Estimating the ImpressionRank of Web Pages

Ziv Bar-Yossef
Dept. of Electrical Engineering
Technion, Haifa 32000, Israel
and
Google Haifa Engineering Center, Israel
zivby@ee.technion.ac.il

Maxim Gurevich[*]
Dept. of Electrical Engineering
Technion, Haifa 32000, Israel
gmax@tx.technion.ac.il

ABSTRACT

The ImpressionRank of a web page (or, more generally, of a web site) is the number of times users viewed the page while browsing search results. ImpressionRank captures the visibility of pages and sites in search engines and is thus an important measure, which is of interest to web site owners, competitors, market analysts, and end users.

All previous approaches to estimating the ImpressionRank of a page rely on privileged access to private data sources, like the search engine's query log. In this paper we present the first *external* algorithm for estimating the ImpressionRank of a web page. This algorithm relies on access to three public data sources: the search engine, the query suggestion service of the search engine, and the web. In addition, the algorithm is *local* and uses modest resources. It can therefore be used by almost any party to estimate the ImpressionRank of any page on any search engine.

En route to estimating the ImpressionRank of a page, our algorithm solves a novel variant of the keyword extraction problem: it finds the most popular search keywords that drive impressions of a page.

Empirical analysis of the algorithm on the Google and Yahoo! search engines indicates that it is accurate and provides interesting insights about sites and search queries.

Categories and Subject Descriptors: H.3.3: Information Search and Retrieval.

General Terms: Measurement, Algorithms.

Keywords: search engines, estimation, ImpressionRank, popular keyword extraction, suggestions, auto-completions, data mining.

1. INTRODUCTION

Background. In recent years search engines have become an indispensable tool for information discovery. Billions of searches are performed by hundreds of millions of users around the world [2]. Since search engines drive a significant fraction of the traffic to web sites (see, e.g., [15]), these sites have become dependent on their visibility in search engines. For commercially oriented sites, this visibility directly affects the number of potential clients and revenues.

The visibility of a web site in search engines is therefore an important metric that web site owners, competitors, market researchers, and even users are interested in. Some search engines (like Google Trends for WebSites[1]) and online market research firms (like Alexa[2], Nielsen[3], and comScore[4]) provide traffic estimates for popular web sites. These rely on private data sources, like search query logs, web server logs, and network router logs.

In this paper we study *external* methods for estimating the visibility of a web page or a web site in a search engine. These methods rely only on public data sources to produce their estimates. Such methods are appealing for a number of reasons. First, they can be used by anyone, not just by search engines or other major companies. Second, they can be applied to any web page or web site, not just the ones for which search engines have decided to disclose visibility information (namely, the popular sites). Third, they can be applied to almost all search engines, even ones that currently do not disclose visibility information. Finally, they can be used to compare visibility of sites and pages in different search engines.

Problem statement. Visibility of a page or site in a search engine is formalized through the notion of *ImpressionRank*. We say that a page/site x has an *impression* on a query q in a search engine, if the user who sent q to the search engine viewed x as one of the results. The ImpressionRank of x is the total number of impressions x has on queries in the search engine within a certain time frame.

Due to the power law distribution of query frequencies [11, 16, 1], most of the impressions of pages and sites come from a small number of queries. For example, in our study we found that 73% of the impressions of the site `www.cnn.com` in Google come from the following 3 queries: "cnn", "election results", and "news". Hence, estimating the ImpressionRank of a page/site x can be reduced to finding the number of impressions x has on these top queries. In the *popular keyword extraction* problem we are given x and would like to find the k queries (a.k.a. *keywords*) on which x has the most impressions. This problem can be viewed as a variant of the classical keyword extraction problem, which has been studied in Information Retrieval for many years.

Our contributions. Our main contribution is the first external algorithm for popular keyword extraction. By the

[*]Supported by the Eshkol Fellowship of the Israeli Ministry of Science.

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

[1]`http://trends.google.com/websites`.
[2]`http://www.alexa.com`.
[3]`http://www.nielsen-online.com`.
[4]`http://www.comscore.com`.

aforementioned reduction, this algorithm can be used also to estimate ImpressionRank. The algorithm relies on three public data sources: (a) the search engine; (b) the query suggestion service of the search engine; (c) the web. Thus, the algorithm can work with any search engine that provides query suggestions (all major search engines do).

Our algorithm is *local*. That is, in order to extract the popular keywords from a page or site x, the algorithm probes its data sources only for information that is related to x. The algorithm does not need to perform a global computation on the entire search index or query log. Consequently, the algorithm requires modest resources. It sends a relatively small number of requests to the search engine (hundreds) and to the query suggestion service (tens of thousands) and fetches a relatively small number pages from the web (hundreds). The algorithm can run on a single PC using a standard broadband Internet connection and takes up to a few hours[5] to complete its task.

Empirical study we performed with the Google and Yahoo! search engines suggests that the algorithm is accurate: given a page, it finds on average 93% of its top keywords in Google and 84% of its top keywords in Yahoo!.

We used our algorithm to measure the visibility of popular and less popular sites as well as to find which keywords are the most popular for these sites.

Motivation. We believe that external algorithms for popular keyword extraction and for ImpressionRank estimation could be used as primitives in a variety of applications. We mention below some possible directions.

Popularity rating of pages and sites. Measuring usage metrics for web sites, and in particular the amount of traffic they get, is a well-established business, involving firms like Nielsen, comScore, and Alexa. Web site popularity ratings are important for marketing and PR as well as for determining advertising rates. The techniques used by the above market research firms require access to private information, like router logs and web server logs.

Assuming that search engine visibility is correlated with the amount of traffic web sites receive, external algorithms for ImpressionRank estimation provide a low-cost alternative for measuring the popularity of sites. In addition, analysis of the most popular queries on which a site has impressions can be used to derive demographic and geographical profiles of the users to whom the site is visible.

Site analytics. In order for web site owners to be able to enhance the amount of traffic they get from search engines, they need tools to measure and analyze their visibility in search engines. There are several products like Google Analytics[6] or OneStat[7] that help web masters analyze the traffic to their site based on web server logs or on scripts embedded in their pages. These products can show information about the search engine queries that resulted in actual *clicks* that led users to the site. However, they do not provide information about *impressions* of the site that have not resulted in clicks. Our algorithms can help site owners compare impressions and clicks and consequently derive *clickthrough rates* for their site on different queries and in different search engines. From these they can learn about the strengths and weaknesses of their site.

[5]Mainly due to rate limiting requests to the search engine.

[6]`http://www.google.com/analytics`.

[7]`http://www.onestat.com`.

