OBJECT SEARCH: SUPPORTING STRUCTURED QUERIES IN WEB SEARCH ENGINES

BY

CUONG KIM PHAM

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor Kevin Chen-Chuan Chang

Abstract

As the web evolves, increasing quantities of structured information is embedded in web pages in disparate formats. For example, a digital camera's description may include its *price* and *megapixels* whereas a professor's description may include her *name*, *university*, and *research interests*. Both types of pages may include additional ambiguous information. General search engines (GSEs) do not support queries over these types of data because they ignore the web document semantics. Conversely, describing requisite semantics through structured queries into databases populated by information extraction (IE) techniques are expensive and not easily adaptable to new domains. This paper describes a methodology for rapidly developing search engines capable of answering structured queries over unstructured corpora by utilizing machine learning to avoid explicit IE. We empirically show that with minimum additional human effort, our system outperforms a GSE with respect to structured queries with clear object semantics.

To My Parents ...

Acknowledgments

I would like to thank my advisor, Prof. Kevin Chen-Chuan Chang, for providing me great guidance and motivation. I would also like to thank Prof Dan Roth, for giving me invaluable discussions and support.

Table of Contents

Chapter 1

Introduction

General search engines (GSEs) are sufficient for fulfilling the information needs of most queries. However, they are often inadequate for retrieving web pages that concisely describe real world objects as these queries require analysis of both unstructured text and structured data contained in web pages. For example, digital cameras with specific *brand*, *megapixel*, *zoom*, and *price* attributes might be found on an online shopping website, or a professor with her *name*, *university*, *department*, and *research interest* attributes might be found on her homepage. Correspondingly, as the web continues to evolve from a general text corpus into a heterogeneous collection of documents, targeted retrieval strategies must be developed for satisfying these more precise information needs. We accomplish this by using structured queries to capture the intended semantics of a user query and learning domain specific ranking functions to represent the hidden semantics of object classes contained in web pages.

It is not uncommon for a user to want to pose an *object query* on the web. For example, an online shopper might be looking for *shopping pages* that sell *canon* digital cameras with 5 megapixels costing no more than *$300*. A graduate student might be looking for *homepages* of *computer science* professors who work in the *information retrieval* area. Such users expect to get a list web pages containing objects they are looking for, or *object pages*, which we will define more precisely in later chapters.

GSEs rarely return satisfactory results when the user has a structured query in mind for two primary reasons. Firstly, GSEs only handle keyword queries whereas structured queries frequently involve data field semantics (e.g. numerical constraints) and exhibit field interdependencies. Secondly, since GSEs are domain-agnostic, they will generally rank camera pages utilizing the same functions as a professor's homepage, ignoring much of the structured information specific to particular domains.

Conversely, vertical search engines (e.g. DBLife, cazoodle.com, Rexa.info, etc.) approach this problem from the information extraction (IE) perspective. Instead of searching an inverted index directly, they first extract data records from text [10, 13]. IE solutions, even with large scale techniques [1], do not scale to the *entire* web and cost significantly more than GSEs. Secondly, creating domain-specific models or wrappers require labeling training examples and

human expertise for each individual site. Thirdly, pre-extracting information lacks flexibility; decisions made during IE are irrevocable, and at query time, users may find additional value in partial or noisy records that were discarded by the IE system.

These issues motivate our novel approach for designing a GSE capable of answering complex structured queries, which we refer to as *Object Search*. At a high level, we search web pages containing structured information directly over their feature index, similarly to GSEs, adding expressivity by reformulating the structured query such that it can be executed on a traditional inverted index. Thus, we avoid the expense incurred by IE approaches when supporting new object domains. From a technical perspective, this work describes a principled approach to customizing GSEs to answer structured queries from any domain by proposing a compositional ranking model for ranking web pages with regards to structured queries and presenting an interactive learning approach that eases the process of training for a new domain.

