Efficiently Retrieving Relevant Pages for Fully-Qualified Entities

Jon Tedesco *
Department of Computer Science
University of Illinois at Urbana-Champaign

Nikita Spirin †
Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

Today's traditional web search engines allow users to submit keyword queries and retrieve relevant pages, yet we know that when users query these search engines, they seek real-world entities and find these entities by examining the relevant pages retrieved. As a result, recent web search research has focused heavily on searching for entities using traditional keyword queries. However, little work has been done on a complementary process: retrieving relevant pages given a real-world entity. Users often seek to find information about a specific entity online, an entity they often already know well. Users accomplish this task today by iteratively forming keyword queries and examining the retrieved results, but this process is tedious, error-prone, and difficult to automate. Through our research, we attempt to create a system that will improve this process, by investigating how we can accurately and efficiently search for relevant web pages using a fully-qualified entity. We implement a search engine that takes a fully-qualified entity as a query and provides more precise search results than traditional web search engines, by exploiting the structured information about this entity and combining results from multiple queries to an existing search engine.

CR Categories: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.3.5 [**Information Storage and Retrieval**]: Online Information Services

Keywords: entity search, web search, machine learning, ranking, query generation

1 Introduction

The advent of the World Wide Web has revolutionized the world, and is unprecedented not only in its size and impact but in its disorganization and lack of regulation. To navigate this vast disarray of knowledge, we use search engines to find pages relevant to specific keywords that relate to a topic. Although web search engines appear to be identical to traditional text search systems on the surface, web search engines face additional problems that arise from the structure and content of the web. The nature and magnitude of the internet necessitates that web search engines design intelligent crawlers to continuously retrieve pages to keep their collection up to date, and because few barriers exist to publish content on the web, search engines face the additional problem of determining the reputability of each page. Current search engines perform these tasks well, and allow users perform keyword queries to find documents containing related content. However, amid this disarray of information, there are inherently structured ideas and text that correspond to real-world entities. As the web, user expectations, and web search technology has evolved, web search has begun to focus how we can search for real-world entities rather than simply relevant pages.

For example, suppose a user is interested in finding more information about a particular brand and model of a camera. This user would form a query containing the camera's name or model number, and would submit it to a search engine to retrieve pages related to this camera. From these pages, the user would then find information about this particular camera. However, the user was not interested in retrieving web pages about this camera, but rather in finding out information about the camera itself. A new breed of search engines, known as entity search engines, addresses this problem by allowing users to search for real world objects on the web.

In the past, web pages were viewed simply as formatted text documents, albeit with additional considerations such as reputability, but overall highly unstructured. However, web pages that contain information about real-world objects are implicitly structured, and as entity search has emerged, research began to focus on the challenging problem of how we can extract this hidden structure from web pages, particularly in a domain-independent way. Perhaps since this new breed of search engines was a natural extension of text-based search engines, queries to these entity search engines have been performed through keyword queries. While this has naturally evolved from traditional web search, little work has been focused towards a complementary problem, namely how users can query for entities using prior information they may hold.

For instance, suppose Sally is a user is interested in finding information about a specific professor in computer science. In order to retrieve information about this professor using the traditional keyword-based web search engines, Sally must try to form

*e-mail: tedesco1@illinois.edu
†e-mail: spirin2@illinois.edu

relevant keyword queries about this entity. If she wants to find information about the professors areas of research, publications, or work experience, she must submit query after query to the search engine, gathering relevant results through this time-consuming and laborious process. She may find many results that are not relevant to the particular entity queried, and will iteratively read through relevant documents to find further information to expand her query about this entity.

However, this process can be time consuming, laborious, and error prone. To find all relevant documents for an entity, a user must iteratively query a search engine until he or she can find no additional relevant documents. With documents retrieved at each stage, the user must find the most important concepts, summarize these concepts into keywords, and use these keywords to find more detailed information about facets of an entity. The user could easily overlook important sections or keywords for relevant documents, lose patience and miss relevant pages, or make any number of errors that result in an incomplete set of relevant pages for a particular entity. Likewise, this process is painful for the user, and wasteful if we can accomplish this task automatically.

Using domain-specific information for the search process, we can ease these problems. Sally may have information about this professor that she is unable to easily express through a single keyword query. She may know that professors often work with a university or research lab, went to graduate school, participate in conferences, and submit publications. We can improve the process of entity search by using structured queries by allowing Sally to form a more accurate and intuitive description of these traits she already knows.

1.1 Idea

Since the advent of modern search engines, web search has surfaced as a prominent subset of information retrieval research. While research in this area was once geared primarily towards how we can retrieve relevant pages for a given keyword query, recent work in the field has begun to focus more on the problem of entity search than on the traditional web search problem.

1.1.1 Context

Traditional modern search engines, as we will refer to them, allow users to submit keyword queries and return pages that appear to be relevant based on keyword similarity alone. From these results, users then find the information they seek. Unlike a traditional text retrieval problem, web search has additional complications that necessitate a complicated scoring and retrieval system. Unlike documents in traditional text retrieval, web pages form a network of links that can be used to judge authoritativeness or authenticity of the page. Likewise, web pages contain extremely heterogeneous page styles, and all components of the page are not equally important.

In contrast, entity search engines, an emerging type of web search engines, allow users to submit keyword queries as with traditional modern search engines, but attempt to retrieve information about relevant real-world objects, rather than simply web pages that contain information about these objects. Often, users seek information about real-world objects, and entity search aims to allow users to retrieve this information directly, rather than indirectly through relevant pages, as is the case with traditional search engines.

Although much research has been focused on how to effectively solve the problem of entity search, little work has been focused on a complementary problem: allowing users to search for relevant pages using a real-world object. This type of problem, which we will refer to as *reverse entity search*, would take in a query representing a real-world object, and retrieve relevant web pages for this entity. The query would be some representation of the entity's structure, such as a database entry, which fully describes a particular instance of the entity.

Specifically, vertical search engines take a keyword query as input, and retrieve a list of instances of real-world objects, typically all the same type of object, that relate the input query. For example, if we enter the query *"The Matrix"* in a vertical search engine, we will receive results relating to the movies in the trilogy. Each result retrieved represents a real-world object, a movie in this case, and each real-world object result is a movie, but represents a different movie in the series. On the other hand, if we wanted to retrieve relevant pages for the movie *The Matrix*, a reverse entity search engine would take in a description of the movie, perhaps including the actors, director, studio, year, and exact title of the movie, and would retrieve web pages that relate to this specific entity. Where a typical vertical search engine retrieves a list of entities with the same schema, a reverse entity search would take in a specific entity.

1.1.2 Problem

We believe that this problem of reverse entity search is a reasonable and common use case for search engines today, yet little work has been done to optimize search engines for this task. Users searching for information about an entity often have prior knowledge about the entity for which they are searching and seek only to find out more information about this real-world object. This common problem relates closely to the problem of reverse entity search, where users query with full knowledge of an entity form a query using the entity's description to retrieve relevant pages from which they can find more information about this entity. Implementing an effective reverse entity search engine is a critical step towards addressing this user demand.

Unfortunately, current search technology does not offer a practical solution to users with this need. Although users can address this problem using a traditional web search engine, the process requires users to iteratively submit queries to the search engine, aggregating the results from individual queries, and ultimately determining the most relevant documents from the aggregate list of results. By its nature, this problem necessitates a high level of recall for any entity. This means that performing this task manually is tedious and time consuming, and by human nature, error-prone. Likewise, keyword query search engines lack the ability for users to properly describe an entity, and while more effective content query systems have been suggested in related work, these query systems are typically designed to allow users to express an entity schema, for example the domain of movie objects, not a particular entity, such as the movie itself.

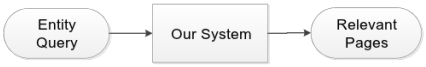
Our search system addresses these problems by allowing users to specify a description of a particular entity, not only an entity schema. Likewise, our system automates the manual process by generating keyword queries from the entity description, querying an existing traditional search engine, and aggregating the results from these queries to present a final list of relevant web pages for the queried entity. Finally, our system can effectively re-rank the aggregate list of results using the verbose information about the entity provided in the initial query to our system. This system allows users to accurately express the entity they seek, and as a result, allows us to provide more accurate and efficient search results in a practical way.

1.2 Formalization

Before we explain the details of our implementation, we first formally describe our problem and the architecture of our application.

1.2.1 Overview

Our reverse entity search system takes a fully-qualified entity as input, which should be represented in some domain-agnostic way, and outputs a list of relevant pages for the entity.



In this paper, we consider two entity types:

1. Computer science professors at the University of Illinois at Urbana-Champaign
2. Restaurants in Champaign

Each query is input as a JSON description, but we assume only that the input entity as an *entity id* that uniquely describes it in the context of entities from any domain. To ensure that our system is domain-independent, we perform our tests across both domains, using only the set of computer science professors to train our learning algorithm for the final stage, but testing each phase on entities from both domains. In the professor domain, we consider the *entity id* to be the name of the professor, and for the restaurants domain, we likewise consider it to be the restaurant name.

Using this entity description, we expect our system to ultimately output a re-ranked list of results relevant to the entity description. This list of results will be aggregated from successive queries to an existing search engine, and in this respect, our system functions similarly to a meta search engine. Since the input query is more verbose than a keyword-based search engine, and multiple queries are submitted to the existing search engine, our system can give more accurate results than a traditional search engine. Likewise, automating the manual process of finding relevant pages for an entity by implementing our system allows us to perform the original task more efficiently than before.

In our paper, we investigate the problem of automating this process, so that we can find relevant pages for an entity using arbitrary entity queries. We assume that we have fully-described entity and seek to retrieve relevant pages for this entity by

querying an existing search engine. Since we have more information about the entity and are able to submit multiple queries to retrieve results, we can combine results of multiple queries to an existing search engine and exploit the structured information about the entity to improve the quality of the search results.

The paper is organized as follows:

1. First, in Section 2 we summarize how similar work relates to and differs from our research
2. In Section 3, we give an overview of our strategy to solve the problem
3. In Section 4, we describe the methodology for our evaluation
4. We describe the details of each experiment we performed in Section 5
5. In Section 6, we display the results from these experiments
6. Finally, we then summarize our findings, and discuss their implications and future work that may stem from our findings in Sections 7 and 8

In each section, we delve deeper into our contributions, but we first begin by discussing the related and motivating problem we address.

2 Related Work

Although entity search is a relatively new and unexplored field, several researchers have already put forth significant effort to understand how we can effectively search for entities in the web.

2.1 Entity Extraction

As vertical web search first began to emerge, many researchers focused on the problem of how to extract structured information from pages on the web. Text content is often viewed as unstructured content, and is searched simply via word keyword scoring, yet text documents implicitly contain structured content in the form of information about real world entities. This structured content is often most effectively exploited when it is extracted into its relational form from text content. This problem is particularly applicable for web content since web pages often contain additional structured information such as tables, lists, or page layout that reflect the semantic structure of the underlying data.