Market research. While all the existing site analytics products rely on information that is available only to site owners, our algorithms can be used by anybody. Hence, they can be used to *compare* the visibility of different sites on different queries in different search engines. This could be useful for site owners to understand their competitive advantages and disadvantages as well as for market analysts who wish to study the whole market.

Search engine evaluation. Search engines try to filter out negative content, such as spam, hate sites, porn, and virus infected pages, from their search results. By estimating the ImpressionRank of a sample of such negative sites on different search engines, one can evaluate the effectiveness of the mechanisms these search engines apply for filtering negative content.

Our methodology. Search engines can compute the popular keywords and the ImpressionRanks for all pages and sites in their index quite easily using the MapReduce framework [3]. Without direct access to the search engine's query log or search index, this task becomes very challenging. First, it is not clear how, given a page, we can find the queries on which the search engine returns the page. Moreover, even if we had these queries, it seems impossible to find how many impressions of the page these queries have generated without accessing the query log. Finally, search engines pose strict limits on the rate of requests they are willing to accept from a single user/IP address.

Our algorithm overcomes these challenges as follows. In order to find queries on which the search engine returns the page, the algorithm resorts to standard IR techniques, like TF-IDF scoring and term proximity analysis, to extract candidate keywords from the page's content, from anchor text, and from similar pages. The algorithm determines which of these candidates generate impressions for the page by checking whether the search engine returns the page on them. In order to calculate the amount of impressions generated, the algorithm estimates the frequency of each candidate in the search engine's query log. To this end, the algorithm resorts to a technique developed in our previous work [1] for estimating query frequencies using the search engine's query suggestion service.

Implementing the above work plan naively is not feasible, because the algorithm may need to sift through numerous possible candidate queries. Evaluating all of them would have required sending a prohibitive number of requests to the search engine and to its suggestion service. To cope with the large amount of candidates, the algorithm applies a best-first search methodology [10] to identify the most promising candidates efficiently. A crucial ingredient of the algorithm is a low-cost procedure for estimating the frequencies of many candidates in bulk using only a small number of requests to the suggestion service.

For lack of space, some details are omitted from the main body of the paper. The interested reader will be able to find most of them in the full draft of the paper.[8]

2. SUGGESTION SERVICES

In this section we overview query suggestion services and how they can be used to estimate query frequencies.

Suggestion services. Query suggestion services such as

[8]Available at `http://www.ee.technion.ac.il/people/zivby`.

Google Suggest, Yahoo! Search Assist, Windows Live Toolbar, Ask Suggest, and Answers.com[9] are increasingly popular tools that assist users in choosing search queries. While the user is typing her query, the suggestion service offers the user auto-completions of the string she has already entered. These auto-completions help users save typing time and also guide them to better query formulations. Most suggestion services leverage "the wisdom of crowds" to generate the auto-completions (or, *suggestions*). Given a string α entered by the user, the suggestion service extracts from its query log the most popular queries of whom α is a prefix and returns them to the user, ordered by popularity.

Frequencies vs. volumes. The *frequency* of a query q, denoted $\text{freq}(q)$, is the number of instances q has in the query log. The *volume* of a string α, denoted $\text{vol}(\alpha)$, is the number of *distinct* queries in the log of whom α is a prefix.

The *shortest exposing prefix* of a query q is the shortest prefix α of q for which the suggestion service returns q as one of the suggestions for α. In a previous work [1], we studied the correlation between the frequency of queries and the volumes of their shortest exposing prefixes. We found that both measures have power law distributions, but with slightly different exponents. Moreover, we discovered that the two measures are "order-correlated". That is, if we order queries once by their frequencies and once by the volume of their shortest exposing prefixes, then the two orders have a high Kendall tau correlation.

As estimating string volumes is more feasible than estimating query frequencies directly, we rely on the above correlation in this paper too. We simply measure the popularity of queries by the volumes of their shortest exposing prefixes.

String volume estimation. In our previous work [1], we showed two techniques for estimating string volumes using only external access to the suggestion service. The first technique is an importance sampling procedure that produces relatively accurate volume estimates, but requires up to thousands of requests to the suggestion service to produce each estimate. The second technique is a heuristic that uses only a few dozens of requests to generate a rough volume approximation.

REMARK. Some suggestion services work differently from what we described above. They may generate suggestions from document titles or contents and not from query logs, they may rank suggestions not by their popularity, or they may offer suggestions that are not strict completions of the string entered by the user (e.g., spelling corrections).

Our algorithm's frequency estimates rely on the ranking provided by the suggestion service. If this ranking is not by query frequency but by some other measure (e.g., popularity in some document corpus), then our algorithm will reflect the same notion of popularity as well. As for suggestions that are not strict string completions, our algorithm simply ignores them and uses only the completion-type suggestions.

3. PROBLEM STATEMENT

Keywords and query logs. Throughout, we fix some search engine in which we want to measure visibility. A *keyword* is a sequence of one or more terms. A *query q* is an event in which a user probes the search engine for a certain keyword w in hope to get matching relevant documents. We say in this case that q is a *query instance* of w.

Fix a "query log", consisting of queries sent to the search engine in a certain time frame, in a certain language, and/or in a certain geographical region. For each keyword w, let $\text{freq}(w)$ be the frequency of w in the log, i.e., the number of query instances of w in the log.

Impressions. We say that a document x has an impression on a query q, if the user who sent q to the search engine viewed x as one of the results. This means: (a) that the search engine returned x as one of the results of q; and (b) that the user actually looked at the result corresponding to x (but not necessarily clicked it).

Previous studies (e.g., Joachims et al. [8]) studied the correlation between impressions/clicks and the position of the document in the result set. In this paper, for simplicity, we assume a naive impression model: every document in the results of a query has an impression. This assumption is not fundamental to our study. One can modify our algorithm by plugging in a more realistic impression model.

For a document x and a keyword w, let $\text{impressions}(x, w)$ be the *impression contribution* of w to x, that is the number of impressions x has on query instances of w in the query log. A simplifying assumption we make in this paper is that the search engine's index stays static throughout the generation of the query log and the time we perform our measurements. This assumption implies a simple characterization of $\text{impressions}(x, w)$:

$$\text{impressions}(x, w) = \text{incidence}(x, w) \cdot \text{freq}(w),$$

where $\text{incidence}(x, w)$ is a Boolean predicate specifying whether the search engine returns x as one of the results of w at measurement time. The assumption obviously does not hold in reality, because search engines continuously update their indices. Hence, our measurements can be viewed only as a static approximation of real impressions on search engines. See more discussion about this in the conclusions section.

We model impressions as a bipartite graph G, which we call the *impression graph*. One side of this graph consists of all the documents indexed by the search engine and the other side consists of all the keywords that have query instances in the log. A document x is connected to a keyword w by an edge if and only if $\text{impressions}(x, w) > 0$.

