arbitrarily. The input argument corresponding to the textual query T that generated the item is ignored.

  $\{I, T, rank\} \rightarrow \{(I, rank)\}$

  If preceded by the R component, then S receives as input items along with their textual query T, as well as $rank$, the rank of i in the result set of T. Component S simply ignores the input argument corresponding to T, and merges the ranks; ties are broken arbitrarily. For example, if $(a, T_1, 1), (b, T_1, 2), (c, T_2, 1), (d, T_2, 2)$ form the input, then one possible output is: $(a, 1), (b, 3), (c, 2), (d, 4)$; yet another one is: $(a, 1), (b, 4), (c, 2), (d, 3)$.

- (W) **Weighting Component**: $\{(I, T, p, n)\} \rightarrow \{(T, w)\}$

  For Iterative Gather-Filter (Section 1), the weighting component is the component that actually evaluates reformulations. The component always follows F, using the results from F to compute weights corresponding to how good different reformulations are in producing items that address Q.

  Component W receives as input items from the Filter Component F, annotated with p and n (the number of Yes and No votes), and the textual query T that generated the items. For each textual query T, given the output of the filtering component F, the weighting component returns a weight based on how useful the textual query is in answering Q.

  There are three variants of W that we consider: W_1, W_2, and W_3, corresponding to three different ways in which weights w_i are assigned to T_i. For describing these variants, for convenience, we introduce two new definitions for the output of F: for a given item, if $p > n$, then we say that the item belongs to the *pass set*, while if $n \geq p$, then we say that the item belongs to the *fail set*.

  - W_1: For each textual reformulation T_i, we set w_i to be the number of items (from that reformulation) in the pass set.
  - W_2: Unlike W_1, which accords non-zero weight to every reformulation with items in the pass set, W_2, preferentially weights only the best reformulation(s). Let the size of the pass set for T_i be x_i, and let $X = \max_i(x_i)$. For each reformulation T_i that has $x_i = X$, we assign the weight $w_i = 1$. Otherwise, we assign the weight $w_i = 0$.
  - W_3: Each reformulation is weighted on how much agreement it has with other reformulations based on the results of F. For instance, if reformulation T_1 has items $\{a, b\}$, T_2 has $\{b, c\}$, and T_3 has $\{a, d\}$ as ranks 1 and 2 respectively, then T_1 is better than T_2 and T_3 since both items a and b have support from other reformulations.

    For the i-th reformulation, we set w_i to be the sum, across all items (from that reformulation), the number of other reformulations that have that particular item. Thus: ($\mathcal{I}$ stands for the indicator function)

    $$w_i = \sum_{\forall a \text{ from } T_i} \sum_{j \neq i} \mathcal{I}(a \text{ is in } T_j\text{'s results}),$$

3 Configurations

We now describe the DataSift configurations that we evaluate in this paper. The goal of each configuration is to retrieve n items in ranked order matching query Q. Some configurations may retrieve $n' \geq n$, and return the top n items.

Given that the components described in the previous section are plug-and-play, there is a large number of configurations that we could come up with; however, we focus our attention on a few that we have found are the most interesting and important:

	Signature	Followed by
G	$h, s \rightarrow \{T\}$	R
F	$\{(I, T, rank)\},\ t \rightarrow \{(I, T, p, n)\}$	W, S
R	$\{T\} \mid \{(T, w)\},\ k \rightarrow \{(I, T, rank)\}$	F, S
S	$\{(I, T, p, n)\} \mid \{I, T, rank\} \rightarrow \{(I, rank)\}$	—
W	$\{(I, T, p, n)\} \rightarrow \{(T, w)\}$	R

Table 1: Components, their function signatures (Q and I are implicit input parameters in all of these functions), and other components that can follow them.