Agichtein, Eskin, and Gravano studied how we can extract relations from unstructured text, eventually introducing a system called *Snowball* which was successfully able extract relations from collections of plain text documents in a domain independent way, without the need for the overhead of a large set of manually-provided example relations. While previous systems were also able to extract relations from unstructured text documents, traditional information extraction strategies typically tried to extract as much information as possible from a single document, and *Snowball* was one of the first systems able to effectively extract this information from a set of documents with only a small set of training examples [Agichtein et al. 2000].

To further improve the efficiency and effectiveness of entity information extraction, Agichtein and Gravano attempted to reduce the overhead for maintaining these mined object structures with additional documents by automatically generating queries to retrieve documents useful for generating entity structures. One of their key observations was that all documents are not equally valuable for learning an entity structure, and so we should not have to scan all documents to learn the structure of an entity. Their query-based algorithm helps address this problem by generating queries to select only the most useful documents for a particular entity. Their system learns from these documents, a small subset of the collection, to tremendously improve the efficiency of the extraction system [Agichtein and Gravano 2003].

Further research followed from Nie, Wu, Wen, and Ma to extend these findings to the domain of web search in particular, exploiting the inherent structure of HTML documents to help extract relational information from web pages. In their research, the authors claim that traditional entity extraction tools used for plain text documents are insufficient for the web domain, because they do not take advantage of the explicitly structured content represented through HTML formatting. Their paper asserts that we should distinguish between high-level structured information, such as real world entities, and web-level structure, such as portions of the page that address a particular object or idea, which they dub *object blocks*. Using this distinction, they introduce an approach called *Object-Level Information Extraction (OLIE)* based on the *Conditional Random Fields* algorithm to better address the problem of entity extraction. Their approach learns templates to extract objects for each web page independently, and maps these object blocks to real-world entities [Nie et al.].

Agichtein, Burges, and Brill likewise took advantage of HTML structure, particularly list and tables, and exploited redundancy in the web corpus to build an effective question answering system. Although their implementation does not exploit entity extraction techniques, the principles use can be easily extended to this domain [Agichtein 2007].

2.2 Entity Search

Following logically from this groundbreaking work in entity extraction, Cheng et al. introduced the concept of entity search, asserting that modern search engines that simply retrieve relevant pages are insufficient for today's search demands. The researchers claim that we often look for a particular entity, which we indirectly find through relevant pages. However, this entity should be provided directly to the user by the search engine directly. They quantify the entity search problem clearly, and demonstrate the feasibility of the problem by constructing a search engine for phone numbers and email addresses using 2TB of crawled data, proposing a scalable architecture for such systems [Cheng et al.]. In their further work, these same researchers introduce a framework for ranking entity search results in such a system, called *EntityRank*, quantifying several unique characteristics of the problem domain entity search. They assert that entity search should be:

1. Contextual
2. Holistic
3. Uncertain
4. Associative
5. Discriminative

Likewise, they define an effective query syntax to search for named entities, allowing users to specify both the target entity's schema and values [Cheng et al. 2007].

Further work on structured web queries appeared in a paper by Liu, Dong, and Halevy, discussing how we can run SQL queries on unstructured data such as web pages. Liu introduced an approach to construct keyword queries from well-structured queries and submit these keyword queries to existing web search engines to indirectly query unstructured web data using structured queries. Their work introduced a domain-independent strategy that exploited a query graph built from the initial structured query [Liu 2006].

Zhou et al. later introduced a framework for building efficient content query systems, called *Data-oriented Content Query System (DoCQS)*, introduces an efficient index structure and query processing algorithm for structured queries. The structured query language is intended to support any type of *content querying*, allowing *DoCQS* to to be used for web-based information extraction, typed-entity search, web-based question answering, and any similar tasks that may surface in the field in the near future [Mianwei Zhou].

In our research problem, we face these same inherent challenges with entity search as did these authors, but since we address a complementary problem, we approach these considerations from a different angle than in any of these papers. In our work, we are not interested in extracting entity information from web pages, so although the *entity extraction* work aforementioned helps setup the problem we address, our work does not directly build on this work. On the other hand, much work in *entity search*, particular in content querying, is an integral part of our work.

The query language that we use, essentially just a JSON description of a target entity, and could be easily generalized to the DoCQS system introduced for efficient content querying systems. However, it is important to note that our system expects a entity instance to be provided for a query, rather than simply an entity schema as some examples portray in earlier work presented with entity search.

2.3 Query Expansion with Entity Search

Our work likewise builds on earlier work in query expansion in the context of entity search.

In typical web search, we often face the challenge of inferring a user's intent to perform effective query expansion. Jiang et al. investigated this problem, claiming that systems often use only keyword queries, and overlook the hidden intent behind a user's query. They introduce an online search system called *AIDE*, which mines 14 millions MSN query log records to find related queries at run time for query expansion [Jiang et al.]. Likewise, Anagnostoploulos et al. take a unique approach to query recommendation by formulating the problem using a query-flow graph, where a sequence of user queries can be see as

traversing edges of this graph. Using the similarities represented through the graph, they provide shortcuts to shorten the user's process of query reformulation [Anagnostopoulos et al.].

While this work has undoubtedly been critical to our understanding of query intent analysis, in our research, we assume that the user's intent is to find relevant pages for a particular entity. Although we face the challenge of how to disambiguate the entity instance itself, there is little ambiguity as to the user's intent, and we are challenged instead in how we can effectively and programmatically generate keyword queries from the user's queries.

2.4 Learning Search Ranking

In our ranking phase, we attempt to effectively order search results based on a set of features for each document. This problem naturally lends itself to machine learning, and was in fact already applied to the problem of effectively ranking search results by Cohen, Schapire, and Singer.

Cohen et al. introduced the groundbreaking concept to apply machine learning to effectively order a list of items. They introduce the concept of a *preference function*, which determines a given element should be ranked above another, and use this function to outline a learning algorithm to effectively and efficiently order search results. The algorithm builds on the idea of ranking experts, and constructs a linear combination of these experts and compute the preference function iteratively to rank the results [Cohen et al. 1998].

Later, Joachims introduced an algorithm that takes a Support Vector Machine (SVM) approach to learn a ranking for a set of training examples in linear time [Joachims 2002] [Joachims 2006]. From his work, SVMRank was created. Our results likewise demonstrate an effective application of a learned ranking strategy, and could be easily extended to use this system.

3 Solution

3.1 System Architecture

Our system is intuitively separated into two phases, which we examine independently:

1. Retrieval Phase

   During this phase, we iteratively query an existing search engine to retrieve relevant pages for the given entity. We measure the effectiveness of this phase by both the total recall achieved by queries and the number of queries. To improve the performance of our system during this phase, we can investigate how we generate queries for the existing search engine from the full entity description. Specifically, we examine the effects of sources for keywords to use in the query expansion, query operators, and query ordering during this phase.

2. Ranking Phase

   During this phase, we rerank the retrieved results using a text indexing framework and the features for each page obtained during the retrieval phase. Our objective during this phase is to provide the most effectively ranked results as possible, and to obtain this, we experiment with each feature of the search results.

As aforementioned, our system takes a full entity description as input, and outputs a final re-ranked list of relevant pages. Thus, data flows through our application in several key stages, shown in Figure 1. We outline each stage below, and its role in our system. Each entry corresponds to a stage in Figure 1.

Entity Query	The fully-qualified entity query as described above
Retrieval Phase	Performs keyword query generation for the entity query
Keyword Queries	Created by the retrieval phase, iteratively sent to to the existing search engine
Search Engine	The existing search engine, Google in our experiments
Results	The results retrieved from the existing search engine, a ranking for each query sent
Ranking Phase	Aggregates the results retrieved from the existing search engine, producing a single ranked list of results
Results	The final ranking produced by our system

In Figure 1 we show two key points at which we can influence the final results:

1. Query Generation

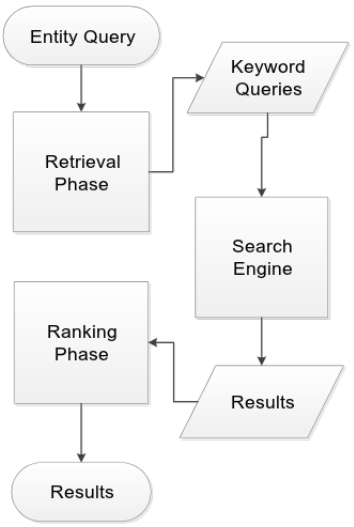
   Although we cannot control the results returned by the existing search engine, we choose how to generate keywords to effectively retrieve relevant pages for the entity given.

2. Document Scoring

   We have control over how we weight the features of relevant pages during the final re-ranking phase, and we use both manual experimentation and learning algorithms to discover the optimum weighting of result features.

We address the problem in two phases, which we will refer to as the *retrieval* and *ranking* phases, referring to the query expansion and re-ranking phases respectively.

Figure 1: *Application Architecture*



3.2 Retrieval Phase

The retrieval phase consists of a Python application that iteratively retrieves results for each entity by querying Google. To avoid changes in results over time and allow us to test our system effectively, we cached all results retrieved during this phase at the beginning of testing, and retrieved all rankings from the existing search engine during a short period of time, as quickly as the Google API allowed. This ensures that our results from Google remained consistent for all tests we performed during the course of our research. Although we do not currently apply machine learning to this phase, the learning process would take a test entity query as input, consist of four, intuitive learning steps, and output a learned query generation model. This process is shown in Figure 2.

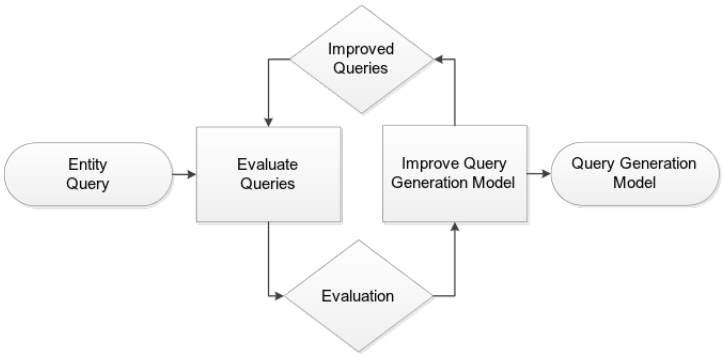
We segment the functionality of the implementation of this first phase into several components:

1. Query Builder

   This component is responsible for generating keyword queries for a given entity, and has no knowledge of the search engine interface, problem domain, or performance of any queries. Each query builder constructs queries by concatenating the entity identifier with a keyword from some keyword source, potentially using some query operator(s). For instance, this keyword source can be the entity schema, entity values, or entity keywords, which could result in queries such as *"John Smith" "institution"*, *"John Smith" "University of Illinois at Urbana-Champaign"*, or *"John Smith" "NSF career award"* for each keyword source respectively, given the professor schema and example entity information shown above.