Chapter 2

Object Search Problem

2.1 The Object Search Problem

The Object Search problem is to find the *object pages* that answer a user's *object query*. An object query belongs to an *object domain*. An object domain defines a set of object attributes. An object query is simply a set of constraints over these attributes. Thus we define an object query as a tuple of n constraints $q \equiv c_1 \wedge c_2 \wedge .. \wedge c_n$, where c_i is a constraint on attribute a_i. More specifically, a constraint c_i is defined as a set of acceptable values θ_i for attribute a_i; i.e. $c_i = (a_i \in \theta_i)$. For example, an equality constraint such as "the brand is Canon" can be specified as $(a_{brand} \in \{Canon\})$ and a numeric range constraint such as "the price is at most $200" can be specified as $(a_{price} \in [0, 200])$. When the user does not care about an attribute, the constraint is the constant *true*.

Given an object query, we want a set of satisfying object pages. Specifically, object pages are pages that represent exactly one inherent object on the web. Pages that list several objects such as a department directory page or camera listing pages are not considered object pages because even though they mentioned the object, they do not represent any particular object. There is often a single object page but there are many web pages that mention the object.

The goal of Object Search is similar to learning to rank problems [12], in that its goal is to learn a ranking function $\rho : \mathcal{D} \times \mathcal{Q} \rightarrow \mathcal{R}$ that ranks any $(document, query)$ pairs. This is accomplished by learning an function over a set of relevant features. Each feature can be modeled as a function that takes the pair and outputs a real value $\phi : \mathcal{D} \times \mathcal{Q} \rightarrow \mathcal{R}$. For example, a *term frequency* feature outputs the number of times the query appears in the document. We define a function $\Phi = (\phi_1, \phi_2, ...\phi_n)$ that takes a $(document, query)$ pair and outputs a vector of numeric features. The original ranking function can be written as $\rho(d, q) = \rho'(\Phi(d, q))$ where $\rho' : \mathcal{R}^n \rightarrow \mathcal{R}$ is a real function; i.e.:

$$\rho \quad = \quad \rho' \circ \Phi \tag{2.1}$$

Despite the similarities, Object Search differs from traditional information retrieval (IR) problems in many respects. First, IR can answer only keyword queries whereas an object query is structured by keyword constraints as well as numeric constraints. Second, Object Search results are "focused", in the sense

that they must contain an object, as opposed to the broad notion of *relevance* in IR. Finally, since object pages of different domains might have little in common, we cannot apply the same ranking function for different object domains.

As a consequence, in a learning to rank problem, the set of features Φ are fixed for all query. The major concern is learning the function ρ'. In Object Search settings, we expect different Φ for each object domain. Thus, we have to derive both Φ and ρ'.

There are a number of challenges in solving these problems. First, we need a deeper understanding of structured information embedded in web pages. In many cases, an object attribute such as professor's university might appear only once in his homepage. Thus, using a traditional bag-of-words model is often insufficient, because one cannot distinguish the professor own university from other university mentioned in his homepage. Second, we will need training data to train a new ranking function for each new object domain. Thus, we require an efficient bootstrapping method to tackle this problem. Finally, any acceptable solution must scale to the size of the web. This requirement poses challenges for efficient query processing and efficient ranking via the learned ranking function.

Chapter 3

Object Search Framework

In this chapter, we illustrate the primary intuitions behind our aproach for an Object Search solution. We describe its architecture, which serves as a search engine framework to support structured queries of any domain. The technical details of major components are left for subsequent chapters.

3.1 Intuition

The main idea behind our proposed approach is that we develop different vertical search engines to support object queries in different domains. However, we want to keep the cost of supporting each new domain as small as possible. The key principles to keep the cost small are to 1) share as much as possible between search engines of different domains and 2) automate the process as much as possible using machine learning techniques. To illustrate our proposed approach, we suppose that an user is searching the web for cameras. Her object query is $q = a_{brand} \in \{canon\} \wedge a_{price} \in [0, 200]$.