Popular keyword extraction. In the classical keyword extraction problem, one is given a document x and would like to find keywords that best "summarize" x. We leverage the indexing mechanisms of search engines, and rephrase this problem as follows: given a document x, find all the keywords w on which a search engine returns x. Under our definition, keyword extraction is search engine-dependent, which we view as a positive side-effect, as it allows comparing the different indexing approaches taken by search engines.

In the classical keyword extraction problem, the quality of the matching between a keyword and a document depends only on intrinsic properties of the keyword and the document. In this paper, we look also at the popularity of keywords among users, in order to determine their quality. Hence, our focus is on the following *popular keyword extraction problem*:

[9] `www.google.com`, `search.yahoo.com`, `toolbar.live.com`, `www.ask.com`, `www.answers.com`.

The Popular Keyword Extraction Problem
Input: A document x and an integer $k \geq 1$.
Output: The k keywords on which x had the most impressions in the query log.

Using the impression graph modeling, what we would like to find is the k neighbors of x of highest frequency in the log.

An optional requirement from algorithms that extract popular keywords is that they extract a *diverse* list of keywords. In particular, we would prefer not to get many keywords all of which are simple variations of the same keyword (e.g., "britney spears", "britney spears songs", "britney spears lyrics", etc.), even if, strictly speaking, these are the most popular keywords incident to the document x. We use a simple notion of diversity in this work: the algorithm should return a prefix-free list of keywords; that is, no keyword can be a prefix of any other keyword.

We focus on *external* algorithms for popular keyword extraction. That is, the algorithms are allowed to use only public data sources, and cannot rely on privileged access to internal search engine data, like its query logs, its index, its document cache, etc. The algorithms can send requests to the search engine and to its suggestion service and they can fetch pages from the web on their own.

The main cost measures for external algorithms are the number of search requests and the number of suggestion requests they send per input instance. Secondary cost metrics are the number of pages the algorithms fetch, their run-time, and their local storage requirements.

ImpressionRank. Impressions induce an order on documents, based on the number of impressions they had in the query log. *ImpressionRank* (or, *irank*, for short) captures this:

$$\text{irank}(x) = \sum_{w} \text{impressions}(x, w) = \sum_{w \in N(x)} \text{freq}(w),$$

where $N(x)$ denotes the keywords incident to x in the impression graph. This definition can be trivially generalized to measure the number of impressions of a set of pages (e.g., a web site): $\text{irank}(X) = \sum_{x \in X} \text{irank}(x)$.

Keyword frequencies in query logs are known to follow a power law distribution [11, 16, 1]. Due to the scale-free nature of power laws, it is reasonable to assume that also the frequencies of keywords generating impressions for a specific document follow a power law (the exponent may differ, though, from document to document). We use this observation to reduce the problem of estimating ImpressionRank to the popular keyword extraction problem.

If the frequencies of keywords incident to a document x perfectly follow a power law, we can use the frequencies of the top k keywords to infer the exponent of the power law through linear regression at the log-log plot. The exponent and the frequency of the most popular keyword can then be used to estimate the total frequency of the distribution's "tail" (the keywords beyond the top k). Summing up the total frequency of the tail with the frequencies of the top k keywords gives us an estimate of the ImpressionRank.

4. POPULAR KEYWORD EXTRACTION

In this section we describe our external algorithm for popular keyword extraction. The algorithm works with any search engine that has a query suggestion service in which ranking of suggestions is based on frequency in a query log. The algorithm uses the suggestion service to infer keyword frequencies, and thus the algorithm computes impressions relative to the query log that underlies the suggestion service.

Overview. Recall that our goal is to find the k most frequent keywords among the neighbors of a given target document x. However, we face two challenges: first, we do not know a priori who are the neighbors of x; second, even if we knew these neighbors, we do not know their frequencies.

Our algorithm generates a comprehensive list of *candidate keywords*, which is likely to cover most, if not all, the document's neighbors. The candidate keywords are combinations of terms from the document's text as well as from related text sources, like anchor text and similar documents. Ideally, the next steps of the algorithm would have been to find which of these candidate keywords are actual neighbors of the target document and then estimate their frequencies using the suggestion-based volume estimation algorithm (see Section 2). The algorithm could have then output the k most frequent neighbors. This naive approach, however, is not feasible, as the number of candidate keywords is typically very large: the document and the related text sources may consist of thousands of distinct terms and the number of their combinations is exponentially larger. Evaluating all these candidates requires sending numerous requests to the search engine and to its suggestion service, much beyond the limits posed by search engines.

Our algorithm therefore applies *best-first search* [10], in order to quickly track down the most promising candidates, evaluate them, and report the top keywords found. To this end, the algorithm starts with a seed set of candidates, consisting only of single terms, and assigns to each candidate a *score*. The score of a candidate is a low-cost predictor of its chances to be a prefix of a high frequency neighbor of the target document. At each iteration, the algorithm picks the candidate of highest score and determines: (a) whether the candidate itself is a high frequency neighbor of the target document; and (b) whether the candidate should be further expanded into additional candidates. To this end, the algorithm estimates the frequency of the candidate using the suggestion-based volume estimator. The algorithm stops whenever no candidates remain (this usually never happens) or when the search engine request budget has been exhausted.

Candidate keywords. To generate its candidate keywords, the algorithm collects a *seed text*. The seed text consists of the target document's text body, title, url, meta keywords, and meta description, of the anchor text of hyperlinks pointing to the target document, of keywords that have already been found to be incident to the target document, and of the contents of documents that the search engine returns alongside the target document on these keywords. Every time the algorithm finds a new keyword that is incident to the target document, it expands the seed text with the terms of this keyword as well as with the contents of the other documents that are incident to this keyword.

All the distinct terms that are found in the seed text comprise the *term pool*. The candidate keywords are all the possible finite-length sequences of terms from the term pool. Note that the order of terms in a keyword matters: "brad pitt" and "pitt brad" are different keywords. Clearly, the number of candidate keywords is infinite. At any given point of its execution, however, the algorithm considers only a finite subset of the candidate keywords.

Candidate keywords can be thought of as nodes in a TRIE, whose alphabet is all the terms in the term pool. Thus, the descendants of a given keyword are all the possible extensions of the keyword by terms from the pool. We call this TRIE the *candidate tree*.