2. Search Engine Façade

Figure 2: *Query Generation Learning Architecture*



This component is responsible for abstracting the particular search engine from the retrieved results, and allows us to submit a keyword query and retrieve the results from an existing search engine. Throughout this paper, we address the results retrieved from Google web search, with all personalized and location-based search features disabled.

3. Query Evaluator

   This component allows us to evaluate the effectiveness of our query according to the metrics of the retrieval phase. We discuss these metrics in depth below.

Our experimental results during the retrieval phase are evaluated using a manually created *golden standard*, a set of relevant URLs carefully constructed for each entity we test. We assume that this set of URLs is the best possible for this entity, and use it as manual relevance labels for our experiments.

3.3 Ranking Phase

Each experiment in the ranking phase uses a text index and search framework called *Whoosh* as its backbone. Whoosh provided accent handling and a porter stemmer for all tests with the ranking phase, and a built-in implementation of vector-space models such as BM25. Likewise, Whoosh provides a pluggable interface for scoring algorithms, allowing us to tweak the internal scoring algorithm used by Whoosh to perform searches.

To perform each query on our text index, we use Whoosh's extended query syntax to require the entity's id to appear in any retrieved documents, but allow only a subset of the keywords to appear. Each keyword is likewise surrounded in quotes, so that the exact keyword must be matched. We use Whoosh's plus-minus syntax to accomplish this, so that each query to the text index is formatted as:

```
+ENTITY_ID or KEYWORD_1 or ... or KEYWORD_N
```

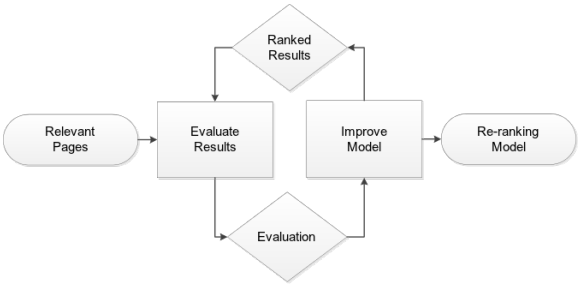
Given this query, Whoosh will apply stemming and accent handling, and return all results that contain the keyword `ENTITY_ID`, and any of the keywords `KEYWORD_1` through `KEYWORD_N`. Through our experiments, we see that this query system does correctly score documents, retrieving all documents containing exactly `ENTITY_ID`, and ranking scoring them based on the additional keywords `KEYWORD_1` through `KEYWORD_N`. We do not perform any experiments to determine the best way to query the Whoosh index, but use this query format consistently in each ranking experiment.

Just as with the retrieval phase, the ranking phase has an intuitive learning architecture that optimizes the quality of the output ranking, shown in Figure 3. We implement this learning system, and use it to learn an optimal weighting for features of our final results.

3.3.1 Non-relevant Pages

In practice, search engines typically contain a diverse collection of documents, varying tremendously in language, writing style, subject and length. This corpus provides many irrelevant documents for any given query. To obtain the same benefit from a set of non-relevant documents, we crawl 25,000 random web pages from *DMOZ.org*, a free directory of reputable pages on the web. Since our research is not concerned with determining the reputability of pages, we use this body of documents for smoothing to try to avoid any issues with spam or other useless documents, and insert each document into our text index and search framework as part of each ranking experiment [DMOZ 2011].

Figure 3: *Ranking Learning Architecture*



These help improve the quality of our text index retrieval results by providing a large corpus of non-relevant documents. Before using these documents, we perform experiments to compare the effectiveness of queries using different quantities of randomly selected documents from this corpus, and ultimately use the entire crawled DMOZ corpus in each ranking experiment we perform.

4 Evaluation Methodology

4.1 Evaluation

We examine the effectiveness of each phase individually, establishing a baseline for evaluation of each phase using a set of manually labeled relevant and non-relevant pages which we dub our *golden standard*.

4.1.1 Golden Standard

To judge the effectiveness of each phase, we establish a system of classifying relevant and non-relevant documents by manually labeling around forty relevant pages for each of the twenty entities from the two domains that we test.

4.1.1.1 Summary

We gather this set of relevant documents by using Google to manually gather relevant results for a given entity, following a simple algorithm, which we describe using Python below:

```
professorName = ...

foundNew = False
newKeywords = []

while foundNewRelevantDocuments:
  for keyword in newKeywords:

    # Query Gooogle with the professor's name and this keyword in quotes
        query(professorName + ' ' + keyword)

        # If we found new relevant results from this query
        foundNew = foundNew or foundNewRelevantResults()

        # Add keywords from these new results
        if foundNew:
        newKeywords += getNewKeywords()
```

We consider a page to be relevant if it specifically mentions the entity for which we are searching, and some information about it. In addition to explicit information about the page, this could include information from context. For instance, if we

see a professor's name mentioned on a conference proceedings page, we have learned that the professor participates in this conference, and would include this conference page in the set of relevant results for the professor.

We continue to query Google with these simple keyword queries, and collect the list of relevant URLs for an entity. On average, completing this process for a single entity took an hour and retrieved forty relevant, typically using at least queries to Google.

4.1.1.2 Validation

Manually extracting relevant results for each entity using this method was tedious, but yielded a highly relevant set of results for each entity. Since each result was inspected manually, we can be sure that each result is relevant to the entity, and although this process may miss some relevant results, we average our experimental results across all entities to minimize variance in these sets from human error. Additionally, since query expansion involved adding a single keyword at each iteration, and this keyword was manually selected from inspected relevant documents, the effectiveness of each query is difficult to algorithmically match, and sends an excellent upper bound for keyword selection performance.

The combination of the high quality of these result sets and number of entities that we use test our experiments ensures that we have an accurate and effective way to determine binary relevance, and in each of our experiments, we use these exact sets as relevance labels for each entity.

4.1.2 Retrieval Phase

During the retrieval phase, we use our *golden standard* to classify retrieved results as relevant or non-relevant, and measure the effectiveness of our query generation techniques using the recall and precision of the final set of results retrieved for each entity.

Specifically, for each entity, we collect the results retrieved from each query to Google, and measure the query generation performance using the recall of the aggregate set of retrieved results. We do not investigate the effectiveness of individual queries, since there may inconsistencies in particular query effectiveness, and so we only examine the results combined from all queries to Google. Likewise, since we will rank these results in the second phase, we are much more interested in the recall and than precision, and so consider the aggregate recall to be the most meaningful indicator of the effectiveness of each query generation technique.

To evaluate the efficiency of our query generation strategy, we likewise examine the total number of queries submitted to Google, and compare this to the number of queries in which the human achieves full recall.

4.1.3 Ranking Phase

In the ranking phase, we again use our *golden standard* to classify relevant and non-relevant documents, but apply more traditional retrieval metrics to gauge the relevance of our final list of results.

In terms of metrics, we calculate the precision and average precision at 1, 10, 20, and 50 results, the recall at 50 results, the r-precision, and precision at 100% recall. We use the traditional calculation of precision, average precision, and recall, but consider the relevant results retrieved, rather than the ideal set of relevant results, to calculate the r-precision and precision at maximum recall.

To establish a meaningful baseline against which to compare our ranking in terms of these metrics, we aggregate the results retrieved from Google for each entity, and return these directly. We aggregate results across all queries to Google for a particular entity by scoring each document based on its rank and frequency of appearance in Google's results. Specifically, for each result, we score each result's relevance using the following function:

$$score_{d_i} = \sum_{q_1}^{q_n} ln(numResults(q_i) - rank(d_i, q_i) + 1)$$

where:

d_i is a particular result retrieved from Google

q_i is a particular query to Google

$numResults(q_i)$ is the total number of results retrieved by query q_i

This scoring strategy yields impressive results alone, and provides a significant performance baseline against which we can compare our experimental results.

4.1.4 Domain Independence

Throughout our experiments, we refrain from making any assumptions about the specific domain of entities that we test. In both the retrieval and ranking phases, we assume that an entity has a string field which uniquely identifies it from any other entities. In the domains we study, this field is the professor's name, or name of a restaurant.

Similar characteristics are equally as likely to exist for other types of entities. We can view this notion of *entity id* as the unique identifier for a database entry, where the entry itself represents the entity instance, and the database table represents the entity schema or problem domain. In this way, we can view the *entity id* as a string representing the combination of properties that uniquely identify the entity

In both phases, we assume that entities are described using a simple key value scheme, by describing each entity as using a single JSON file. In applications similar to our system, we imagine that entities would be conveniently represented as a relational database, and our assumptions about an entity's description would allow us to use such a representation of the input entities with negligible changes to our system.

Likewise, to verify that our solution is indeed domain independent, we conduct our experiments across two types of entities: computer science professors and restaurants.

4.1.4.1 Professor Domain Below, we show the schema used for each professor used in our experiments. We assume that this schema is fixed for the entire problem domain.

```
{
    "name" :  ...  ,
    "institution" :  ...  ,
    "undergraduate education" :  ...  ,
    "graduate education" : [  ...  ],
    "research interests" : [  ...  ],
    "associations" : [  ...  ],
    "teaching" : [  ...  ],
    "publications" : [  ...  ],
    "keywords": [  ...  ]
}
```

Using this schema, each professor entity was expanded manually. Below, we show an example fully-qualified entity for a fictional data mining professor. Each entity contains values for each portion of its schema, and the *keywords* field provides the important keywords summarizing the entity, and is expected to be a part of the schema for each entity domain. This keywords field is expected to supplement the other fields, but not act as a replacement for the rest of the schema, and is only tested in combination with the other attributes of an entity.

```
{
    "name" : "John Smith" ,
    "institution" : "University of Illinois at Urbana-Champaign",
    "undergraduate education" : "Massachusetts Institute of Technology",
    "graduate education" : [
        "Stanford University"
    ],
    "research interests" : [
        "data mining",
        "natural language processing",
        "search engines"
    ],
    "associations" : [
        "WWW",
        "WSDM",
        "DAIS",
        "ACM"
    ],
    "teaching" : [
        "database systems",
        "data mining"
    ],
    "publications" : [
        "Something About Data Mining",
        "Something About Web Search"
    ],
    "keywords": [
        "University of Illinois at Urbana-Champaign",
        "Stanford University",
        "data mining",
        "databases",
        "information retrieval",
        "associate professor",
        "NSF career award"
    ]
}
```

4.1.4.2 Restaurant Domain

Likewise, we test our system using restaurants in the Champaign, where we consider a particular restaurant instance to be the branch of the restaurant in Champaign. Below, we show the schema for the restaurant domain:

```
{
    "name" : "...",
    "city" : "...",
    "cuisine" : ["..."],
    "state" : "...",
    "zipcode" : "...",
    "phone" : ["..."],
    "address" : ["..."],
    "attire" : "...",
    "delivery" : "...",
    "take-out" : "...",
    "dishes" : ["..."],
    "ingredients" : ["..."]
}
```

And below, we show this schema filled in with some minimal values:

```
{
    "name" : "Bombay Grill",
    "city" : "Champaign",
    "cuisine" : [
        "indian"
    ],
    "state" : "Illinois",
    "zipcode" : "61820",
    "phone" : [
        "(217) 398-8400"
    ],
    "address" : [
        "401 E. Green St. Champaign IL, 61820"
    ],
    "attire" : "casual",
    "delivery" : "yes",
    "take-out" : "yes",
    "dishes" : [
        "chicken curry",
        "chicken korma",
        "chicken vendaloo",
        "chicken saag",
        "lamb aloo gosht",
        "lamb saag",
        "lamg aloo ginger curry"
    ],
    "ingredients" : [
        "curry",
        "saag",
        "tandoori",
        "lamb",
        "shrimp",
        "pakoras",
        "naan",
        "tikka",
    ]
}
```

5 Experiments

To develop a good overall system to perform reverse entity search, we experiment with each phase independently, and combine the most effective strategies from each phase. In each experiment, we only consider web page results, ignoring binary files, and ultimately combine the first and second phases into an aggregate experiment augmenting the final re-ranking with machine learning.