First, we have to automatically learn a function ρ that ranks web pages given an object query as described in Section 2.1. We observe relevant object pages and notice several salient features such as "the word *canon* appears in the title", "the word *canon* appears near *manufacturer*", "interesting words that appear include *powershot*, *eos*, *ixus*", and "a price value appears after '$' near the word *price* or *sale*". Intuitively, pages containing these features have a much higher chance of containing the Canon camera being searched. Given labeled training data, we can learn a ranking function that combines these features to produce the probability of a page containing the desired camera object.

Furthermore, we need to answer user query at query time. We need to be able to look up these features efficiently from our index of the web. A naïve method to index the web is to store a list of web pages that have the above features, and at query time, union all pages that have one or more features, aggregate the score for each web page, and return the ranked result. There are three problems with this method. First, these features are dependent on each object domain; thus, the size of the index will increase as the number of domains grows. Second, each time a new domain is added, a new set of features needs to be indexed, and we have to extract features for every single web page

again. Third, we have to know beforehand the list of camera brands, megapixel ranges, price ranges, etc, which is infeasible for most object domain.

However, we observe that the above query dependent features can be computed efficiently from a query independent index. For example, whether "the word *canon* appears near *manufacturer*" can be computed if we index all occurrences of the words *canon* and *manufacturer*. Similarly, the feature "the word *canon* appears in the title" can be computed if we index all the words from web pages' title, which only depends on the web pages themselves. Since the words and numbers from different parts of a web page can be indexed independently of the object domain, we can share them across different domains. Thus, we follow the first principle mentioned above.

Of course, computing query dependent features from the domain independent index is more expensive than computing it from the naïve index above. However, this cost is scalable to the web. As a matter of fact, these features are equivalent to "phrase search" features in modern search engines.

Thus, at a high level, we solve the Object Search problem by learning a domain dependent ranking function for each object domain. We store basic domain independent features of the web in our index. At query time, we compute domain dependent features from this index and apply the ranking function to return a ranked list of web pages. In this paper, we focus on the learning problems, leaving the problem of efficient query processing for future work.

3.2 System Architecture

The main goal of our Object Search system is to enable searching the web with object queries. In order to do this, the system must address the challenges described in Section 2.1. From the end-user's point of view, the system must promptly and accurately return web pages for their object query. From the developer's point of view, the system must facilitate building a new search engine to support his object domain of interest. The goal of the architecture is to orchestrate all of these requirements.

Figure 3.1 depicts Object Search architecture. It shows how different components of Object Search interact with an end-user and a developer. The end-user can issue any object query of known domains. Each time the system receives an object query from the end-user, it translates the query into a domain independent feature query. Then the Query Processor executes the feature query on the inverted index, aggregates the features using learned function ρ', and returns a ranked list of web pages to the user.

The developer's job is to define his object domain and train a ranking function for it. He does it by incrementally training the function. He starts by annotating a few web pages and running a learning algorithm to produce a ranking function, which is then used to retrieve more data for the developer to annotate. The process iterates until the developer is satisfied with his trained