- RS: (Only possible if Q is textual) This configuration refers to the traditional information retrieval approach: component R uses the query Q to directly retrieve the top n items with ranks using the keyword search API. In this case, component S does nothing, simply returning the same items along with the ranks.
- RFS: (Only possible if Q is textual) From the $n' \geq n$ items retrieved by component R, component F uses humans to better identify which items are actually relevant to the query Q. Component S then uses the output of F to sort the items in the order of the difference in the number of Yes and No votes for an item as obtained by F, and return n items along with their ranks.
- GRS: (Gather from Section 1) Component G gathers textual reformulations for Q, asking h human workers for s reformulations each. Subsequently, R retrieves the top $n/(hs)$ items along with ranks for each of these $h \times s$ reformulations. Then, S sorts the items by simply merging the ranks across the $h \times s$ reformulations, with ties being broken arbitrarily. Items are returned along with their ranks.
- GRFS: (Gather-Filter from Section 1) Component G gathers $h \times s$ textual reformulations, after which component R retrieves $n'/(hs)$ items with ranks for each of the reformulations. Then, component F filters the n' items using human workers. Subsequently, the n' items are sorted by component S based on the difference in the number of Yes and No votes for each item, and the top n are returned along with their ranks; ties are broken arbitrarily (the input argument corresponding to the textual reformulation is ignored).
- GRFW_iRFS for $i = 1, 2, 3$: (Iterative Gather-Filter from Section 1) Component G gathers $h \times s$ textual reformulations, after which component R retrieves δ items from each of the reformulations (δ is a small sample of results from each reformulation, typically much smaller than n). Component F then filters the set of $\delta \times h \times s$ items. The output of F provides us with an initial estimate as to how useful each reformulation is in answering the query Q.

  Subsequently, component W (either W_1, W_2, or W_3) computes a weight for each of the textual reformulations based on the results from F. These weights are then used by component R to preferentially retrieve $n' - \delta \times h \times s$ items in total across reformulations in proportion to the weight. Component F filters the retrieved items once again. Eventually, the component S sorts the items in the order of the difference between the number of Yes and No votes (ignoring the input argument corresponding to the reformulation, and breaking ties arbitrarily), and returns the items along with their ranks.

For now, we consider only $\mathsf{GRFW_1RFS}$ (and not $\mathsf{W_2}$ or $\mathsf{W_3}$), which we refer to as GRFWRFS. We will consider other variants of W in Section 6.

4 Implementation

We provide a very brief overview of the DataSift implementation followed by details regarding the crowdsourced components.

DataSift is implemented in Python 2.7.3 using Django, the popular web application development library. We use Amazon's Mechanical Turk (Mechanical Turk 2013) as our marketplace M. We leverage the Boto library (Boto Web Services Library 2013) to connect to Mechanical Turk, and the Bootstrap library (Twitter Bootstrap 2013) for front-end web templates. A complete trace of activity from previous queries on DataSift, along with the results, are stored in a MySQL 5 database. The current version of DataSift connects to four corpora: Google Images (Google Images 2013), YouTube Videos (Google Inc. 2013b), Amazon Products (Amazon Inc. 2013), and Shutterstock Images (Shutterstock Inc. 2013).

We now provide specifics regarding the implementation of the crowdsourced components:

- (G) **Gather Component**:

  Using the search query Q provided, component G issues Amazon Mechanical Turk Human Intelligence Tasks (HITs) which solicits textual reformulations from human workers. Here, we discuss some of the design features of the HIT seen by a human worker.

  Humans are allowed to refine their reformulations before submission by using a "test search" feedback loop to probe their reformulations against the corpus. The top ten results from the corpus for the corresponding reformulation are displayed to the human worker so that he or she can decide if the results are satisfactory.

  Initially, we discovered that instructions were unclear, due to which humans were providing reformulations that were simple paraphrasing of the search predicate. To gather better reformulations, we added examples to the instructions to allow human workers to provide reformulations based on contextual knowledge. One such example we provided hinted that a possible reformulation for Q = *"SF bridge; night scene"* might be *"golden gate bridge night"*. In addition, we extracted and included tips from Google's "Basic Search Help" (Google Inc. 2013a) to enable better reformulations.

  The gather component prevents human workers from submitting duplicate reformulations for the same query Q, and distinct humans from submitting identical reformulations. While it might be possible to make inferences about the relevance of a reformulation given duplicate reformulations, we hypothesized that it is best to not have duplicates to ensure diversity and maximize utility, and then have downstream components take care of eliminating less relevant reformulations later.