5.1 Retrieval Phase

During the retrieval phase, we have control over the queries we submit to the existing search engine, both how we form them and the order in which we submit them. We try to optimize our query strategy in two dimensions:

1. Quality

   The quality of the results, which we measure using recall of the aggregate set of results of retrieved using the query strategy.

2. Efficiency

   The cost of retrieving the aggregate set of results, which we measure by the number of queries we submit to the existing search engine.

Each experiment begins with the fully described entity as an input and generates queries from this description. Each query is comprised of keywords or keyword phrases, consisting of the entity id and a keyword or keyword phrase selected from the entity description.

In each query, we include the entity id as the first search term, which would ideally be specified from the description itself. As with many existing search engines, Google acknowledges the order of keywords in the query, recognizing the first term as the most important of the query. This helps ensure that results are relevant specifically to this entity by telling Google that the entity id is the most important keyword of the query, and helps prevent content drift in our results.

5.1.1 Experimental Setup

Each retrieval setup is segmented into query generation and submission from Google. We experimentally tweak the query generation phase to construct and order queries to be sent to Google, and measure the both the aggregate set of results retrieved and number of queries submitted to Google.

5.1.2 Baseline

To establish a good baseline for quality and efficiency, we consider the list of queries submitted during the retrieval of the *Golden Standard*. The aggregate set of results retrieved establishes the upper bound for recall, at 1.0, and the lower bound for the number of queries, since each iteration of query expansion was performed with manually selected keywords from the relevant documents retrieved. Since humans can adapt to each entity domain equally well, inspect the entity description for semantic meaning, and adaptively select ideal keywords from relevant documents, the quality and efficiency of manually performing this process establishes an excellent realistic upper bound for the performance and efficiency of our system.

Likewise, we establish a naive baseline for the retrieval phase by examining the quality and efficiency of retrieving results from a single query using the entity id only. This baseline is very efficient, since it only requires one query, but does not achieve as high of recall as with other query strategies.

5.2 Keyword Source Experiments

First, we examine the source of keywords as a variable to tweak to improve the overall performance of the retrieval phase. We assume that the entire entity description is structured using keys and values, and investigate the keys, keywords describing the problem domain schema, and values, the keywords specific to this entity, separately. Likewise, we try using Yahoo's search API to select the most meaningful queries from the entity description.

Specifically, for each keyword source experiment, we examine the following sources of keywords:

- Entity Attribute Names (Keys)

  Keywords generated from the schema of the entity domain

- Entity Attribute Values

  Keywords generated from the description of the entity instance

- Both Entity Attribute Names and Values

  Keywords generated from both the entity schema and instance

- Yahoo Keywords

  Keywords selected by the Yahoo YQL search API. These keywords are a handful of keywords or keyword phrases selected from the entire entity description that Yahoo asserts are the most semantically meaningful keywords.

Each source of keywords is tested in combination with each operator and ordering query strategy, as described below.

5.2.1 Query Operator Experiments

Next, we use the keywords retrieved using each keyword strategy mentioned above in combination with query operators provided by Google. Specifically, we investigate the following query operator experiments. In each example, suppose k is a given keyword search term or phrase submitted to Google.

k — No operators

Simply use a space-delimited concatenation of the keywords selected from the previous stage

"k" — Quotes

Enforces that the entity id and expanded keyword are matched exactly for each query

$\sim$ k — Approximate

Include related words for each keyword to be counted as hits, for both the entity id and the expanded keyword and any substring of the query

$\sim$ "k" — Exact-Approximate

Include words or phrases related to the exact entity id or keyword phrase submitted

Unfortunately, we cannot know the specific implications of each of these query operators, but we do know their semantic meaning based on Google's description. Again, each of these operator schemes is combined with each keyword source and ordering mechanism

5.2.2 Query Ordering Experiments

Finally, we can take the results the queries generated by tweaking operator and keyword use, and reorder the queries to retrieve the most meaningful results first, allowing us to help minimize the number of queries sent to the existing search engine. Since we cannot reorder the queries based on the number of new results each will retrieve, we attempt estimate the meaningfulness of queries using the keyword significance of each query.

Specifically, we score each query by summing the values of the polysemy score assigned by WordNet to each keyword in the query. The score is provided by WordNet to reflect the uniqueness of the word in the corpus of the English language. We verify that each keyword is spelled correctly in the entity description, and consequently, that any word not recognized by WordNet is a proper noun that bears significant meaning to the entity. In our results, we empirically determined the value to assign in this case, based on the recall of the top ten, fifteen, and twenty queries returned by this scoring.

We calculate the score for each word for a query using:

$$score(k) = \begin{cases} 15 & \text{if } \textit{WordNet} \text{ does not recognize } k \\ wn(k) & \text{if } \textit{WordNet} \text{ recognizes } k \end{cases}$$

where

$wn(k)$ = the sum polysemy scores assigned by WordNet to every part of speech interpretation associated with k

Then, an aggregate score is assigned to each query by simply finding the sum of scores of each word in the query:

$$score_q = \sum_{k_1}^{k_n} wn(k_i)$$

We then sort the queries in descending aggregate score, so that the highest scoring and most important queries are submitted first.

5.3 Ranking Phase

We evaluate the performance the ranking phase, the final output of our search system, using a set of traditional search quality metrics. Each metric is computed using the *golden standard* set gathered for the entities.

To properly judge the effectiveness of each ranking scheme, we establish a baseline using the aforementioned scoring system from Google's results. This baseline is generated for every set or results we experimentally rank, and compared against our

experimental results.

5.3.1 Experimental Setup

To run each ranking experiment, the set of results created from the retrieval phase is fed into the ranking phase as input. Using these results, we first gather additional features for each result, and insert the completed results into an index created by Whoosh, a pure Python search and indexing framework.

For each document and experiment, we clean all text content before inserting it into the system, and removed stop words and removing any HTML content from the web page. Likewise, we use porter stemmer and accent handling, both provided by Whoosh, and a Whoosh multifield parser to support scoring multiple fields simultaneously.

Below, we show a code snippet outlining initialization of our index with this integrated features from Whoosh:

```
# Build the stemmer
analyzer = StemmingAnalyzer()

# Add accent handling
analyzer |= CharsetFilter(accent_map)

# Define the Whoosh index schema
indexSchema = Schema(

  # The URL
  url=ID(stored=True, unique=True),

  # The cleaned page content
  content=TEXT(analyzer=analyzer, stored=True),

          .
          .
          .

  # The PageRank of the page
  pageRank=NUMERIC(stored=True)
)
```

For each text-based feature of a document, we use the BM25 score computed by the framework, and for numerical scores such as PageRank and our baseline score, we augment the final numerical score provided by Whoosh. To support weighting for different document fields, we use Whoosh's integrated feature weighting system, called *term boosts* in the Whoosh package.

To illustrate our setup, we show a brief code snippet below, which defines the

```
# Open the index
index = open_dir(indexLocation)

# Create a searcher object for this index
searcher = index.searcher(weighting=BM25)

# Build the query from an entity description
query = buildQuery(entity, entityId)

# Search the index
results = searcher.search(query)
```

For each experiment, we use the same query scheme, which requires that the entity id appear in retrieved documents, and further scores each document using the additional keywords provided. Below, we show a code snippet depicting the formation of such a query from an entity description:

```
def buildQuery(entity, entityId)

  # Get all the keywords from the entity
  entityKeywords = selectKeywords(entity)

  # Build the query itself
  query = "+" + entityId + " "
  for keyword in :
    query += keyword + " "

  return query
```

For each document, we gather the following fields:

1. Content

   The cleaned text content of the web page

2. Title

   The title text of the page

3. Description

   The meta description text of the page

4. Keywords

   The meta keywords text of the page

5. Headers

   The text from HTML header tags of the page

6. Yahoo Keywords

   Relevant keywords for the page gathered from the Yahoo search API

7. Expanded Yahoo Keywords

   Yahoo keywords for the page, expanded to include synonyms for all words using WordNet

8. Baseline Score

   Calculated from the unranked results retrieved from Google.

9. PageRank

   Retrieved using the Google toolbar URL

Since we were not interested in investigating the specifics of BM25 scoring efficiency on the text fields of our results, we allow the BM25 scoring to be delegated to Whoosh. Below, we offer an in-depth description of the non-trivial fields that we added to each document, from the list above.

5.3.1.1 Yahoo Keywords

For each document, we retrieve the representative keywords from Yahoo's YQL search API, which typically gives us 100-200 keywords for the document. This forms a keyword summary for the document using the content text of the document itself. The text of this summary field is again scored with BM25 similarity for the query, which remains fixed through each ranking experiment.

To send the result content text to Yahoo, we split the content into 2000 character chunks, and concatenate the resulting keywords into a space-delimited keyword summary for the document. A simplified code snippet performing this keyword extraction for a result is shown below:

```
# Clean the content of stop words &
#  formatting tags
content = cleanContent(content)

# Group the content into chunks
#  to send to Yahoo API
contentChunks = breakUp(content)

# Get the keywords for each chunk
keywords = []
for contentChunk in contentChunks:

  # Get the keyword data from Yahoo
  newKeywords = getKeywords(contentChunk)
  keywords += newKeywords
```

5.3.1.2 Expanded Yahoo Keywords

For each document, we take the Yahoo keyword string and further expand the keywords by finding the synonyms for each keyword using WordNet. This introduces a larger, more inclusive keyword summary for each result. This improves the performance of our indexing framework, compensating for any shortcomings in Whoosh's finding related keywords for queries. We introduce this feature because the Whoosh scoring framework will not find related keywords, only stems of the same word.

Below, we show a simplified code snippet which outlines our strategy for expanding these keywords with WordNet:

```
# Collect the expanded list of keywords
expandedKeywords = []
for keyword in keywords:

  # The WordNet command, we get synonyms for
  #  noun, adjective, and verb senses
  command = "wn " + keyword + \
            " -synsa -synsv -synsn"
  output = execute(command)

  # Expand the list of keywords
  expandedKeywords += parse(output)
```

5.3.1.3 Baseline Score

We likewise include the baseline score calculated from the original Google ranking as feature, and parameterize between this score and the score assigned by Whoosh to the document. We can tweak the weighting of this feature by changing the coefficient to this baseline score in the final document scoring function.