4.1 Main flow

The algorithm, whose main flow is given in Figure 1, performs best-first search on the candidate tree, in order to identify the neighbors of highest frequency.

```
ExtractPopularKeywords($x$,$k$)
 1: crawl the seed text
 2: add the terms in the seed text to the term pool
 3: score all the terms in the pool
 4: insert the terms into the candidate heap
 5: while candidate heap $\neq \emptyset$ and budget not reached do
 6:    $w$ := top candidate from the candidate heap
 7:    remove $w$ from the candidate heap
 8:    send $w$ to the suggestion service
 9:    $S_w := w \cup \{$top suggestions for $w\}$
10:    for all $u \in S_w$ do
11:       if $u$ is incident to $x$ then
12:          estimate freq($u$)
13:          add $u$ to the top keywords heap, if possible
14:          expand the seed text with $u$ and the documents
             that are incident to $u$
15:          add all the new terms to the term pool
16:          rescore all terms in the pool
17:          clear the candidate heap
18:          insert all terms into the candidate heap
19:       end if
20:    end for
21:    if should expand $w$ then
22:       score the children of $w$
23:       add the children of $w$ to the candidate heap
24:    end if
25: end while
26: return the keywords in the top keyword heap
```

Figure 1: The main flow of the algorithm for extracting the k most popular keywords from a document.

Before we start describing the algorithm, we note that an important ingredient of the algorithm is a *cache*. The algorithm caches all the requests it sends to the search engine and to its suggestion service, all the responses it receives from them, and all the web pages that it fetches. Therefore, whenever the algorithm recalculates quantities it has calculated before (e.g., candidate scores), the recalculation can be done very quickly as no requests or fetches are needed.

The algorithm maintains a *candidate heap*, which consists of the "frontier" of the search space. Candidates are inserted into the heap based on their *scores*, which predict the potential of these candidates to lead to good keywords. Scoring is described in Section 4.2.

In addition, the algorithm holds a *top keywords heap*. This is a bounded-size heap of size at most k that consists of the currently known highest frequency keywords that are incident to the target document. The keywords are organized in the heap based on their estimated frequencies. The top keywords heap and the candidate heap are mutually exclusive: a keyword can reside only in one of them at a time.

Initially, the candidate heap is initialized with all the terms from the term pool (lines 1–4). At each iteration, the algorithm picks the candidate whose score is highest, removes it from the candidate heap, and gets the top suggested keywords for this candidate from the suggestion service (lines 6–9).

If the candidate or any of its suggestions are incident to the target document, the algorithm estimates their frequencies using the expensive volume estimator (see Section 2) and tries to add them to the top keywords heap (lines 9–10). A keyword is added to the heap, if the heap contains less than k keywords or if the keyword's frequency is higher than that of the bottom keyword in the heap.

A newly found incident keyword also opens up an opportunity for finding more keywords that are incident to the target document. The algorithm therefore augments (line 14) the seed text with the terms of the keyword as well as with the contents of all the documents that are incident to the keyword. The new terms are added to the term pool and all the terms are rescored (lines 15–16). This step is necessary, because the additional text may change scores of existing terms significantly. The term rescoring also affects candidate scores, and thus rebuilding the candidate heap is necessary too (lines 17–18). This step essentially resets the search process. Note that due to the cache used by the algorithm, the work done earlier is not lost.

Next, the algorithm determines whether the candidate should be expanded or pruned. In the former case, the algorithm scores the candidate's children and adds them to the candidate heap (lines 22–23).

The algorithm stops when there are no more candidates in the candidate heap or when the search requests budget or the suggestion requests budget have been exhausted (see more below on how these budgets are determined).

Deciding whether to expand a candidate. The procedure that decides whether to expand a candidate w first checks whether w itself, or any of the top suggestions for w, is a neighbor of x. If it is and if we require diversity in the keywords found by the algorithm (see Section 3), there is no point in further expanding w because even if we find matching keywords among its descendants, they will be suppressed.

Next, the procedure tests whether any of the descendants of w has a chance to be a neighbor x. If none of them does, there is no point in further expanding w and it is pruned. See below how the incidence test is carried out.

The procedure then determines whether any of the descendants has positive frequency. If none of them does, again there is no point in expanding w. This test is easy to do: if the suggestion service returns no suggestions for w, we know none of its descendants has positive frequency.

At this point, the procedure tries to assess whether any descendant of w has a chance to make it to the top keywords heap. If the top keywords heap hasn't reached it size limit yet, then any positive frequency descendant of w has a chance to make it to the top k, and thus the procedure decides to expand w. If the heap has reached its size limit, the procedure compares the bottom keyword on the heap with the top descendant of w (which is the top suggestion returned for w by the suggestion service). If the estimated frequency of the top descendant is lower than the estimated frequency of the bottom keyword on the heap, we are guaranteed that none of the descendants of w can make it to the top k. It is therefore safe in this case to prune w. Otherwise, w is expanded.

Testing incidence. Testing whether a given candidate keyword w is incident to the target document x is very easy: we simply send w as a query to the search engine and check whether x is returned as one of the top N results, where N is a tunable parameter ($N = 10$ is a typical choice).

Testing whether no descendant of w is incident to x is a bit trickier, because it is infeasible to send all the descendants of w to the search engine. To this end, we resort to the "inurl:" search option, which enables a user to restrict her search to documents whose url starts with a given string. We send the query "inurl:url(x) w" to the search engine. If there are no results returned, we conclude that neither w nor any of its descendants are incident to x. If results are returned, we cannot conclude one or the other.

Cost analysis. A simple analysis shows that the algorithm requires at most $N + 2$ search requests per iteration, where N is the maximum number of suggestions returned by the suggestion service. The number of suggestion requests is few dozens if the expensive frequency estimation is not performed during the iteration, and may reach few thousands if it is performed. We found that on average, a few hundreds of suggestion requests are needed for each iteration.

Setting the requests budgets. Budgeting search requests is easy, because we know each iteration takes at most $N + 2$ search requests. The budget can then be set according to the number of candidates we want the algorithm to go over. We found that usually a few dozens of iterations are sufficient (in extreme cases, we needed a few hundreds).

Budgeting suggestion requests is more difficult. Suggestions are used by two processes: (1) during candidate processing; (2) during frequency estimation. The former consumes a bounded number of suggestion requests, while the latter may need a variable number of requests, depending on the popularity of the keyword whose frequency is being estimated. We thus set separate budgets for candidate processing and for frequency estimation. A budget of a few tens of thousands of requests for candidate processing and of a few hundreds of thousands of requests for frequency estimation is sufficient in most cases. Note that the suggestion requests are more lightweight than regular search requests and are thus subject to much higher rate limits.

4.2 Candidate scoring

The most critical part of the popular keyword extraction algorithm is the scoring of candidate keywords. Good scoring will enable the algorithm to zero in on the most promising keywords quickly. Weak scoring will cause the algorithm to get lost in the large search space.