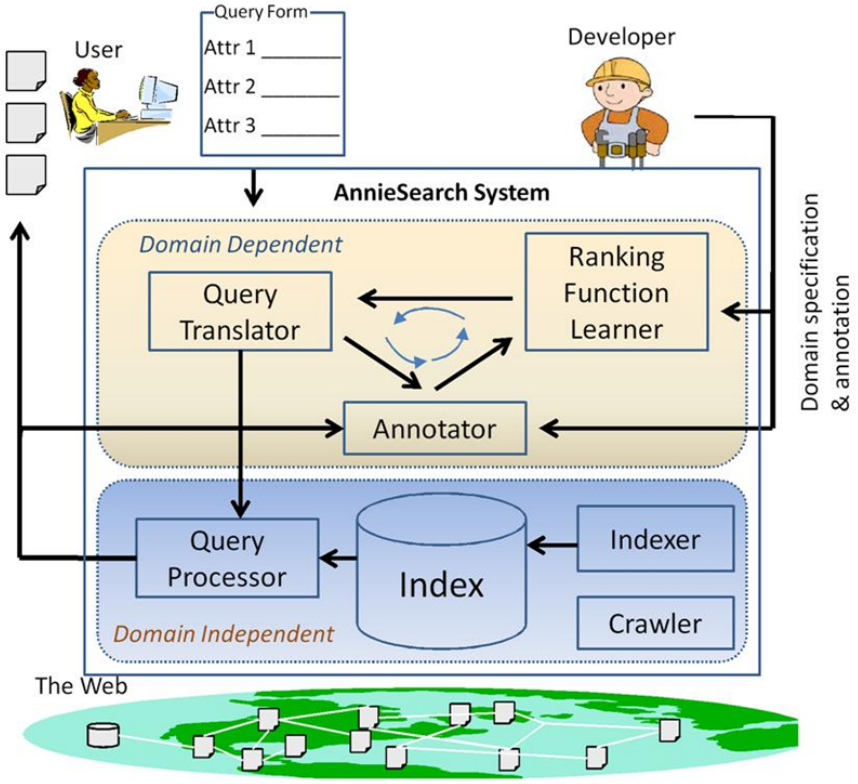

Figure 3.1: Object Search Architecture

ranking function for the object domain.

More specifically, the Ranking Function Learner module learns the function ρ' and Φ as mentioned in Section 2.1. The Query Translator instantiates Φ with user object query q, resulting in $\Phi(q)$. Recall that Φ is a set of feature functions ϕ_i. Each ϕ_i is a function of a (d, q) pair such as "term frequency of a_k in title" (a_k is an attribute of the object). Thus we can instantiate $\phi(q)$ by replacing a_k with θ_k, which is part of the query q. For example, if $\theta_k = \{canon\}$ in the previous example, then $\phi(q)$ is "term frequency of *canon* in title". Thus $\phi(q)$ becomes a query independent feature and $\Phi(q)$ becomes a feature query that can be executed with our domain independent inverted index by the Query Processor.

Chapter 4

Learning for Structured Ranking

We now describe how we learn the domain dependent ranking function ρ, which is the core learning aspect of Object Search. As mentioned in the previous section, ρ differs from existing learning to rank work due to the structure in object queries. We exploit this structure to decompose the ranking function into several components (Section 4.1) and combine them using a probabilistic model. Existing learning to rank methods can then be leveraged to rank the individual components. Section 4.2 describes how we fit individual ranking scores into our probabilistic model by calibrating their probability.

4.1 Ranking Model

As stated, ρ models the joint probability distribution over the space of documents and queries $\rho = P(d, q)$. Once estimated, this distribution can rank documents in $\mathcal{D}$ according to their probability of satisfying q. Since we are only interested in finding satisfying object pages, we introduce a variable ω which indicates if the document d is an object page. Furthermore, we introduce n variables ζ_i which indicate whether constraint c_i in the query q is satisfied. The probability computed by ρ is then:

$$\begin{aligned}
P(d,q) &= P(\zeta_1,\ldots,\zeta_n,d) \\
&= P(\zeta_1,\ldots,\zeta_n,d,\omega) \\
&\quad +P(\zeta_1,\ldots,\zeta_n,d,\overline{\omega}) \\
&= P(d)P(\omega|d)P(\zeta_1,\ldots,\zeta_n|d,\omega) \\
&\quad +P(d)P(\overline{\omega}|d)P(\zeta_1,\ldots,\zeta_n|d,\overline{\omega}) \\
&= P(d)P(\omega|d)P(\zeta_1,\ldots,\zeta_n|d,\omega) \qquad (4.1) \\
&\simeq P(\omega|d)\prod_{i=1}^{n} P(\zeta_i|d,\omega) \qquad (4.2)
\end{aligned}$$

Equation 4.1 holds because non-object pages do not satisfy the query, thus, $P(\zeta_1,\ldots,\zeta_n|d,\overline{\omega}) = 0$. Equation 4.2 holds because we assume a uniform distribution over d and conditional independence over ζ_i given d and ω.