5.3.1.4 PageRank

We include the PageRank of each document, which we use to scale the final score assigned to the document or to add a constant multiple to the score assigned by Whoosh. We tweak the weighting of each use independently in our learning experiments, but include scale by the score assigned by Whoosh to demonstrate the improvement in ranking quality that this feature alone can provide.

In our experiments, we show the impact of each feature individually on our final ranking. For each feature of a result, we compare the performance using this feature against the baseline of scoring the result only using the BM25 score for the result.

Likewise, we investigate the effect of smoothing on our ranking by introducing different amounts of non-relevant documents into the index and comparing the results of ranking using only the BM25 score of each result's content against the same scoring strategy without any smoothed documents.

5.3.1.5 Learning

Next, we tweak these features more precisely using a simple coordinate descent learning algorithm and RankSVM to obtain the optimum weighting for the features of each document, training the ranking using all training entities and comparing the effectiveness of the two strategies. We then evaluate the effectiveness of the resulting ranking for each individual training entity, and compare it to the ranking baseline for each training entity.

We represent each document as a feature vector consisting of BM25 scores of the content, title, header, meta keywords, meta description, Yahoo keywords, and expanded Yahoo keywords, as well as the PageRank and baseline score. For each test, the index is again smoothed using just over 20,000 non-relevant DMOZ documents. Each document is labeled either relevant or non-relevant, based on the *golden standard* set for the entity, which provides RankSVM with a binary preference function to use to effectively rank the results.

6 Results

6.1 Retrieval Phase

During the retrieval phase, we investigate how to effectively generate our queries. This comprises of selecting the optimal keywords, using query operators effectively, and ordering queries to minimize the number of queries necessary. To evaluate the results of each experiment, we compare the results of our experiments to two baselines:

1. EntityID (Naive)

   This baseline involves submitting a single query, the entity ID, to the existing search engine. Each query generation strategy cannot perform fewer queries than this baseline, but must outperform it in terms of recall.

2. Human

   This baseline establishes an upper bound for the performance of our query generation strategy. We cannot outperform this baseline in terms of performance, since the human queries are used for relevance labels, but we should be able to match or improve upon the number of queries required by the human, while maintaining good recall.

In the following sections, we discuss the results of each experiment performed according to these metrics, and ultimately compare an automated query generation strategy to each baseline.

6.1.1 Query Keywords

In our results for query generation keyword sources, we investigate the performance and efficiency through recall and precision, and number of queries, respectively.

In Figure 4, we can see a summary of both metrics for each source of keywords for query expansion that we investigated. Noteworthy is that naively using each keyword of the full entity description, with both attributes and values, retrieved the highest recall, at nearly 0.70. However, this required an average of 30 queries, significantly higher than the average of 22 required by using the attribute values only, which achieved comparable recall. Both of these naive strategies significantly outperformed the naive attribute names strategy, which required only 10 queries on average, but achieved a disappointing recall of 0.3.

Using relevant keywords generated by the Yahoo YQL API, submitted in no particular order and without using any operators, we were able to submit just over 10 queries, as with the attribute values queries, but achieve a higher recall of 0.5. We compare the performance of each scheme directly in Figure 5

This means that using Yahoo keywords selection, or a similar keyword selection technique, we can significantly reduce the number of queries required to programmatically achieve a high recall. We compare the efficiency of each scheme directly in Figure 6

We average the results for experiment across all twenty entities we tested, from both professor and restaurant entities. This ensures that these results are indeed domain independent, and generalize to arbitrary types of entities.

Figure 4: *Summary of Efficiency and Performance for Keyword Source*

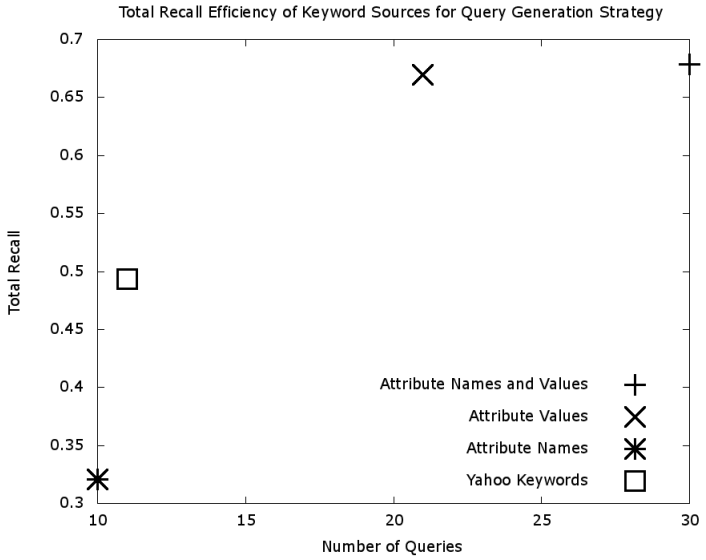


Figure 5: *Performance for Keyword Source*

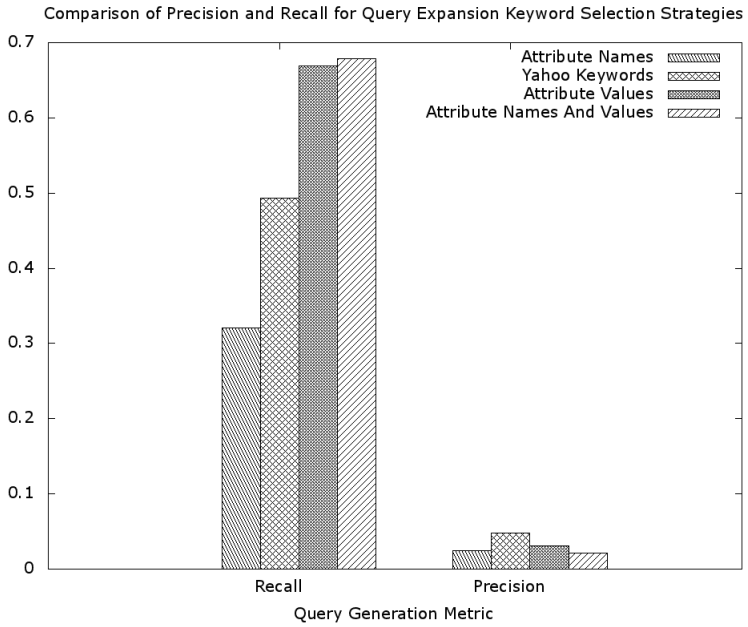
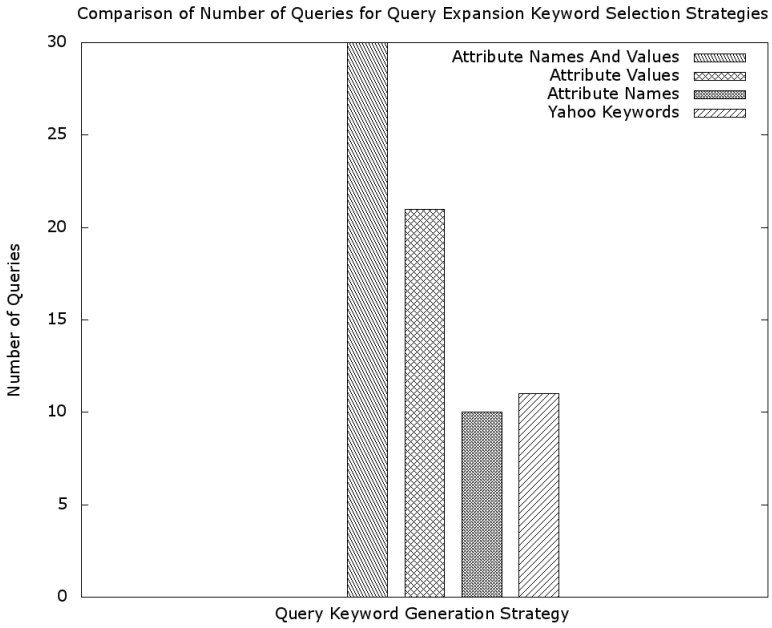


Figure 6: *Efficiency for Keyword Source*



6.1.2 Query Operators

Likewise, we can augment these selected keywords using the query operators supported by the search engine. While this is independent of the number of queries submitted to the search engine, this does affect the recall of any single query. In the Figure 7, we use both entity attribute name and value keywords in combination with the specified keywords to investigate the effect of each on the recall of a single query.

We likewise investigated the effectiveness of exclusive operators, such as $-$, which penalize or exclude results that include a specified search term, but decided to avoid it for experiments. This operator is frequently used on unrelated terms for ambiguous queries, but since we attempt to specify or queries programmatically, and there are typically highly related terms generated from the same entity, this operator would likely only introduce further complexity.

From Figure 7, we can see that using exact operator semantics reduced the overall recall on average, even in combination with the *related* operator. This makes sense intuitively, because enforcing an exact term match in the search engine typically reduces the recall while improving precision. On the other hand, we can see that using the *related* query operator did slightly improve over using no query operators.

Again, we average the results for experiment across all twenty entities we tested, ensuring that these results are also domain independent.

6.1.3 Query Execution Order

Finally, we use WordNet to score the keyword queries, determining the most meaningful queries and allowing us to reorder queries to submit the most meaningful queries first. In Figure 8, we show that reording these queries, we can retrieve a higher recall than a random ordering using only the top ten queries, and match the maximum recall achieved using random ordering with significantly fewer queries.

This tells us that using WordNet's keyword significance alone for query scoring does help find the most impactful queries. Intuitively this makes sense, and we have experimentally shown that it provides significant improvement.