While the quality of the scoring function is very important, its efficiency (i.e., how many requests to the search engine/suggestion service it requires) is critical too. Since we need to calculate scores for thousands of candidates the feasibility of the whole algorithm depends on the cost of the scoring function. The scoring procedure we present below is effective at finding good candidates early and at the same time uses manageable resources.

The scoring function needs to promote candidate keywords that are more likely to be both neighbors of the target document and of high frequency. Recall that the *impression contribution* of a keyword w to the document x is defined as $\text{impressions}(x, w) = \text{freq}(w) \cdot \text{incidence}(x, w)$. The scoring function we introduce tries to produce (unnormalized) estimates of impression contributions. An ordering of candidates by their scores then approximates their ordering by impression contributions.

The score of each candidate is the product of two scores: a *frequency score* and an *incidence score*. The former tries to estimate the term $\text{freq}(w)$ and the latter tries to estimate the term $\text{incidence}(x, w)$.

Frequency scores are computed using the suggestion service. To this end, we developed a new low-cost procedure for producing rough frequency estimates for many keywords at bulk.

The incidence score of each candidate is the product of a TF-score (*term frequency score*) with an IDF-score (*inverse-document frequency score*). Here, we build on the common IR wisdom that terms of high TF-IDF values are more "uniquely representative" of the document. Hence, the search engine is more likely to match the document to such terms when they (or combinations thereof) are being sent as queries.

To summarize the score of each candidate keyword w is defined as the product of three terms:

$$\text{score}(w) = \text{fscore}(w)^{\alpha} \cdot \text{tfscore}(w)^{\beta} \cdot \text{idfscore}(w)^{\gamma},$$

where $\alpha, \beta, \gamma \geq 0$ are weights that are used to properly balance among the three scores (we assume here $0^0 = 1$, so if some weight is 0, the corresponding score is ignored). In our experiments we empirically optimized the setting of these weights; see Section 5.

We describe below how frequency scores are calculated. Since the calculation of the TF-scores and IDF-scores is somewhat routine and due to lack of space we omit its description here. More details can be found in the full draft of the paper.

Overview of frequency scoring. The frequency scorer is given as input a keyword w and it outputs frequency estimates for all the children of w in the candidate tree. The need to score all children of a candidate comes up in two situations: (1) when the algorithm initializes the candidate heap it scores all the single terms, which can be viewed as the children of the root (= the empty string); (2) when the algorithm expands a keyword it scores all its children.

Note that the number of children of w is the same as the number of terms in the term pool. Hence, the scorer needs to produce frequency estimates for up to thousands of keywords. Running the expensive volume estimator (see Section 2) requires a few thousands of suggest requests per frequency estimation. Doing this for all the thousands of candidates is infeasible. The procedure we introduce below applies the cheap volume estimator to a small number of the candidates and derives frequency estimates for all the candidates. The produced frequency estimates are rough, but are sufficient to differentiate between high frequency candidates and low frequency candidates.

The keyword TRIE. Let S be the set of keywords for which the frequency scorer needs to compute frequency scores. S consists of all the children of w, and all of these keywords have the same prefix (= w). The frequency scorer starts by constructing a character-level *keyword TRIE* T for S.

Recall that in order to estimate the frequency of a keyword u, the frequency estimator needs to estimate the volume of the shortest exposing prefix of u. In other words, it needs to find the top-most ancestor of u in T for which u is one of the top suggestions.

Cheap volume estimations. As mentioned above, the frequency scorer uses the cheap volume estimator to produce volume estimates. This estimator has the useful property that given a node $r \in T$, it estimates as a byproduct not only the volume of r but also the volumes of all the children of r.

The frequency scorer is given a priori a budget M on the number of invocations of the cheap volume estimator. Thus, the frequency scorer can estimate the volumes of at most M nodes of T and their children. Using this information the scorer needs to somehow derive frequency estimates for all the keywords in S.

We now need to address two questions: (a) how to select the nodes for volume estimation? (b) how to derive frequency estimates from these volume estimates?

Selecting nodes for volume estimation. When the scorer selects nodes for volume estimation, it tries to maximize the number of keywords for which it will be able to later derive frequency estimates. If the scorer estimates the volume of a node $r \in T$, then it will be able to derive frequency estimates for all the keywords that are descendants of r. Hence, the guideline in choosing the nodes for volume estimation is to prefer ones that have many keyword descendants.

The scorer calculates for each node r in the TRIE the number of keyword descendants, $d(r)$, it has. The scorer employs a max-heap, to which TRIE nodes will be inserted based on their $d(\cdot)$ values. The heap is initialized with the root of the keyword TRIE (corresponding to the empty string). The scorer iteratively estimates volumes of nodes from the TRIE until the heap is empty or the budget for the number of volume estimations is exhausted.

The scorer picks the top node r from the heap. This is the node that has the most keyword descendants among all the nodes in the TRIE whose child volumes have not yet been estimated. The scorer estimates the volumes of r and of its children by invoking the cheap volume estimator.

The scorer next considers the produced volume estimates for the children of r. If the volume for a child r' is 0 or 1, it means that the suggestion service returns no suggestions for r' (except, maybe, for r' itself). In particular, this means that all the keyword descendants of r' in the keyword TRIE have 0 frequency in the query log. The scorer therefore prunes the TRIE at r'; The frequency scores of the keyword descendants of r' are set 0.

If the volume of r' is greater than 1, then it is still possible that one or more of its keyword descendants has positive frequency in the log and thus r' is added to the heap.

A useful property of the above selection strategy is that if a node u is selected for volume estimation, it is guaranteed that that also all the ancestors of u have been selected. This property will be used below in the frequency estimations.

Frequency estimation. Next, we address the question of how to derive frequency estimates from the volume estimates produced. Let u be some keyword in S. There are two possible cases to consider.

The easy case is that u has some ancestor r in T that satisfies the following: (a) the scorer estimated the volume of r; (b) u is one of the top suggestions for r. In this case, let r' be the top-most ancestor of u, for which u is one of the top suggestions. Note that $r' = r$ or r is an ancestor of r and thus the scorer has a volume estimate for r' too. The scorer now derives a frequency estimate for u from the volume estimate of r' and from the position of u in the suggestions for r'.

The harder case is that u has no known ancestor for which it appears as one of the top suggestions. Let r be the lowermost ancestor of u for which the scorer estimated volume (at least one such ancestor always exists, because the scorer estimates the volume of the root). Suppose that one orders all the queries in the log of whom r is a prefix by their frequencies. The top N of these are returned by the suggestion service as suggestions for r. Let us call the rest the *suggestion tail of* r. Since u does not show up at the top suggestions, either it does not show up in this list at all (i.e., it does not show up in the query log) or it belongs to the suggestion tail of r. Since the scorer cannot tell apart the two cases, it will always assume the latter.