Thus, the rest of the problem is estimating $P(\omega|d)$ and $P(\zeta_i|d,\omega)$. The difference between these probability estimates lies in the features we use. Since

ω depends only in d but not q, we use query independent features. Similarly, ζ_i only depends on d and c_i, thus we use features depending on c_i and d.

4.2 Calibrating Ranking Probability

In theory, we can use any learning algorithm mentioned in [12]'s survey to obtain the terms in Equation 4.2. In practice, however, such learning algorithms often output a ranking score that does not estimate the probability. Thus, in order to use them in our ranking model, we must transform that ranking score into a probability.

For empirical purposes, we use the *averaged* Perceptron [6] to discriminatively train each component of the factored distribution independently. This algorithm requires a set of input vectors, which we obtain by applying the relational feature functions to the paired documents and queries. For each constraint c_i, we have a feature vector $\mathbf{x}_i = \Phi_i(d, q)$. The algorithm produces a weight vector of parameters $\mathbf{w}_i$ as output. The probability of c_i being satisfied by d given that d contains an object can then be estimated with a sigmoid function as:

$$P(c_i|d, \omega) \equiv P(true|\Phi_i(d, q)) \equiv \frac{1}{1 + exp(-\mathbf{w}_i^\mathsf{T}\mathbf{x}_i)} \tag{4.3}$$

Similarly, to estimate $P(\omega|d)$, we use a feature vector that is dependent only on d. Denoting the function as Φ_0, we have $P(\omega|d) = P(true|\Phi_0(d, q))$, which can be obtained from (4.3).

While the sigmoid function has performed well empirically, probabilities it produces are not calibrated. For better calibrated probabilities, one can apply Platt scaling [14]. This method introduces two parameters A and B, which can be computed using maximum likelihood estimation:

$$P(true|\Phi_i(d, q)) \equiv \frac{1}{1 + exp(A\mathbf{w}_i^\mathsf{T}\Phi_i(d, q) + B)} \tag{4.4}$$

In contrast to the sigmoid function, Platt scaling can also be applied to methods that give un-normalized scores such as RankSVM [4].

Substituting (4.3) and (4.4) into (4.2), we see that our final learned ranking function has the form

$$\rho(d, q) = \prod_{i=0}^{n} \frac{1}{(1 + exp(A_i\mathbf{w}_i^\mathsf{T}\Phi_i(d, q) + B_i))} \tag{4.5}$$

Chapter 5

Learning Based Programming

Learning plays a crucial role in developing a new object domain. In addition to using supervised methods to learn ρ, we also exploit active learning to acquire training data from unlabeled web pages. The combination of these efforts would benefit from a unified framework and interface to machine learning. Learning Based Programming (LBP) [16] is such a principled framework. In this section, we describe how we applied and extended LBP to provide a user friendly interface for the developer to specify features and guide the learning process. Section 5.1 describes how we structured our framework around Learning Based Java (LBJ), an instance of LBP. Section 5.2 extends the framework to support interactive learning.

5.1 Learning Based Java

LBP is a programming paradigm for systems whose behaviors depend on naturally occurring data and that require reasoning about data and concepts in ways that are hard, if not impossible, to write explicitly. This is exactly our situation. Not only do we not know how to specify a ranking function for an object query, we might not even know exactly what features to use. Using LBP, we can specify abstract information sources that might contribute to decisions and apply a learning operator to them, thereby letting a learning algorithm figure out their importances in a data-driven way.

Learning Based Java (LBJ) [15] is an implementation of LBP which we used and extended for our purposes. The most useful abstraction in LBJ is that of the *feature generation function* (FGF). This allows the programmer to reason in terms of feature *types*, rather than specifying individual features separately, and to treat them as native building blocks in a language for constructing learned functions. For example, instead of specifying individual features such as the phrases "professor of","product description", etc., we can specify a higher level feature type called "bigram", and let an algorithm select individual features for ranking purposes.