6.2 Ranking Phase

In our ranking phase experiments, we first compare the effectiveness of each feature we add to the scored documents in Whoosh's framework. In each figure, we show the result of scoring documents using the BM25 scored content in along

Figure 7: *Recall for Query Operators*

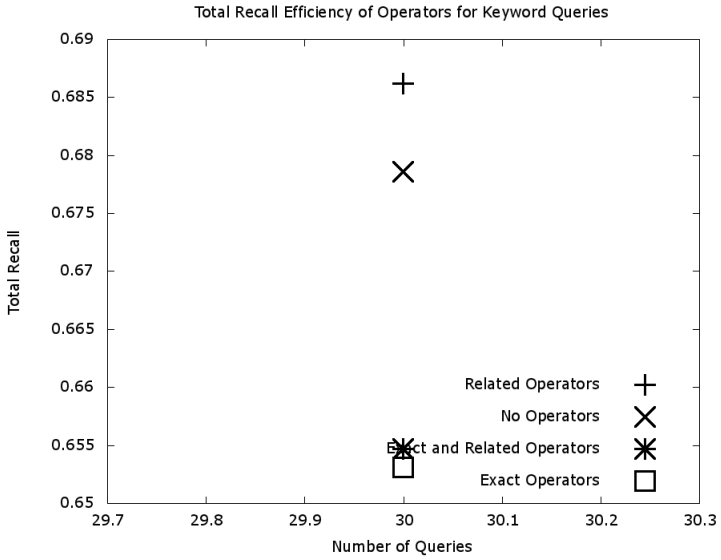


Figure 8: *Recall for Ordered Keyword Queries*

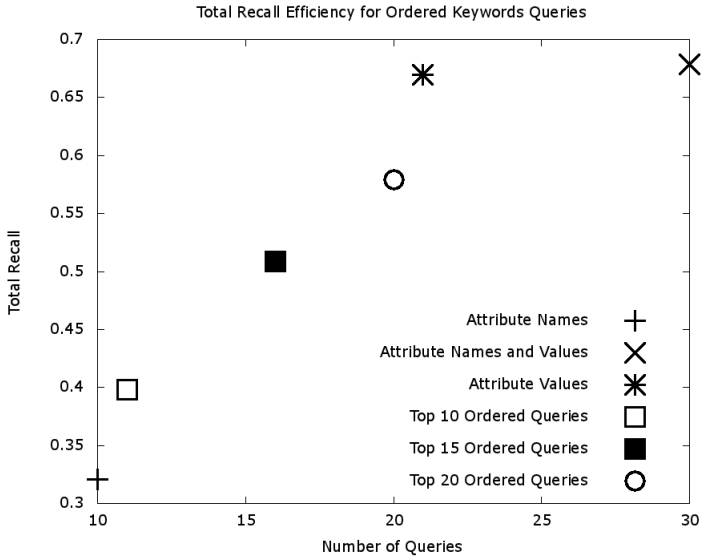
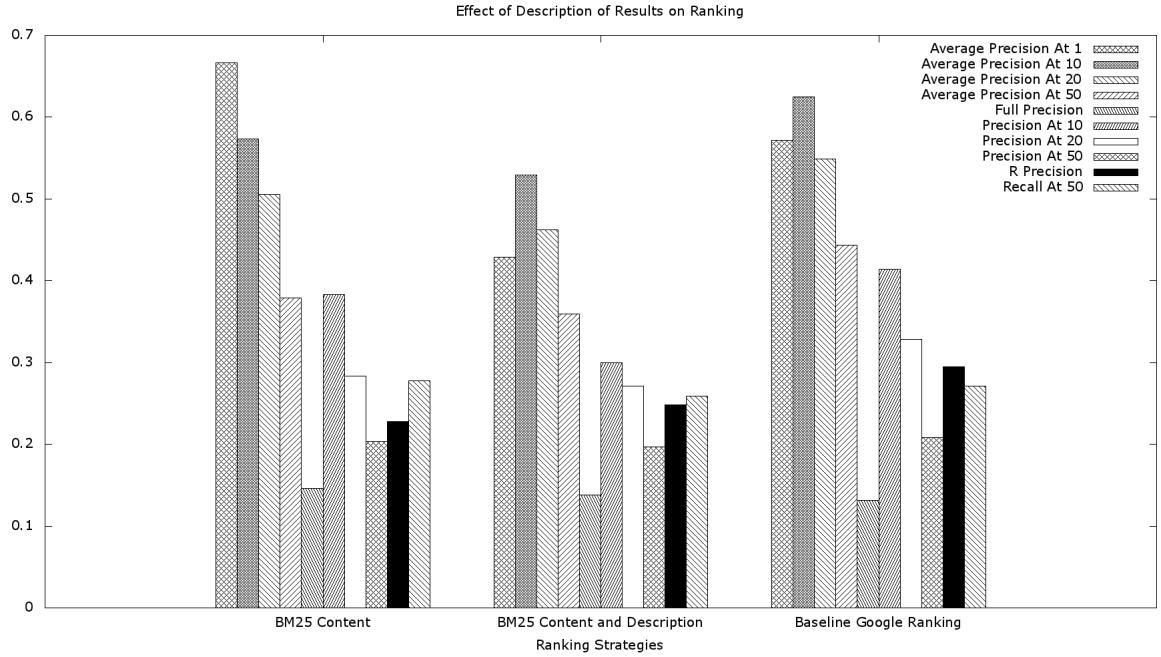


Figure 9: *Effect of Meta Description Feature on Ranking*



with the new feature, and compare the performance of this scheme to that of the original BM25 content scoring and out Google baseline. As we can see through each figure, the aggregate Google baseline score ranking significantly outperforms the basic BM25 content scoring scheme.

6.2.1 Single Ranking Features

Our experiments with meta tags, specifically the keywords and description tags of pages shown in figures 9 and 10, indicate that including keywords and descriptions actually hinders the performance of the ranking in our application. Unfortunately, our Yahoo keywords feature shows a similar decrease in performance across the board in Figure 11

On the other hand, our comparison for the PageRank feature shows that including PageRank improves the quality of the ranking overall. Although Figure 12 indicates a lower average precision @ 1, our results indicate improved average precision scores for the overall ranking of the results, and comparable performance by our other metrics. The results of our test introducing the title feature, shown in Figure 13, similarly demonstrates decreased performance in the *average precision at 1* metric, but improved performance in average precision scores in the overall ranking.

However, our the two most promising features appear to be the Yahoo Keywords expanded using WordNet, shown in figure 15, and headers text, shown in figure 14. Although neither ranking individually outperforms the baseline Google ranking, each demonstrates an excellent improvement over using BM25 scores based solely on the content of the page.

6.3 Summary

To achieve reasonable performance using our automated query generation system, we should be able to match the human standard in terms of the number of queries, and outperform the naive query generation strategy (using only the entity id) in terms of recall. As we show in Figure 16, we are able to outperform the recall of our naive query generation strategy by a factor of over 2.5, improving recall by over 0.3. Likewise, as we show in Figure 17, we even outperform the human baseline in terms of number of queries, requiring an average of one query fewer than the human baseline.

To outperform the established Google baseline, we must attempt to combine each of the features tested individually using machine learning. In Figure 18, we show the preliminary results for the coordinate descent ranking algorithm, which shows a promising improvement over the Google baseline for one of our test entities. This indicates that our strategy has the potential to outperform Google, using only a simple, naive learning algorithm for the ranking function. More sophisticated learning algorithms, such as RankSVM, would undoubtedly provide superior performance and generalizability for each entity domain.