- (F) **Filter Component**:

  Recall that the filter component F takes as input a set of items I, and uses human workers to check if items satisfy query Q. As a first step, for every item, F asks $k = 3$ human workers to verify if the item satisfies query Q. Subsequently, these $|I| \times 3$ answers are used to learn the probabilities of human error (i.e., the probability that an item satisfies Q, but a human answers otherwise, and vice versa), and the a-priori probability of an item satisfying Q. Unfortunately, since DataSift is completely unsupervised, we do not know the true values for items. Instead, we use a simple heuristic to infer true values of items: if the number of Yes answers is greater than the number of No answers, then we assume that the item satisfies Q, and that the item does not satisfy Q otherwise. Using the true values, we can learn the probabilities of human error and a-priori probability. These quantities are then input to the strategy learning module (Parameswaran et al. 2012), which outputs a strategy that determines how many additional answers are needed (beyond the three initial ones) for each item. For instance, the strategy might output that if 3 Yes answers have been obtained for an item, then no additional answers are needed, but if 2 Yes and 1 No answers have been obtained, then one more question needs to be asked.

  To reduce cost and improve accuracy, to each human worker, we provide a batch of items to be evaluated at the same time (i.e., whether or not they satisfy Q) as one HIT on Mechanical Turk. We set our batch size to be 50 to provide workers the ability to have enough items to compare, and yet not be fatigued by the task. To reduce bias, we randomized the assignment of items to batches, as well as the ordering of items within a batch.

5 Initial Evaluation on Textual Queries

We perform an initial evaluation of the configurations described in Section 3. Specifically, we assess how much benefit we can get from using various crowd-powered configurations over the traditional fully-automated retrieval approach (RS). Since rich media queries are simply not supported by traditional retrieval approaches, for our initial comparison, we focus on fully textual queries. (We consider rich media queries in the next section.)

Setup: We hand-crafted a set of 20 diverse textual queries (some shown in Table 2). We executed these 20 queries using each of four configurations RS, RFS, GRS, GRFWRFS on the Google Images corpus. For each of the configurations, we set n', i.e., the total number of items retrieved, to be 50. For both GRS and GRFWRFS, we used $h = w = 3$ and for GRFWRFS, we used $\delta = 3$.

Evaluation: To evaluate the quality of the ranked results, we measure the fraction of true positives in the top-n items, i.e., the number of items in the top n satisfying Q divided by n. Note that this quantity is precisely *precision@n*. To determine the number of true positives, we manually inspected

Easy Queries (5)
funny photo of barack obama eating things
bill clinton waving to crowd
matrix digital rain
eiffel tower, paris
5 x 5 rubix cube
Hard Queries (5)
tool to clean laptop air vents
cat on computer keyboard with caption
handheld thing for finding directions
the windy city in winter, showing the bean
Mitt Romney, sad, US flag
Selected Others
funny photos of cats wearing hats, with captions
the steel city in snow
stanford computer science building
database textbook

Table 2: List of textual queries for initial evaluation

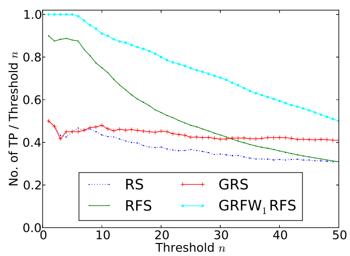


Figure 3: Precision curve

the results, carefully checking if each item returned satisfies the query Q or not.

Basic Findings: Our results can be found in Figure 3. We plot the fraction of true positives in the top-n result set for each of the configurations, on varying the threshold n. As an example, for threshold $n = 30$, GRS and RFS have precision 0.4 (i.e., 0.4 * 30 = 12 items satisfy Q on average from the top 30), while RS has precision 0.35, and GRFWRFS has precision 0.7, 100% higher than the precision of RS. Therefore, sophisticated configurations combining the benefits of the crowdsourced components F and G perform much better than those with just one of those components, and perform significantly better than fully automated schemes.