The scorer now estimates the frequency of u as follows. First, the scorer estimates the total frequency mass of the suggestion tail of r (i.e., the sum of the frequencies of all the suggestions in the tail). This can be done using the assumed power law distribution of query frequencies and using the volume estimate for r. See more details in our previous paper [1]. Let us denote this quantity F. Since the scorer has no information about the position of u within this tail, it simply assumes that u's frequency is the average frequency in the tail, which is $\frac{F}{\mathrm{vol}(r)-N}$, where N is the number of top suggestions for r.

In some instances the above frequency estimation leads to anomalies. If the number of descendants of r in the keyword TRIE that do not appear as top suggestions for r is bigger than $\mathrm{vol}(r) - N$, the total frequency mass they will receive by the above estimate will be higher than F. In order to avoid such anomalies, the scorer calculates the number n of keyword descendants of r that are not top suggestions for r. It then assigns the frequency score of u as follows:

$$\mathrm{fscore}(u) = \min\{\frac{F}{\mathrm{vol}(r) - N}, \frac{F}{n}\}.$$

5. EXPERIMENTAL RESULTS

Quality metrics. We first describe the quality metrics we used for evaluating popular keyword extraction. Given a document x, let $N(x)$ be the set of all keywords that are incident to x. Given a popular keyword extraction algorithm A, let $A(x)$ be the set of keywords that the algorithm returns on x. Achieving a precision of 1 (i.e., $A(x) \subseteq N(x)$) is trivial. Hence, our main quality metric is *recall*. We define two variants of recall: (a) $\mathrm{recall}_U(A, x) = |A(x)|/|N(x)|$, which measures the success the algorithm in extracting keywords with no regard to their popularity; and (b) $\mathrm{recall}_F(A, x) = \sum_{w \in A(x)} \mathrm{freq}(w) / \sum_{w \in N(x)} \mathrm{freq}(w)$, which measures the success of the algorithm in extracting popular keywords.

The above two metrics measure the quality of the algorithm on a specific document x. We define two corresponding metrics that aggregate the quality of the algorithm on all documents. $\mathrm{recall}_U(A) = \mathbb{E}(\mathrm{recall}_U(A, X))$, where X is a random document chosen proportionally to its degree in the impression graph, i.e., $\Pr(X = x) \propto |N(x)|$. $\mathrm{recall}_F(A) = \mathbb{E}(\mathrm{recall}_F(A, X))$, where X is chosen proportionally to ImpressionRank, i.e., $\Pr(X = x) \propto \mathrm{irank}(x)$.

The choice of these specific document distributions maintains the principle that in the former metric all keywords are treated equally, while in the latter keywords are weighted by their frequencies.

We use the following procedure to estimate the recall metrics: (1) Sample a random keyword w from the query log either uniformly (for estimating recall_U) or proportionally to its frequency (for estimating recall_F), using the sampling algorithms from [1]. (2) Send w to the search engine. (3) Select a uniformly chosen document x from the results of w. (4) Compute $A(x)$. (5) Return 1 if w belongs to $A(x)$ and 0 otherwise. In the full draft of the paper we prove that this procedure indeed estimates $\text{recall}_U(A)$ and $\text{recall}_F(A)$.

Note that the above recall measures are quite pessimistic, as they measures recall relative to the entire list of keywords that are incident to the target document. One could define also more refined measures, like recall@k, which consider coverage of only the top keywords. These measures, however, are harder to compute. We therefore focus in this paper on the above recall measures and keep in mind that they are lower bounds on the realistic recall.

Recall analysis. In order to evaluate the effectiveness of our algorithm, we measured the above two recall measures when running the algorithm on Google and on Yahoo!. We report the results in Table 1 for different settings of the parameters α, β, and γ, which determine the relative weight of the frequency score, TF score, and IDF score in our algorithm. The first row in this table corresponds to the best configuration of the algorithm we found (we reached this configuration using a hill-climbing search on the space of possible values for α, β, γ with the average of the estimated recall_F values on Google and Yahoo! as the objective function). The results indicate that the algorithm achieves high coverage on both Google and Yahoo! (93% and 84%, respectively, for the recall_F measure). The algorithm is more successful at extracting popular keywords than unpopular keywords, as indicated by the difference between the measured recall_F and recall_U values.

The three other rows of the table demonstrate the significance of each of the three scores we use in the algorithm. To this end, we eliminated each of the scores, by setting their corresponding weights to 0, and measured the resulting recall values. The results demonstrate that the TF score is the most crucial for achieving high recall, especially on Google. Without this score, recall_F on Google drops down from 93% to only 2%. The two other scores have marginal effect on recall_F but have more significant effect on recall_U.

α, β, γ	recall_F, **Google**	recall_F, **Yahoo!**	recall_U, **Google**	recall_U, **Yahoo!**
$0.2, 1, 0.6$	0.93 ± 0.08	0.84 ± 0.14	0.62 ± 0.06	0.37 ± 0.05
$0, 1, 0.6$	0.91 ± 0.09	0.80 ± 0.16	0.52 ± 0.06	0.27 ± 0.04
$0.2, 0, 0.6$	0.02 ± 0.01	0.07 ± 0.08	0.24 ± 0.05	0.17 ± 0.04
$0.2, 1, 0$	0.92 ± 0.08	0.82 ± 0.14	0.50 ± 0.06	0.17 ± 0.04

Table 1: Estimated recall values (together with measured standard deviations) for the popular keyword extraction algorithm.

A more refined breakdown of the recall_F analysis is shown in Figure 2. Here we split the random keywords selected by the recall estimator into 5 equally sized buckets, based on their frequency. We then measured the fraction of keywords in each bucket that the algorithm managed to detect. The results indicate that our algorithm performs much better on more popular keywords. At the top bucket, for example, it achieved a recall of almost 100% on both Google and Yahoo!. The recall deteriorates as the frequency of keywords goes down. This is to be expected, since our algorithm indeed tries to find the most popular keywords. We note that in 26% of the cases the algorithm did not find some keyword on Google and in 60% of the cases it did not find some keyword on Yahoo!, the algorithm did find at least one other more popular keyword.

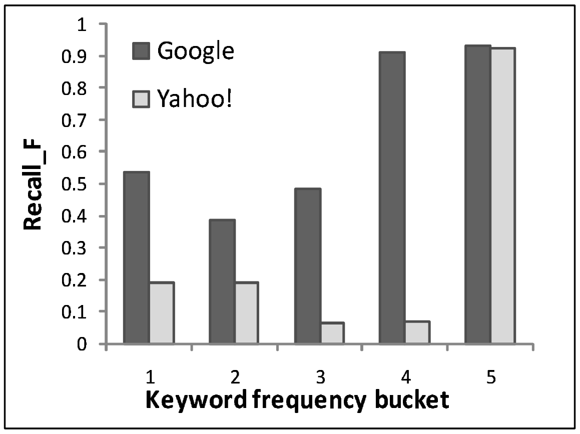

Figure 2: Recall as a function of keyword frequency.