From the programming point of view, LBJ provides a clean interface and abstracts away the tedium of feature extraction and learning implementations. This enabled us to build our system quickly and shorten our development cycle.

5.2 Interactive Machine Learning

We advocate an interactive training process [5], in which the developer iteratively improves the learner via two types of interaction (Algorithm 5.2.1).

The first type of interaction is similar to active learning where the learner presents unlabeled instances to the developer for annotation which it believes will most positively impact learning. In ranking problems, top ranked documents are presented as they strongly influence the loss function. The small difference from traditional active learning in our setting is that the developer assists this process by also providing more queries other than those encountered in the current training set.

The second type of interaction is feature selection. We observed that feature selection contributed significantly in the performance of the learner especially when training data is scarce. This is because with little training data and a huge feature space, the learner tends to over-fit. Fortunately in web search, the features used in ranking are in natural language and thereby intuitive to the developer. For example, one type of feature used in ranking the *university* constraint of a professor object query is the words surrounding the query field as in "university of ..." or "... university". If the learner only sees examples from the University of Anystate at Anytown, then it's likely that *Anytown* will have a high weight in addition to *University* and *of*. However, the *Anytown* feature will not generalize for documents from other universities. Having background knowledge like this, the developer can unselect such features. Furthermore, the fact that *Anytown* has a high weight is also an indication that the developer needs to provide more examples of other universities so that the learner can generalize (the first type of interaction).

Algorithm 5.2.1 Interactive Learning Algorithm

1: The developer uses keyword search to find and annotate an initial training set.
2: The system presents a ranked list of features computed from labeled data.
3: The developer adds/removes features.
4: The system learns the ranking function using selected features.
5: The developer issues queries and annotates top ranked unlabeled documents returned by the system.
6: If performance is not satisfactory, go to step 2.

The iterative algorithm starts with zero training data and continues until the learner's performance reaches a satisfactory point. At step 2, the developer is presented with a ranked list of features. To determine which features played the biggest role in the classifier's decision making, we use a simple ranking metric called *expected entropy loss* [7]. Let f represent the event that a given feature is active. Let C be the event that the given example is classified as *true*. The conditional entropy of the classification distribution given that f occurs

is $H(C|f) \equiv -P(C|f)\log(P(C|f)) - P(\overline{C}|f)\log(P(\overline{C}|f)$ and similarly, when f does not occur, we replace f by $\overline{f}$. The expected entropy loss is

$$\begin{aligned} L(C|f) &\equiv H(C) - \mathrm{E}[H(C|f)] \\ &= H(C) - (P(f)H(C|f) + \\ &\quad P(\overline{f})H(C|\overline{f}) \end{aligned} \tag{5.1}$$

The intuition here is that if the classification loses a lot of entropy when conditioned on a particular feature, that feature must be very discriminative and correlated with the classification itself.

It is noted that feature selection plays two important roles in our framework. First, it avoids over-fitting when training data is scarce, thus increasing the effectiveness of our active learning protocol. Second, since search time depends on how many features we use to query the web pages, keeping the number of features small will ensure that searching is fast enough to be useful.

Chapter 6

Experiments

In this chapter we present an experiment that compares Object Search with keyword search engines.

6.1 Experimental Setting

Since we are the first to tackle this problem of answering structured query on the web, there is no known dataset available for our experiment. We collected the data ourselves using various sources from the web. Then we labeled search results from different object queries using the same annotation procedure described in Section 5.

We collected URLs from two main sources: the open directory (DMOZ) and existing search engines (SE). For DMOZ, we included URLs from relevant categories. For SE, we manually entered queries with keywords related to professors' homepages, laptops, and digital cameras, and included all returned URLs. Having collected the URLs, we crawled their content and indexed them. Table 6.1 summarizes web page data we have collected.