Notice that the configuration RFS is better than GRS for smaller n. Configuration RFS retrieves the same set of items as RS, but the additional crowdsourced filter F component ensures that the items are ranked by how well they actually satisfy Q. Configuration GRS on the other hand, gathers a number of reformulations, ensuring a diverse set of retrieved items. However, the items may not be ranked by how well they actually satisfy Q – the good items may in fact be lower ranked. As a result, for smaller n, RFS does better, but GRS does better for larger n.

In addition, we plotted the precision curve with error bars in Figure 4. Unlike the other configurations, the error bars for GRFWRFS are initially zero and increase with n, indicating that GRFWRFS produces consistently relevant results for the top 5 items. GRS has initial error bars that are higher than the rest because the top results might be relevant to one of the reformulations but not to Q. However, as more results are considered, its easier to find items that are relevant to the original query Q.

Summary: Crowd-powered configurations RFS, GRS, *and* GRFWRFS *outperform* RS. GRFWRFS *clearly does the best, with 50-200% higher precision than* RS *on average, followed by* GRS. RFS *is better than* GRS *for smaller n due to* F, *but* GRS *does better for larger n.*

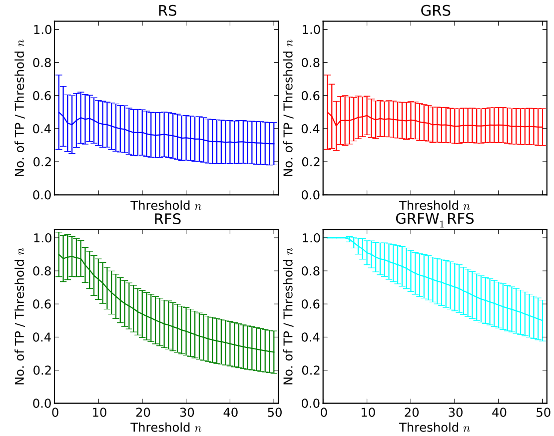


Figure 4: Precision curve with error bars showing 95% confidence interval

Query Difficulty: To study the impact of query difficulty on results, we ordered our queries based on the number of true positive results in the top 10 results using the traditional retrieval approach. We designated the top 5 and the bottom 5 queries as the *easy* and the *hard* queries respectively — see Table 2 for the list of queries in each category. We then plotted the fraction of true positives on varying n for each category. We depict the results in Figure 5. The general trend is consistent with Figure 3 except that for easy queries, RFS and RS outperforms GRS. This somewhat counterintuitive result makes sense because for easy queries, most of the results from the traditional retrieval approach are already good, and therefore it is more beneficial to use the filter component rather than the gather component. In fact, the gather component may actually hurt performance because the reformulations may actually be worse than the original query Q. On the hard queries, GRFWRFS performs significantly better than the other configurations, getting gains of up to 500% on precision for small n.

Summary: Crowd-powered configurations RFS *and* GRFWRFS *outperform* RS *even when restricted to very hard or easy queries. However, the benefits from using crowd-powered configurations is more evident on the hard queries.*

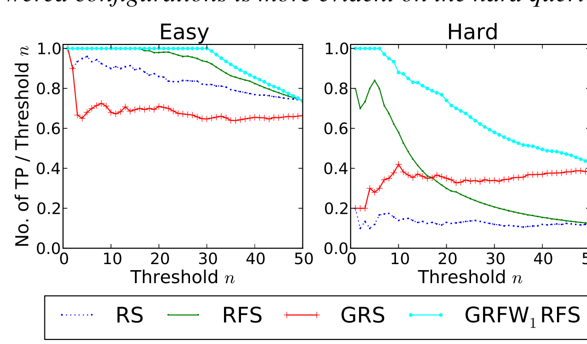


Figure 5: Precision curves for easy vs hard queries

Rich Queries (5)
buildings around <IMAGE: UC Berkeley's Sather Tower>
device that reads from <IMAGE: Iomega 100MB Zip Disk>
where to have fun at <IMAGE: Infinity Pool at Marina Bay Sands hotel in Singapore>
tool/device that allows me to do hand gestures such as in: <VIDEO: motion sensing demonstration using fingers >
type of cable that connects to <IMAGE: USB B-Female socket of a printer>