The lower recall values measured for Yahoo! are due to the fact that the Yahoo! suggestion service seems to expose a larger set of low-frequency queries than Google. Hence, when choosing a random keyword proportionally to frequency, the chances to pick a low-frequency keyword in Yahoo! are higher than the chances to pick a low-frequency keyword in Google.

Cost analysis. To evaluate the efficiency of the algorithm, we measured how quickly it finds the keywords that it eventually outputs. We measure time in terms of the fraction of the request budget used. Progress is measured in terms of the fraction of the eventual keywords found, where keywords are weighted by their frequency.

The results, depicted in Figure 3, show that the algorithm finds most of the keywords very early on. On Google, the algorithm manages to find about 80% of the keywords after consuming as little as 20% of the budget. On Yahoo! the progress curve is a bit more moderate, again due to the abundance of low-frequency keywords in the log.

We also measured the effect of the frequency score, the TF score, and the IDF score on the costs of our algorithm. We found that when the frequency score was eliminated, the amortized number of search engine requests per keyword found grew by 64% on Google and 42% on Yahoo!. The amortized number of suggestion requests grew by 45% on Google, but did not change on Yahoo!. The TF score and the IDF score had more variable effects on the costs. See the complete results in the full draft of this paper.

ImpressionRank estimations. We used our popular keyword extraction algorithm to estimate the ImpressionRank of several popular sites on Google and Yahoo!. Figures 4 and 5 shows our ImpressionRank estimates, as of January-February 2009, for two sets of sites: news sites and travel

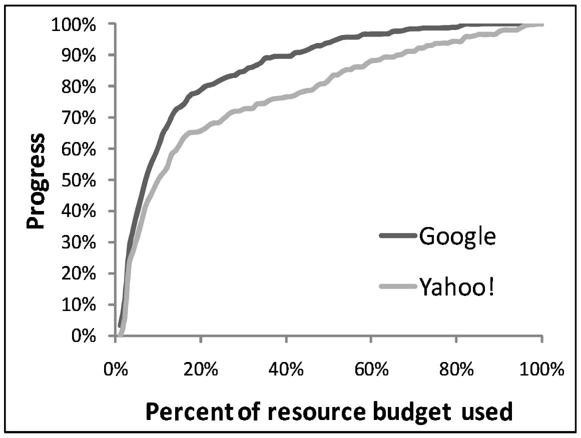

Figure 3: Keyword extraction progress.

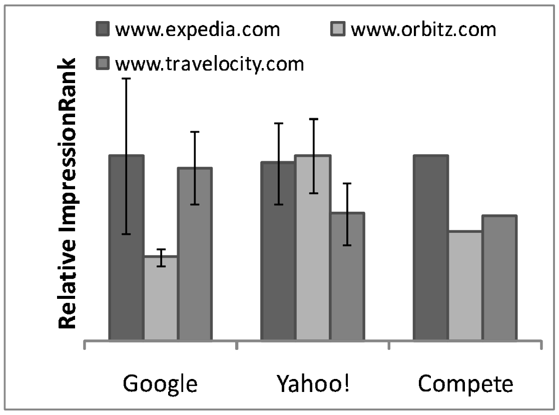

Figure 5: ImpressionRank estimates for travel sites.

sites. Note that the ImpressionRank estimates are relative within each search engine (the estimates for Google and Yahoo! are not comparable to each other). For comparison, we show traffic reports for the same sites as published by Compete[10], a firm that produces site traffic reports based on ISP, panel, and toolbar information. (We chose to show their reports, because they are the only traffic reporting firm we found that freely provides actual numbers, as opposed to unlabeled charts.)

Most of our estimates seem to be roughly consistent with the data provided by Compete, where the measurements on Google tend to demonstrate higher consistency. Note that even if our algorithm would have worked perfectly, we wouldn't have expected absolute consistency with the Compete data, because: (a) we measure impressions, not clicks, which correspond to actual visits to the site; (b) we measure potential traffic to the site coming from a search engine, while the Compete data reflects the overall traffic to the site, including traffic that does not originate from a search engine.

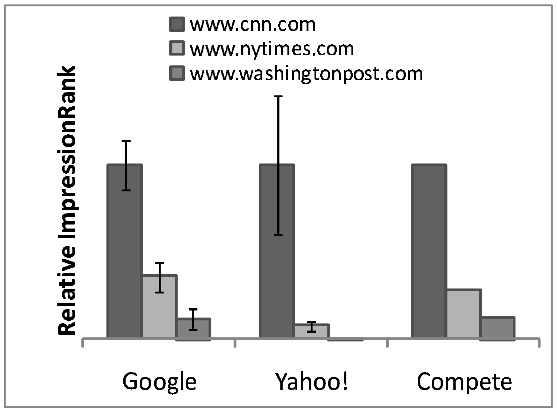

Figure 4: ImpressionRank estimates for news sites.

Popular keyword extraction. Table 2 shows the most popular keywords our algorithm found for several sites and pages of interest. Some of the interesting observations we found are the following. While hard-core news issues, like Barack Obama's election and the war in Iraq, drive many impressions to the news sites, these sites are also popular due to other things, like weather (cnn), crossword puzzles (New York Times), and comics (Washington Post). Travel sites earn many of their impressions from people looking for cheap flights, hotels, and car rentals. Expedia receives significant attention also from users looking for ski holidays, cruises, and vacation packages. Popular keywords for people's homepages reveal what they are best known for. For example, Kleinberg's book with Tardos and his papers about the small world phenomenon drive many impressions to his homepage. Bill Gates, on the other hand, is popular among searchers who wish to view pictures of his house.