We split the data randomly into two parts, one for training and one for testing, and created a single inverted index for both of them. The developer can only see the training documents to select features and train ranking functions. At testing time, we randomly generate object queries, and evaluate on the testing set. Since Google's results come not from our corpus but the whole web, it might not be fair to compare against our small corpus. To accommodate this, we also added Google's results into our testing corpus. We believe that most 'difficult' web pages that hurt Google's performance would have been included in the top Google result. Thus, they are also available to test ours. In the future, we plan to implement a local IR engine to compare against ours and

domain	# pages	train	test
homepage	22.1	11.1	11
laptop	21	10.6	10.4
camera	18	9	9
random	97.8	48.9	48.8
total	158.9	79.6	79.2

Table 6.1: Number of web pages (in thousands) collected for experiment

Field	Keywords	Example
Laptop domain		
brand	laptop,notebook	*lenovo* laptop
processor	ghz, processor	2.2 ghz
price	$, price	$1000..1100
Professor domain		
name	professor, research professor, faculty	research professor *scott*
university	university, university of	$stanford$ university

Table 6.2: Sample keyword reformulation for Google

Qry	Professor		Laptop	
	OSE	Google	OSE	Google
1	**0.92** (71)	0.90(65)	**0.7** (15)	0.44 (12)
2	0.83(88)	**0.91**(73)	**0.62** (12)	0.26 (11)
3	0.51(73)	**0.66**(48)	**0.44** (40)	0.31 (24)
4	**0.42**(49)	0.3(30)	**0.36** (3)	0.09 (1)
5	**0.91**(18)	0.2(16)	**0.77** (17)	0.42 (3)

Table 6.3: Average precision for 5 random queries. The number of positive documents are in brackets

conduct a larger scale experiment to compare to Google.

We evaluated the experiment with two different domains: professor and laptop. We consider homepages and online shopping pages as object pages for the professor and laptop domains respectively.

For each domain, we generated 5 random object queries with different field configurations. Since Google does not understand structured queries, we reformulated each structured query into a simple keyword query. We do so by pairing the query field with several keywords. For example, a query field $a_{brand} \in \{lenovo\}$ can be reformulated as "lenovo laptop". We tried different combinations of keywords as shown in table 6.2. To deal with numbers, we use Google's advanced search feature that supports numeric range queries[1]. For example, a *price* constraint $a_{price} \in [100, 200]$ might be reformulated as "price $100..200". Since it is too expensive to find the best keyword formulations for every query, we picked the combination that gives the best result for the first Google result page (Top 10 URLs).

6.2 Result

We measure the ranking performance with average precision. Table 6.3 shows the results for our search engine (OSE) and Google. Our ranking function outperforms Google for most queries, especially in the laptop domain. In the

[1]A numeric range written as "100..200" is treated as a keyword that appears everywhere a number in the range appears

professor domain, Google wins in two queries ("UC Berkeley professor" and "economics professors"). This suggests that in certain cases, reformulating to keyword query is a sensible approach, especially if all the fields in the object query are keywords. Even though Google can be used to reformulate some queries, it is not clear how and when this will succeed. Therefore, we need a principled solution as proposed in this paper.

Chapter 7

Related Work

Many recent works propose methods for supporting structured queries on unstructured text [9], [3], [8]. These works follow a typical extract-then-query approach, which has several problems as we discussed in section 1. [1] proposed using several large scale techniques. Their idea of using specialized index and search engine is similar to our work. However those methods assumes that structured data follows some textual patterns whereas our system can flexibly handle structured object using textual patterns as well as web page features.

Interestingly, the approach of translating structured queries to unstructured queries has been studied in [11]. The main difference is that SEMEX relies on carefully hand-tuned heuristics on open-domain SQL queries while we use machine learning to do the translation on domain specific queries.

Machine Learning approaches to rank documents have been studied extensively in IR [12]. Even though much of existing works can be used to rank individual constraints in the structured query. We proposed an effective way to aggregate these ranking scores. Further more, existing learning to rank works assumed a fixed set of features, whereas, the feature set in object search depends on object domain. As we have shown, the effectiveness of the ranking function depends much on the set of features. Thus, an semi-automatic method to learn these was proposed in section 5.