Table 3: List of Rich Queries

6 Rich Queries and Parameter Tuning

We now describe our results on running the sophisticated configurations on rich media queries, and also describe our experiments on choosing appropriate values for parameters for the sophisticated configurations. For both these objectives, we generated a test data-set in the following manner:

Data Collection: We constructed 10 queries: 5 (new) fully textual queries and 5 queries containing non-textual fragments — that we call *rich* queries. (See Table 3 for the list of rich queries.) For each query, we gathered 25 reformulations (5 human workers $\times$ 5 reformulations per worker), then retrieved a large (>100) number of items for each reformulation, and filtered all the items retrieved using crowdsourced filter component F. This process actually provided us with enough data to simulate executions of all configurations (described in Section 3) on any parameters $h, s \le 5, n' \le 100$. Moreover, by randomizing the order of human participation in G, we can get multiple executions for a fixed configuration with fixed parameters. That is, if we have $h = 3$, then we get $\binom{5}{3}$ simulated executions by allowing G to get reformulations from any 3 workers out of 5.

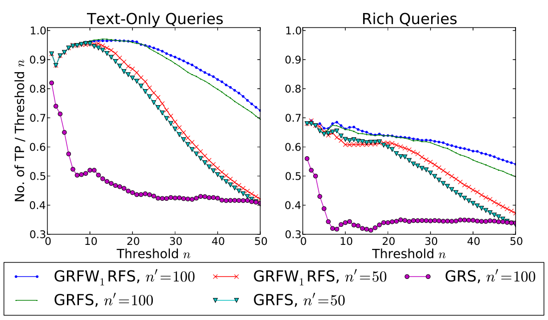


Figure 6: Curves for: (a) textual queries (b) rich queries

Monetary Costs: So far, while we have compared the configurations against each other on precision, these configurations actually have different costs. We tabulate the costs for each configuration in symbolic form in Table 4. In addition to precision, we will use the costs described above to compare configurations in subsequent experiments.

Basic Findings: We first study the differences in performance of DataSift configurations on rich queries and on textual queries. We set $h = s = 3, \delta = 1$, and simulated the execution of configurations GRS (for $n' = 50$), GRFS (for $n' = 50, 100$), GRFWRFS (for $n' = 50, 100$). We plot the the average fraction of true positives in the top n, divided by n, on varying n from $1 - 50$, for textual queries in Figure 6(a) and for rich queries in Figure 6(b). As can be seen in the two figures, the relative positions of the five configurations are similar in both figures.

Configuration	Cost
RS	Free
RFS	$n' \times \tau \times C_1$
GRS	$h \times s \times C_2$
GRFS	$h \times s \times C_2 + n' \times \tau \times C_1$
GRFWRFS	$h \times s \times C_2 + n' \times \tau \times C_1$

Table 4: Breakdown of monetary costs associated with each configuration. τ is the expected number of human workers used to filter the item. C_1 is the cost of asking for a reformulation and C_2 is the cost of getting a human worker to filter a single item. Typical values are $C_1 = \$0.003$ (for images), $C_2 = \$0.10$, $\tau = 4$.

We focus on the rich queries first (Figure 6(b)). As can be seen in the figure, for $n' = 50$, GRFWRFS has higher precision than GRFS (with the differences becoming more pronounced for larger n), and much higher precision than GRS. For instance, for $n = 50$, GRFWRFS has 15% higher precision than GRS and GRFS — the latter two converge at $n = 50$ because the same set of $n' = 50$ items are retrieved in both configurations. For $n' = 100$, GRFWRFS has higher precision than GRFS and GRS, as well as the plots for $n' = 50$. For instance, for $n = 50$, GRFWRFS with $n' = 100$ has close to 100% higher precision than GRS, and close to 50% higher precision than GRFWRFS with $n' = 50$. This is not surprising because retrieving more items and filtering them enables us to have a better shot at finding items that satisfy Q (along with ordering them such that these items are early on in the result set). We study the behavior relative to n' in more detail later on. GRS continues to perform similarly independent of the items n' retrieved since only the top n items are considered, and since $n' \ge n$.