Google	Yahoo!
www.cnn.com	
cnn, election results, news, obama, video, polls, health	weather, cnn, news of the world, obama, cnn news, presidential election
www.nytimes.com	
new york times, fashion, obama girl, crossword puzzles	new york times, ny times, crossword, the new york times, new times
www.washingtonpost.com	
obama tax plan, washington post, washington, post, the post	washington post, comics, newspaper, iraq news
en.wikipedia.org/wiki/PageRank	
page rank, ranking, google ranking, google page rank, pagerank	pagerank, google algorithm, google rank, page rank
www.expedia.com	
expedia, travel, travel agents, ski holidays, cruises, cruise deals	expedia, hotels, cheap hotels, travel, vacation packages
www.orbitz.com	
orbitz, car rental, cheap hotels, flights, airline tickets,	hotels, cheap tickets, cheap flights, orbitz, travel, flights
www.travelocity.com	
travelocity, travel, flights, flight tickets, travelocity flights,	hotels, cheap flights, travelocity, travel, flights, airline tickets
www.cs.cornell.edu/home/kleinber	
scientific american articles, kleinberg, kleinberg tardos, small world phenomenon	kleinberg, small world phenomenon, jon kleinberg, world phenomenon, robin ec08 parts
www.microsoft.com/BillGates	
bill gates, bill gates home, william gates, william gates iii bill gates speeches,	gates, william gates, william gates iii, pictures of bill gates house
infolab.stanford.edu/∼sergey	
sergey brin, favorite books, brin sergey, stanford sergey brin, data mining search engines	sergey brin, brin sergey, stanford home page, cs stanford, brin page, page brin

Table 2: Popular keywords for various sites and pages, ordered by frequency.

6. RELATED WORK

Keyword extraction. Algorithms for keyword extraction have been studied quite extensively [13, 4, 7, 6, 14, 9]. A commonly used framework for keyword extraction is as

[10]`http://siteanalytics.compete.com`

follows. A set of candidate keywords (e.g., phrases) are extracted from the target document's content. Each keyword is associated with a set of features (e.g., a TF-IDF score, the distance from the document's beginning, capitalization, etc). A classifier is then used to decide which candidates are returned as keywords and which are discarded. These works are not applicable to our problem, because when they select keywords they neither care whether a search engine returns the target document on these keywords nor they care about the popularity of these keywords.

Recent works [12, 5] on keyword suggestions for the purpose of keyword-based online advertising, perform keyword extraction similarly to previous methods, but use the popularity of keywords as an additional feature in their classifiers. However, these works rely on access to a private search query log in order to estimate keyword frequencies. Similarly, the Google AdWords Keyword Tool[11] is a keyword suggestion tool that presumably relies on access to Google's query log. Our algorithm is external and does not need access to private data sources.

All the above mentioned keyword extraction techniques perform an exhaustive search on all the candidate keywords they generate in order to find the best keywords. In our setting, however, the number of candidates is very large and thus brute-force search is not feasible. Our algorithm uses best-first search to zero in on the most promising candidates.

ImpressionRank estimation. In our previous work [1] we introduced the notion of ImpressionRank and presented an algorithm for sampling random pages from a search engine's index proportionally to their ImpressionRank. This sampling algorithm cannot be used to estimate the ImpressionRank of a given specific page. In [1] we also proposed a technique for estimating the popularity of a given keyword using the suggestion service. This technique is an essential component in the algorithm we develop in this paper.

Google Trends for WebSites enables comparison of the number of visitors to popular web sites over time and provides some additional information like related queries and geographic visitation patterns. The data for this service comes from internal Google data: search logs, Google Analytics data, etc. Similarly, comScore, Nielsen, Alexa, and other traffic reporting tools collect their information from private data sources.

7. CONCLUSIONS

In this work we introduced the popular keyword extraction and ImpressionRank estimation problems. We presented the first *external* and *local* algorithm for these problems.

Our algorithm assumed that the search results we get at the time we run the algorithm are the same as they were during the time frame the query log underlying the suggestion service has been collected. Obviously, this assumption does not hold in practice as search engine indices are constantly being updated. Our algorithm may produce unreliable results for pages corresponding to "dynamic" topics, where keyword popularity and/or search results change rapidly.

The reliability of our algorithm depends on the quality of the data set underlying the suggestion service. If the suggestion service filters many queries from its data set or if it does not refresh the data set frequently enough, our algorithm may produce biased/stale results. Another source of bias in our algorithm is the suggestion-based volume estimation of keywords, which as shown in our previous paper may have some bias.

Our algorithm produces only *relative* estimates of the ImpressionRank of documents. Producing absolute estimates would require knowledge of internal search engine data, like the size of the query log.

We note that while generally suggestion services seem to reflect query logs rather well, they deliberately filter out some negative keywords (e.g., porn or hate related). Our algorithms therefore have difficulty in estimation the ImpressionRank of documents whose popular keywords are negative.

Despite the shortcomings highlighted above, we believe our algorithms effectively use the limited public data available, and produce useful results.

[11] `https://adwords.google.com/select/KeywordToolExternal`.

8. REFERENCES

[1] Z. Bar-Yossef and M. Gurevich. Mining search engine query logs via suggestion sampling. In *34th VLDB*, 2008.

[2] comScore. 61 billion searches conducted worldwide in August. `www.comscore.com/press/release.asp?press=1802`, 2008.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[4] E. Frank, G. W. Paynter, I. H. Witten, C. Gutwin, and C. G. Nevill-Manning. Domain-specific keyphrase extraction. In *16th IJCAI*, pages 668–673, 1999.

[5] A. Fuxman, P. Tsaparas, K. Achan, and R. Agrawal. Using the wisdom of the crowds for keyword generation. In *17th WWW*, pages 61–70, 2008.

[6] J. Goodman and V. R. Carvalho. Implicit queries for email. In *CEAS*, July 2005.

[7] A. Hulth. Improved automatic keyword extraction given more linguistic knowledge. In *EMNLP*, pages 216–223, 2003.

[8] T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately interpreting clickthrough data as implicit feedback. In *28th SIGIR*, pages 154–161, 2005.

[9] D. Kelleher and S. Luz. Automatic hypertext keyphrase detection. In *22nd IJCAI*, 2005.

[10] S. J. Russell and P. Norvig. *Artificial Intelligence. A Modern Approach.* Prentice-Hall, 2nd edition, 2003.

[11] P. C. Saraiva, E. S. de Moura, R. C. Fonseca, W. M. Jr., B. A. Ribeiro-Neto, and N. Ziviani. Rank-preserving two-level caching for scalable search engines. In *24th SIGIR*, pages 51–58, 2001.

[12] W. tau Yih, J. Goodman, and V. R. Carvalho. Finding advertising keywords on web pages. In *15th WWW*, pages 213–222, 2006.

[13] P. D. Turney. Learning algorithms for keyphrase extraction. *Inf. Retr.*, 2(4):303–336, 2000.

[14] P. D. Turney. Coherent keyphrase extraction via web mining. In *20th IJCAI*, pages 434–439, 2003.

[15] M. B. Valentine. Google drives 70 percent of traffic to most web sites. `http://searchengineoptimism.com/Google_refers_70_percent.html`.

[16] Y. Xie and D. R. O'Hallaron. Locality in search engine queries and its implications for caching. In *21st INFOCOM*, 2002.