Our interactive learning protocol inherits features from existing works in Active Learning (see [18] for a survey). [5] coined the term "interactive machine learning" and showed that a learner can take advantage of user interaction to quickly acquire necessary training data. [17] proposed another interactive learning protocol that improves upon a relation extraction task by incremetally modifying the feature representation.

Finally, this work is related to document retrieval mechanisms used for question answering tasks [19] where precise retrieval methods are necessary to find documents which contain specific information for answering factoids [2].

Chapter 8

Conclusion

We introduces the *Object Search* framework that searches the web for documents containing real-world objects. We formalized the problem as a learning to rank for IR problem and showed an effective method to solve it. Our approach goes beyond the traditional bag-of-words representation and views each web page as a set of domain independent features. This representation enabled us to rank web pages with respect to *object query*. Our experiments showed that, with small human effort, it is possible to create specialized search engines that outperforms GSEs on domain specific queries. Moreover, it is possible to search the web for documents with deeper meaning, such as those found in *object pages*. Our work is a small step toward semantic search engines by handling deeper semantic queries.

References

[1] E. Agichtein. Scaling Information Extraction to Large Document Collections. *IEEE Data Eng. Bull*, 28:3, 2005.

[2] E. Agichtein, S. Lawrence, and L. Gravano. Learning search engine specific query transformations for question answering. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 169–178, New York, NY, USA, 2001. ACM.

[3] M. Cafarella, C. Re, D. Suciu, and O. Etzioni. Structured Querying of Web Text Data: A Technical Challenge. In *CIDR*, 2007.

[4] Y. Cao, J. Xu, T.-Y. Liu, H. Li, Y. Huang, and H.-W. Hon. Adapting Ranking SVM to Document Retrieval. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 186–193, New York, NY, USA, 2006. ACM.

[5] J. A. Fails and D. R. Olsen, Jr. Interactive machine learning. In *IUI '03: Proceedings of the 8th international conference on Intelligent user interfaces*, pages 39–45, New York, NY, USA, 2003. ACM.

[6] Y. Freund and R. E. Schapire. Large Margin Classification Using the Perceptron Algorithm. *Machine Learning*, 37(3):277–296, 1999.

[7] E. J. Glover, G. W. Flake, S. Lawrence, A. Kruger, D. M. Pennock, W. P. Birmingham, and C. L. Giles. Improving Category Specific Web Search by Learning Query Modifications. *Applications and the Internet, IEEE/IPSJ International Symposium on*, 0:23, 2001.

[8] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien. How to Build a WebFountain: An Architecture for Very Large Scale Text Analytics. *IBM Systems Journal*, 2004.

[9] A. Jain, A. Doan, and L. Gravano. SQL Queries Over Unstructured Text Databases. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1255–1257, 2007.

[10] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *IJCAI*, pages 729–737, 1997.

[11] J. Liu, X. Dong, and A. Halevy. Answering Structured Queries on Unstructured Data. In *WebDB*, 2006.

[12] T.-Y. Liu. Learning to Rank for Information Retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, 2009.

[13] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the Construction of Internet Portals with Machine Learning. *Information Retrieval*, 3(2):127–163, 2000.

[14] J. Platt. Probabilistic outputs for support vector machines and comparison to regularized likelihood methods. In *In Advances in Large Margin Classifiers*. MIT Press, 1999.

[15] N. Rizzolo and D. Roth. Modeling Discriminative Global Inference. In *Proceedings of the First International Conference on Semantic Computing (ICSC)*, pages 597–604, Irvine, California, September 2007. IEEE.

[16] D. Roth. Learning Based Programming. *Innovations in Machine Learning: Theory and Applications*, 2005.

[17] D. Roth and K. Small. Interactive feature space construction using semantic information. In *CoNLL '09: Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 66–74, Morristown, NJ, USA, 2009. Association for Computational Linguistics.

[18] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin-Madison, 2009.

[19] E. M. Voorhees. The trec question answering track. *Nat. Lang. Eng.*, 7(4):361–378, 2001.