Recall that GRFWRFS has the same cost as GRFS (Table 4). Thus, GRFWRFS strictly dominates GRFS in terms of both cost and precision. On the other hand, GRFWRFS may have higher cost than GRS, but has higher precision.

We now move back to comparing text and rich queries. As can be seen in the two figures, the gains in precision for textual queries from using more sophisticated configurations are smaller than the gains for rich queries. Moreover, the overall precision for the rich queries (for similar configurations) is on average much lower than that for text-only queries; not surprising given that the rich queries require deeper semantic understanding and more domain expertise.

Summary: On average, the relative performance of DataSift *configurations is similar for both textual and rich queries, with lower precision overall for rich queries, but higher gains in precision on using sophisticated configurations. For both textual and rich queries, on fixing the total number of items retrieved n' and the number of reformulations,* GRFWRFS *does slightly better than* GRFS*, and does significantly better than* GRS*. For individual queries, the gains from using* GRFWRFS *may be even higher. On increasing the number of items retrieved,* GRS *continues to performs similarly, while* GRFS *and* GRFWRFS *both do even better.*

Optimizing GRFWRFS: Previously, we have found that of the configurations considered so far, GRFWRFS provides the best precision. We now focus our attention on optimizing the parameters of GRFWRFS for even better precision. Specifically, we try to answer the following questions:

1. How do the variations of W_i, $i = 1, 2, 3$ perform against each other?
2. How do the number of human workers (h) and number of reformulations per worker (s) affect the results?
3. How should the sample size δ (used to evaluate the reformulations) be determined?
4. How does the number of target items n' affect precision?

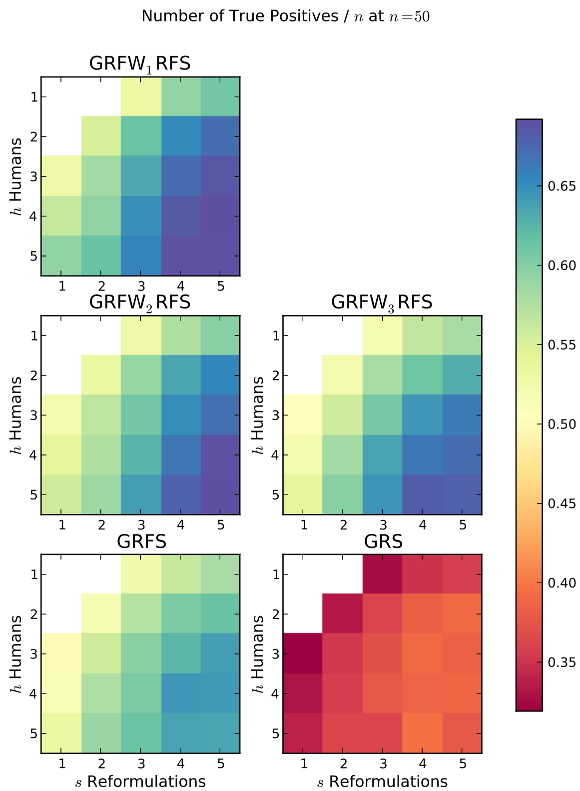


Figure 7: Heat map of no. of true positives for the top 50 items. Each configuration uses $\delta = 3, n' = 100$. The 3 white-colored cells on the top left in each grid are masked due to insufficient data. *Note: view this figure in color!*

Questions 1 and 2: Varying h, s and Varying W_{1-3}: We simulate the five configurations: GRS, GRFS, GRFW$_{1-3}$RFS on the 10 textual and rich queries, for $n' = 100, \delta = 3$. (Similar results are seen for other parameter settings.) We depict the fraction of true positives in the top-50 on varying h, s, as a heat map in Figure 7. In general, GRFW$_{1-3}$RFS has a higher number of true positives than GRFS, and GRFS has a higher number of true positives than GRS. We see a clear trend across rows and across columns: fixing one dimension while increasing the other increases the fraction of true positive results. For the 3 GRFW$_i$RFS configurations, having 1 worker with 5 reformulations outperforms 5 workers with 1 reformulation each; additionally, recreating the benefits of two workers with four reformulations each (a total of 8) requires at least five workers with three or more reformulations each (a total of 15). These results indicate that forcing more reformulations from a human workers prompts them to think deeper about Q, and provide more useful reformulations overall. We see diminishing returns beyond three workers providing five reformulations each.