Figure 10: *Effect of Meta Keywords Feature on Ranking*

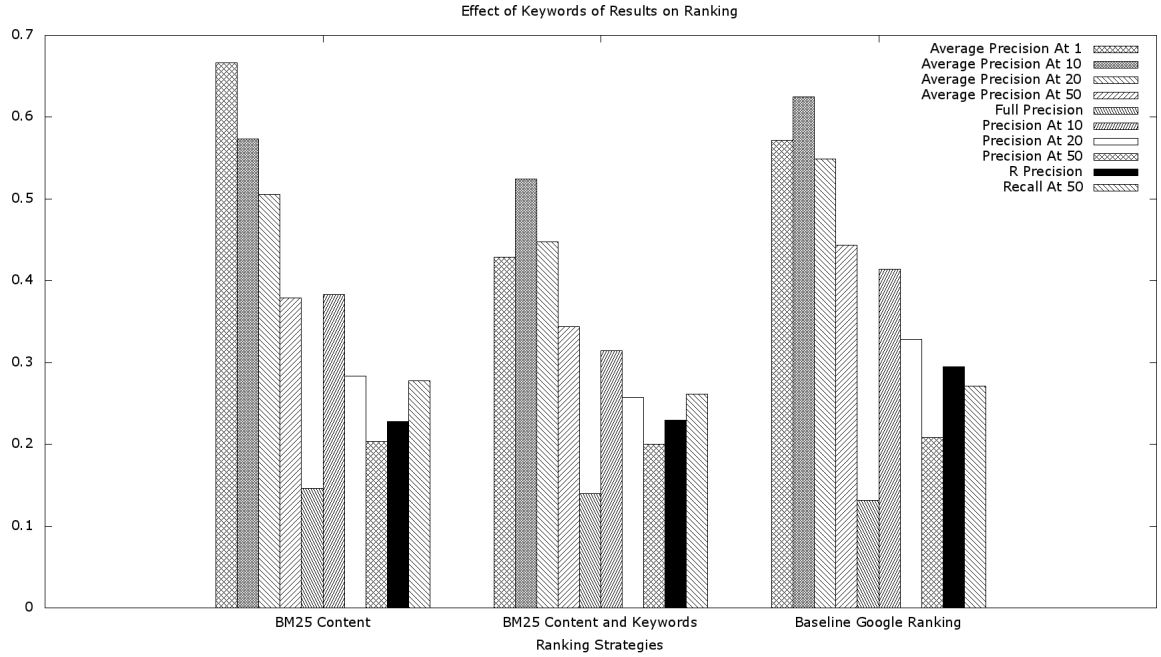


Figure 11: *Effect of Yahoo Keywords Feature on Ranking*

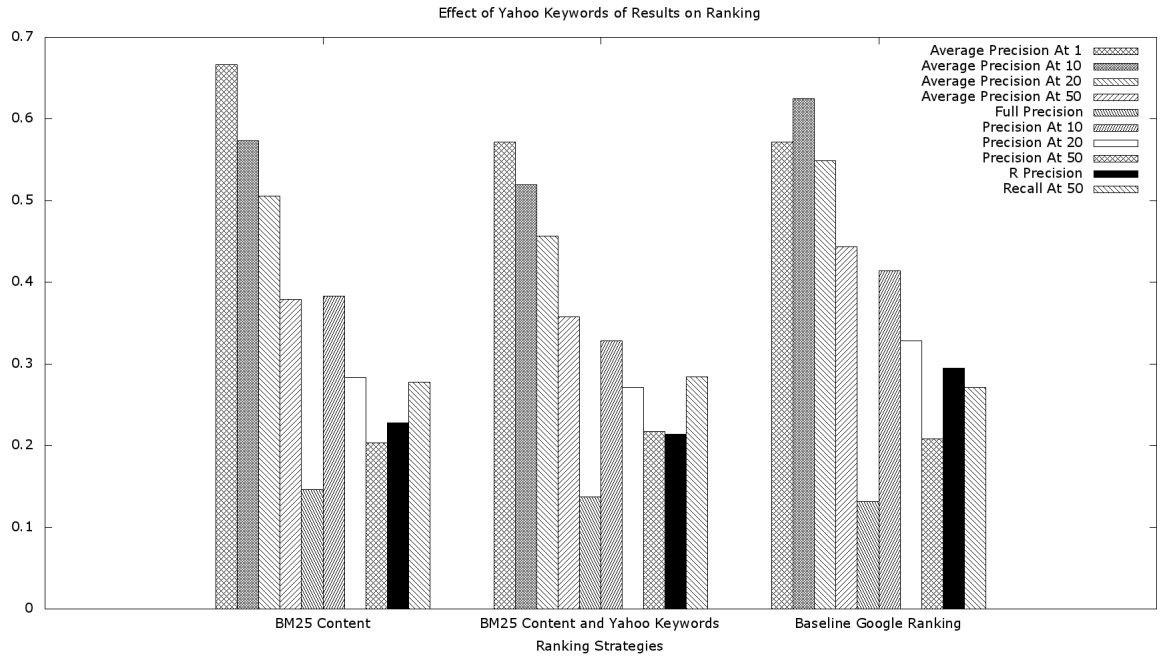


Figure 12: *Effect of PageRank Feature on Ranking*

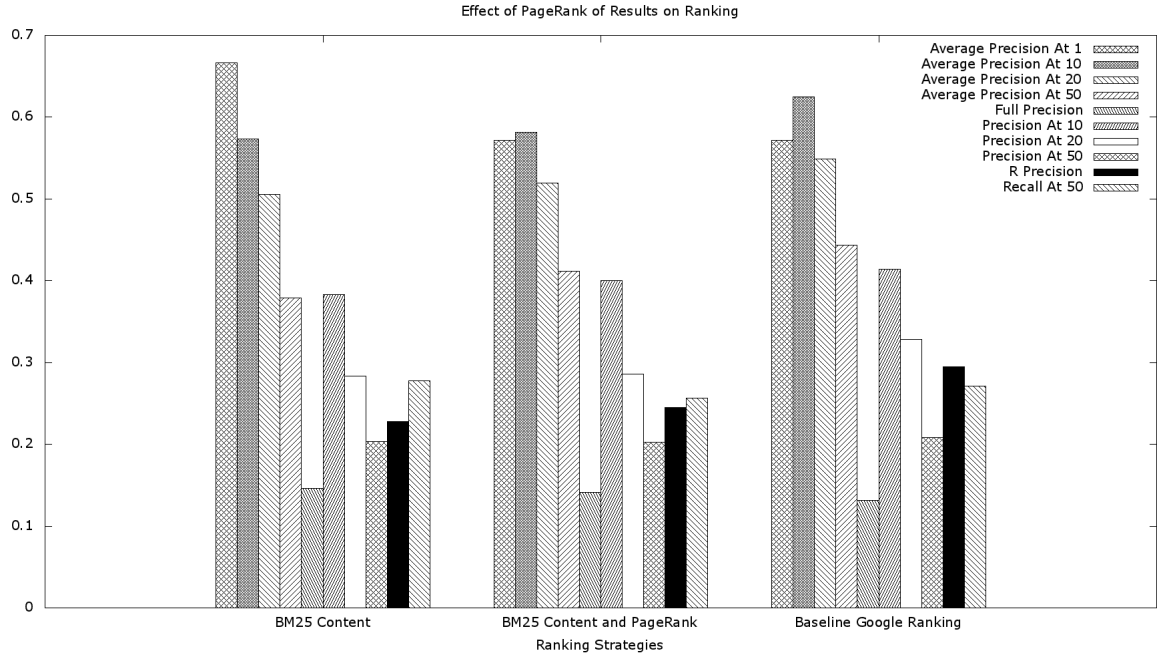


Figure 13: *Effect of Title Feature on Ranking*

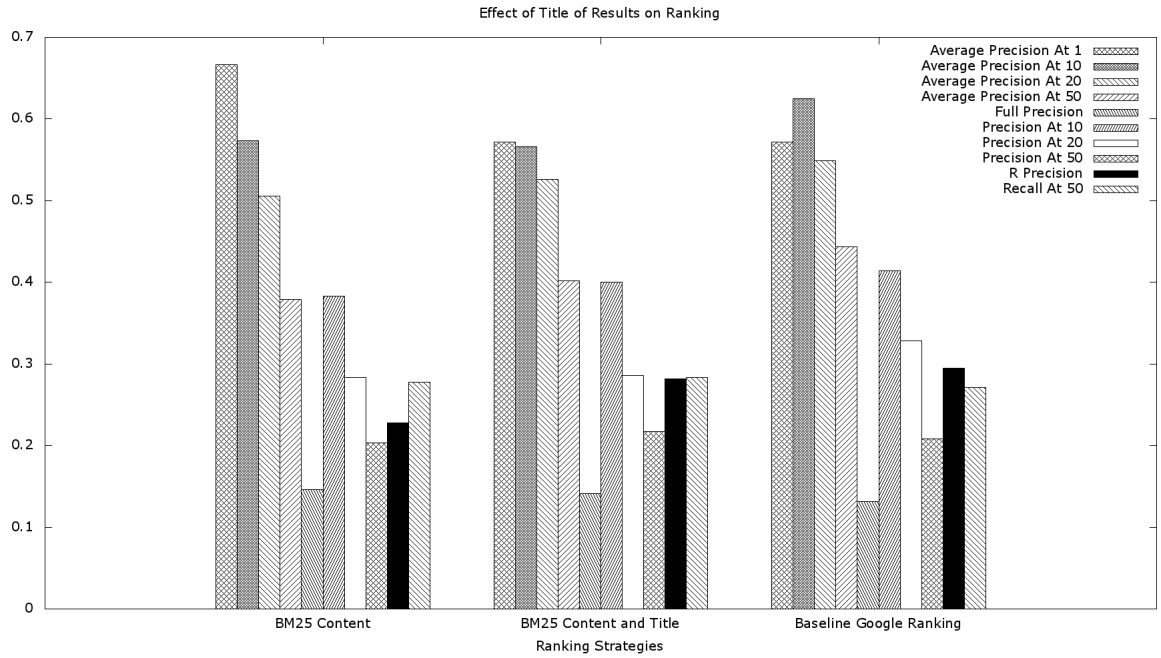


Figure 14: *Effect of Headers Feature on Ranking*

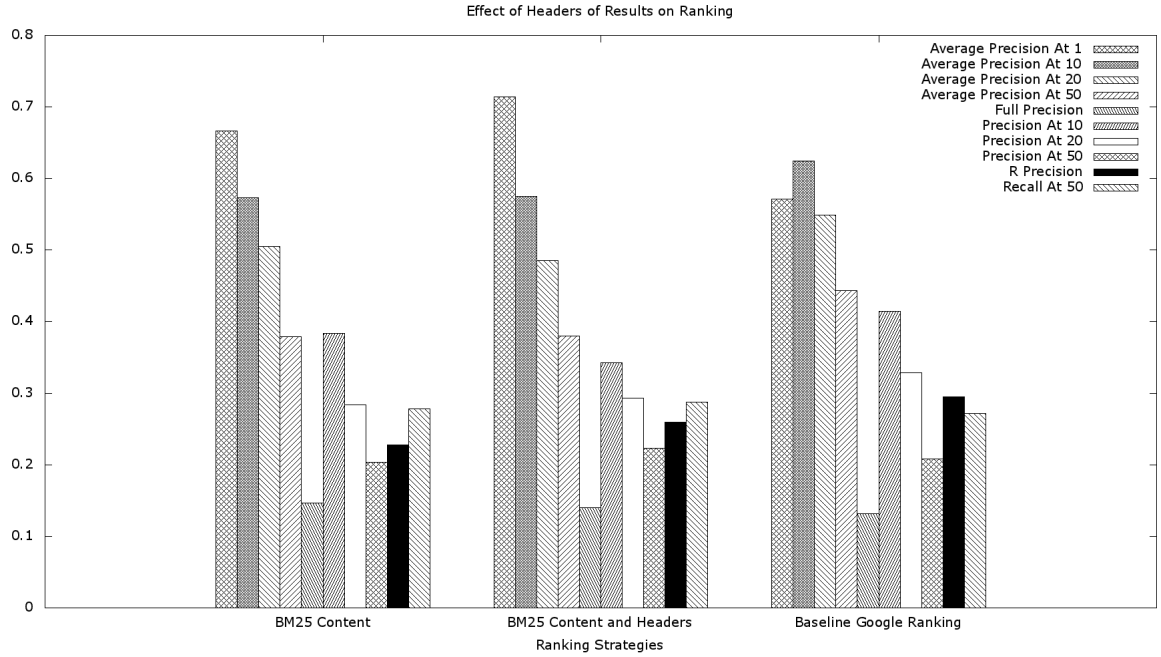


Figure 15: *Effect of Expanded Yahoo Keywords Feature on Ranking*

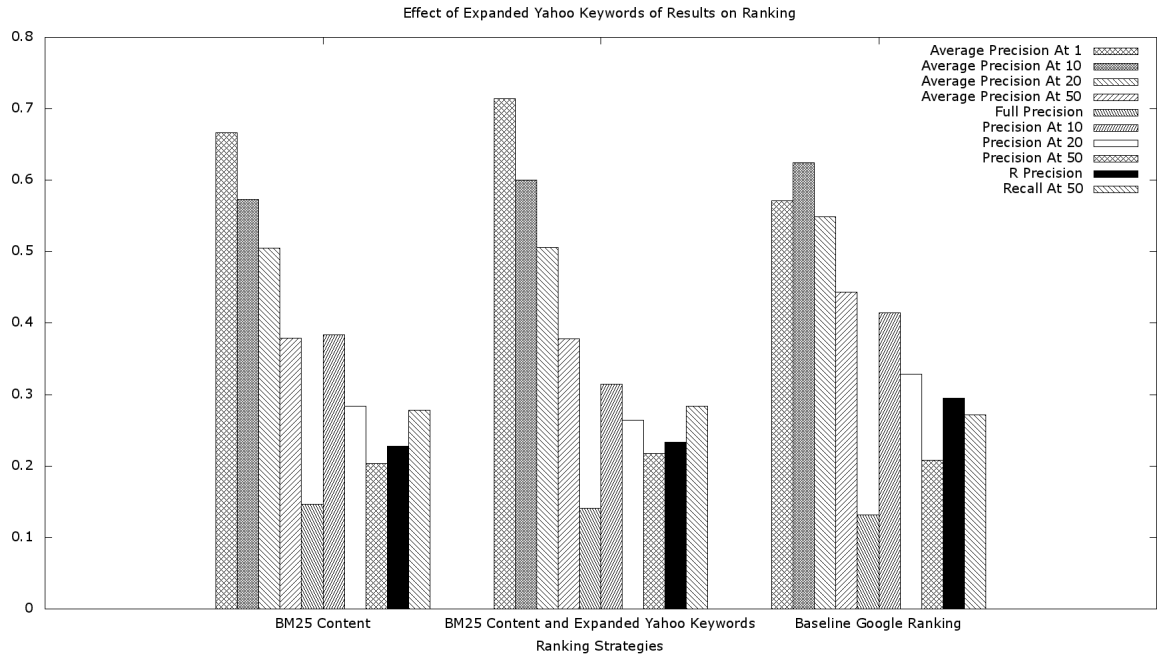


Figure 16: *Recall for Baseline Comparison*

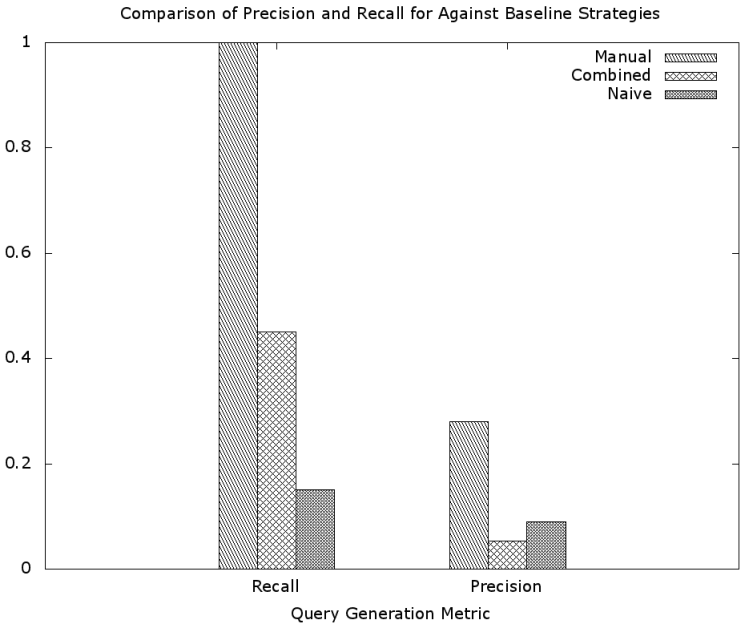


Figure 17: *Efficiency for Baseline Comparison*

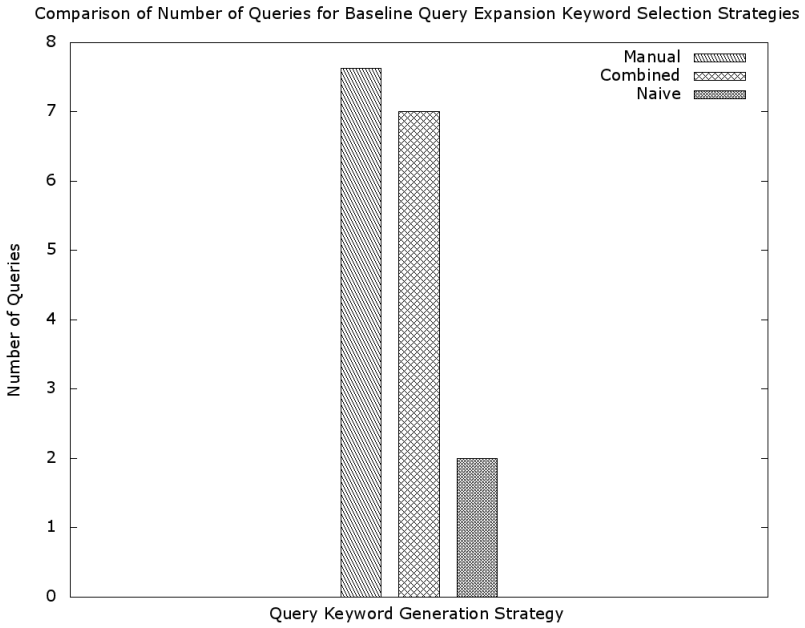
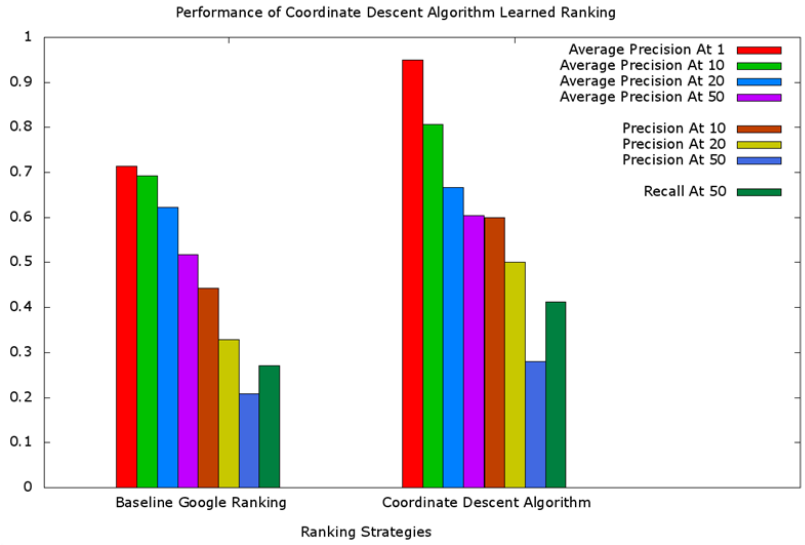


Figure 18: *Preliminary Results Using Coordinate Descent for One Entity*



This significantly improves over the existing technology in this area for several reasons:

- Problem Formalization

  This problem has not been formalized and addressed by current research, yet represents a critical problem to solve.

- Domain-Independence

  Our solution has been tested across two entity domains: professors and restaurants, which contain different properties based on their domain. For example, restaurants contain properties that contain numbers of incomplete words, such as phone numbers, addresses, cities, or zipcodes. On the other hand, professor entities contain primarily keyword attributes. The results of our strategies have been averaged across both domains, and thus represent a domain-independent solution to the problem.

- Efficiency and Performance

  Our solution uses fewer queries than the human baseline, and significantly outperforms our lower bound baseline, the naive query generation strategy, which submits the entity id as a single query.

- Ranking Performance

  Our results show that using a learning algorithm with these programatically generated queries, we can aggregate results from an existing search engine and outperform it in terms of both precision and recall by simply weighting features of each document from the aggregate results.

7 Conclusions

As web search matures, users are no longer satisfied to retrieve relevant pages as results to keyword queries. Users expect to find information about real-world entities directly, not indirectly through relevant URLs. While we have investigated how to retrieve relevant entities from keyword queries, little research has focused on the complementary problem of finding relevant pages from an entity.

Modern search engines fail to accomplish this task effectively, because users must manually transform entities into keyword queries and retrieve relevant results for this entity. This process is time-consuming, error prone, and extremely impractical. Likewise, modern lack the interface to allow users to easily and accurately express the entity instances through structured queries.

Our research addresses these issues by introducing a novel formalization of the domain and creating an automated search system that allows users to submit fully-qualified entities as queries and retrieve relevant pages for this entity in a domain-independent

way. Our system decomposes into two phases: query generation and ranking aggregation. During the query generation stage, our system uses the entity description to create keyword queries that represent the attributes of the entity, and submits each query to an existing search engine. In the second phase, our system aggregates the results of each query, reranks them, and returns them as the final output of the application to the user.

During the query generation stage, we attempt to maximize the total recall and number of queries, using manual relevance labels for a set of twenty entities from two domains. We investigate how to select keywords for query expansion, how to improve recall of these queries using query operators allowed by the search engine, and reorder these queries to submit the most effective queries first, allowing us to reduce the total number of queries. During the second phase, we attempt to maximize the precision of the final results and investigate the usefulness of text-based relevance scores for various aspects of the page, the PageRank score, and the ranking in the existing search engine results. Ultimately, we show that our system outperforms the results provided by Google on this task.