Summary: W_1 *performs marginally better than W_2 and W_3. The precision improves as we increase h and s for all configurations, however having fewer human workers providing more reformulations each is better than more human workers providing fewer reformulations each.*

Question 3: Varying δ (size of retrieved sample) in GRFW$_{1-3}$RFS: We fixed $n' = 100$, and plotted the number of true positives in the top 50 items as a function of the number of the number of items sampled δ. The results are displayed for $h = s = 5$ in Figure 8a, and for $h = s = 3$ in Figure 8b.

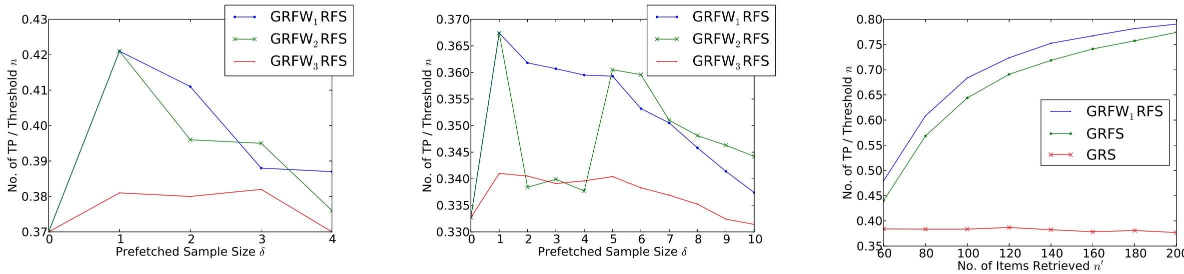
We focus first on $h = s = 5$. Since the total number of items retrieved n' is fixed, there is a tradeoff between *exploration* and *exploitation*: If $\delta = 1$, then a total of $h \times s \times \delta = 25$ items are sampled and evaluated, leaving us $n' - 25 = 75$ items for the second phase of retrieval. On the other hand, if $\delta = 3$, then a total of $h \times s \times \delta = 75$ items are sampled and evaluated — giving us a better estimate of which reformulations are good, however, we are left with only $n' - 75 = 25$ items to retrieve from the good reformulations. With $\delta = 1$, we do very little exploration, and have more budget for exploitation, while with $\delta = 3$, we do a lot of exploration, and as a result, have less budget for exploitation.

Figure 8a depicts the effects of exploration versus exploitation: the number of true positives for all three plots increases as δ is increased, and then decreases as δ goes beyond a certain value. When $\delta = 0$, the configurations are identical to one another and have the same effect as GRFS. Increasing δ by 1 gives a ≈15% improvement in precision of results with the exception of GRFW$_3$RFS. GRFW$_3$RFS (which uses a weighting component based on the agreement across reformulations) shows a dome-shaped curve which peaks at 1-3 items. As δ is increased further, the number of true positives decreases as n' is wasted on exploration rather than exploitation.

The results in Figure 8b are similar, however, GRFW$_2$RFS's trend is erratic. This is because taking the single best-looking reformulation may not be a robust strategy when using smaller h and s. For $\delta = 0$ and 1, for both figures, the number of true positives for GRFW$_2$RFS is similar to GRFW$_1$RFS. This is expected since the weighting approach used is similar in practice for the two corner cases.

Summary: On fixing the total number of items retrieved n', retrieving and filtering a sample of $\delta = 1$ items from each reformulated query is adequate to find the best queries from which to retrieve additional items.

Question 4: Varying Target Number of Items n': Figure 8c shows the effect of varying the number of retrieved items n' on the number of true positives in the top 50 items. We use $h = s = 4$ for each configuration, and $\delta = 3$ for GRFWRFS. As is evident from the plot, GRS is unable to effectively utilize the additional items retrieved when n' is increased. On the other hand, we see a positive trend with the other two configurations, with diminishing returns as n'



(a) (b) (c)

Figure 8: (a) Effect of varying sampled items δ in GRFW$_{1-3}$RFS. Using $n' = 100, n = 100, h = s = 5$ (b) Effect of varying sampled items δ in GRFW$_{1-3}$RFS. Using $n' = 100, n = 100, h = s = 3$. (c) Effect of varying target number of items n'

increases. Note that for GRFS and GRFWRFS cost is directly proportional to n' (ignoring a fixed cost of gathering reformulations) — see Table 4 — so the figure still holds true if we replace the horizontal axis with cost.

Summary: The fraction of true positives increases as n' increases, with diminishing returns.

7 Conclusion

We presented DataSift, a crowd-powered search toolkit that can be instrumented easily over traditional search engines on any corpora. DataSift is targeted at queries that are hard for fully automated systems to deal with: rich, long, or ambiguous queries, or semantically-rich queries on non-textual corpora. We presented a variety of configurations for this toolkit, and experimentally demonstrated that they produce accurate results — with gains in precision of 100-150% — for textual and non-textual queries in comparison with traditional retrieval schemes. We identified that the best configuration is GRFW$_1$RFS, and identified appropriate choices for its parameters.

As future work, we plan to incorporate user intervention during execution as feedback to the DataSift toolkit, enabling a more adaptive execution strategy. In addition, while we already enable users to view partial results as they are computed, we plan to focus on optimizing DataSift to generate partial results. Finally, we plan to investigate the benefits of adding a crowdsourced ranking component in place of the filtering component.

Acknowledgements

We thank Akash Das Sarma for help with the implementation, and Alkis Polyzotis for initial discussions.

References

Adamic, L. A.; Zhang, J.; Bakshy, E.; and Ackerman, M. S. 2008. Knowledge sharing and yahoo answers: everyone knows something. In *WWW*.

Amazon Inc. 2013. http://www.amazon.com.

Bernstein, M. S.; Little, G.; Miller, R. C.; Hartmann, B.; Ackerman, M. S.; Karger, D. R.; Crowell, D.; and Panovich, K. 2010. Soylent: a word processor with a crowd inside. In *UIST*, 313–322.

Boto Web Services Library. 2013. https://github.com/boto/boto.

Dai, P.; Mausam; and Weld, D. S. 2010. Decision-theoretic control of crowd-sourced workflows. In *AAAI*.

Franklin, M. J.; Kossmann, D.; Kraska, T.; Ramesh, S.; and Xin, R. 2011. Crowddb: answering queries with crowdsourcing. In *SIGMOD*.

Google Images. 2013. http://images.google.com.

Google Inc. 2013a. Basic search help.

Google Inc. 2013b. http://www.youtube.com.

Horowitz, D., and Kamvar, S. D. 2010. The anatomy of a large-scale social search engine. In *WWW*.

Little, G.; Chilton, L. B.; Goldman, M.; and Miller, R. C. 2009. Turkit: tools for iterative tasks on mechanical turk. In *HCOMP*.

Mechanical Turk. 2013. http://www.mturk.com.

MOOC on Searching the Web. 2013. Google Inc.

Morris, M. R., and Teevan, J. 2009. Collaborative web search: Who, what, where, when, and why. *Synthesis Lectures on Information Concepts, Retrieval, and Services* 1(1):1–99.

Morris, M. R.; Teevan, J.; and Panovich, K. 2010. What do people ask their social networks, and why?: a survey study of status message q-a behavior. In *CHI*.

Parameswaran, A.; Garcia-Molina, H.; Park, H.; Polyzotis, N.; Ramesh, A.; and Widom, J. 2012. Crowdscreen: Algorithms for filtering data with humans. In *SIGMOD*.

Park, H.; Pang, H.; Parameswaran, A.; Garcia-Molina, H.; Polyzotis, N.; and Widom, J. 2012. Deco: A system for declarative crowdsourcing. In *VLDB*.

Shutterstock Inc. 2013. http://www.shutterstock.com.

Twitter Bootstrap. 2013. twitter.github.com/bootstrap/.

Zhang, H.; Law, E.; Miller, R.; Gajos, K.; Parkes, D. C.; and Horvitz, E. 2012. Human computation tasks with global constraints. In *CHI*, 217–226.