8 Future Work

While our research is certainly a step towards solving the entity search problem, there are several directions in which future research may progress.

First, we could introduce machine learning into our query generation phase. This could be accomplished by optimizing the number of queries and recall for a learned query generation strategy, and would undoubtedly improve our query generation phase. This learning stage could help select the best query generation model to use for any domain, or to rank the attributes of specific entity types.

Second, we could investigate the effects, if any, of querying additional existing search engines to gather and aggregate results. Likewise we could adapt our system to use existing meta search engines.

Finally, we could introduce domain-dependent optimizations for both the query generation phase and the aggregate ranking phase, such as additional keywords relevant to all entities in a particular domain, or binary features to the final results specific to a domain. For instance, for the professor domain we investigate, it may be advantageous to perform query expansion using keywords that are relevant to all professors or to always include university and location keywords for a professor to disambiguate between similar professors with at different institutions. Likewise, we could increase the weight of results that come from academic domains or even parse binary files to search for papers or presentations from a professor.

9 Acknowledgments

We would like to thank our faculty adviser, Kevin Chang, for his invaluable help and guidance with our project.

References

AGICHTEIN, E., AND GRAVANO, L. 2003. Querying text databases for efficient information extraction. In *In Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE*, 113–124.

AGICHTEIN, E., ESKIN, E., AND GRAVANO, L. 2000. Combining strategies for extracting relations from text collections. In *In Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*.

AGICHTEIN, E., 2007. Question answering over implicitly structured web content.

ANAGNOSTOPOULOS, A., BECCHETTI, L., CASTILLO, C., AND GIONIS, A. An optimization framework for query recommendation. In *WSDM '10 Proceedings of the third ACM international conference on Web search and data mining*.

CHENG, T., YAN, X., AND CHANG, K. C.-C. Entity search engine: Towards agile best-effort information integration over the web. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar*.

CHENG, T., YAN, X., AND CHANG, K. C.-C. 2007. Entityrank: Searching entities directly and holistically. In *In VLDB*, 387–398.

COHEN, W. W., SCHAPIRE, R. E., AND SINGER, Y. 1998. Learning to order things. *Journal of Artificial Intelligence Research 10*, 243–270.

DMOZ, 2011. http://dmoz.org.

ETZIONI, O., CAFARELLA, M., DOWNEY, D., KOK, S., POPESCU, A.-M., SHAKED, T., SODERLAND, S., WELD, D. S., AND YATES, A. Web-scale information extraction in knowitall: (preliminary results). In *WWW '04 Proceedings of the 13th international conference on World Wide Web*.

GOOGLE, 2011. http://www.google.com.

JIANG, Y., YANG, H.-T., CHANG, K. C.-C., AND CHEN, Y.-S. Aide: Ad-hoc intents detection engine over query logs. In *SIGMOD Conference*.

JOACHIMS, T. 2002. Optimizing search engines using clickthrough data. In *KDD '02 Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*.

JOACHIMS, T. 2006. Training linear svms in linear time. In *KDD '06 Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*.

LIU, J. 2006. Answering structured queries on unstructured data. In *In WebDB*, 25–30.

MIANWEI ZHOU, TAO CHENG, K. C.-C. C. Data-oriented content query system: searching for data into text on the web. In *WSDM '10 Proceedings of the third ACM international conference on Web search and data mining*.

NIE, Z., WU, F., WEN, J.-R., AND MA, W.-Y. Extracting objects from the web.

ROBERTSON, S. E. On term selection for query expansion. In *Journal of Documentation*.

SVMRANK, 2011. http://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html.

SYNTACTIC, G., AND OF QUERY REFORMULATION, S. M. Maria holmqvist and massimiliano ciaramita and daniel mahler and stefan riezler.

WHOOSH, 2011. https://bitbucket.org/mchaput/whoosh/wiki/home.

YQL, Y., 2011. http://developer.yahoo.com/yql/.