Exploiting Web Search Engines to Search Structured Databases

Sanjay Agrawal, Kaushik Chakrabarti, Surajit Chaudhuri, Venkatesh Ganti
Arnd Christian König, Dong Xin
Microsoft Research
One Microsoft Way
Redmond, WA 98052
{sagrawal, kaushik, surajitc, vganti, chrisko, dongxin}@microsoft.com

ABSTRACT

Web search engines often federate many user queries to relevant structured databases. For example, a product related query might be federated to a product database containing their descriptions and specifications. The relevant structured data items are then returned to the user along with web search results. However, each structured database is searched in isolation. Hence, the search often produces empty or incomplete results as the database may not contain the required information to answer the query.

In this paper, we propose a novel integrated search architecture. We establish and exploit the relationships between web search results and the items in structured databases to identify the relevant structured data items for a much wider range of queries. Our architecture leverages existing search engine components to implement this functionality at very low overhead. We demonstrate the quality and efficiency of our techniques through an extensive experimental study.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval; H.3.1 [**Information Systems**]: Content Analysis and Indexing

General Terms

Design, Performance, Experimentation

Keywords

Structured Database Search, Entity Search, Entity Extraction, Entity Ranking

1. INTRODUCTION

Many user queries issued against web search engines do not look for web pages per se, but instead are seeking information from structured databases. For example, the intent of a query such as [canon XTI] is to get information about the specific product. This query can be better served by information from a product database. In order to provide more informative results for these scenarios, the web query is federated to one or more structured databases. Each structured database is searched individually and the relevant structured data items are returned to the web search engine. The search engine gathers the structured search results and displays them along side the web search results. The need for structured data search is illustrated by the proliferation of vertical search engines for products [3, 1], celebrities [2], etc. Typically, these structured databases contain information about named entities like products, people, movies and locations.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

Currently, the search in each structured database is *"silo-ed"* in that it exclusively uses the information in the specific structured database to find matching entities. That is, it matches the query terms only against the information in its own database. The results from the structured database search are therefore independent of the results from web search. We refer to this type of structured data search as *silo-ed search*. This approach works well for some queries. Consider a product database containing the attributes name, description, and technical specifications for each product. A search for a specific product such as [canon XTI] on this database might find several good matches since the name and description attributes contain the requisite information to answer this query. However, there is a broad class of queries where this approach would return incomplete or even empty results. For example, consider the query [light-weight gaming laptop]. Dell XPS M1330 may be a light-weight laptop that is suitable for gaming but the query keywords {light-weight, gaming} may not occur in its name, description or technical specifications. Silo-ed search over the above product database would fail to return this relevant product. One or more reviews of the product may describe it using those terms and can help deduce that the product is relevant to the query. However, the structured database may not contain the comprehensive set of reviews of each product necessary to identify the relevant products. Hence, a silo-ed search against the structured database may miss relevant results.

In contrast, several web documents from product review sites, blogs and discussion forums may mention the relevant products in the context of the query keywords {light-weight, gaming, laptop}. Therefore, the documents returned by a web search engine are likely to mention products that are relevant to the user query. However, each of the returned web documents may contain many relevant products and even some irrelevant ones. To assemble the relevant products mentioned in documents returned by a web search engine, a user has to read through a potentially large number

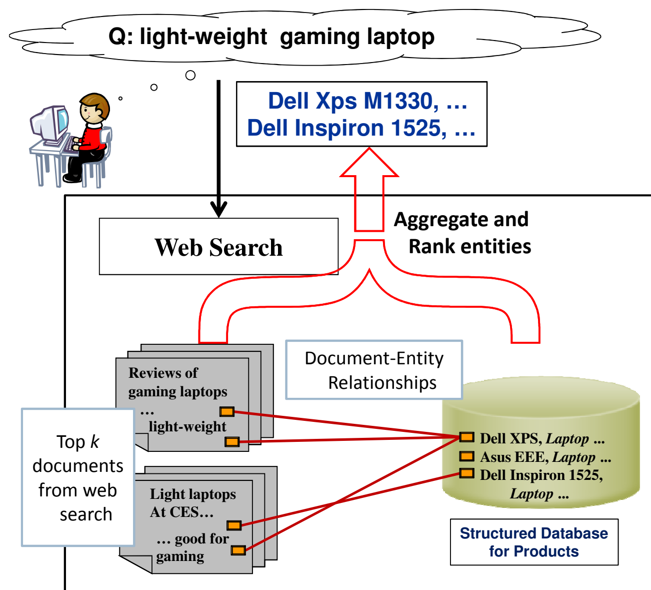

Figure 1: Overview over Structured Data Search

of documents, usually beyond the top 2 or 3 documents. Assuming that we can identify the mentions of the products in web documents, we can automatically identify the most relevant products amongst those mentioned in the top N web search results based on the following insights. First, how and where a product is mentioned in the individual returned documents provides evidence about its relevance to the query. For example, a product mentioned closer to the query keywords in the returned document is likely to be more relevant than that mentioned farther away from the query keywords. Second, how often a product is mentioned across the top web search results also provides important hint about its relevance to the query. A product mentioned often across many returned web documents is likely to be more relevant than another product which is mentioned only a few times. Hence, we can identify the most relevant products by first assessing the evidence each individual returned document provides about the products mentioned in it and then *"aggregating"* those evidences across the returned documents.

Our Design: In this paper, we study the task of efficiently and accurately identifying relevant information in a structured database for a web search query. Recall that we consider structured databases that contain information about named entities. Therefore, we also refer to search for relevant information in a structured database as entity search. Our approach, illustrated in Figure 1, is to first establish the *relationships* between web documents and the entities in structured databases. Subsequently, we leverage the top web search results and the relationships between those returned documents and the entities to identify the most relevant entities. Our approach can return entity results for a much wider range of queries compared to silo-ed search.

While implementing the above high level approach, our main design goals are: (i) be effective for a wide variety of structured data domains, (ii) be integrated with a search engine and to exploit it effectively. Our techniques use a combination of pre-processing and web search engine adaptations in order to implement the entity search functionality at very low (almost negligible) space and time overheads. We now briefly discuss the intuition behind our techniques for achieving these design criteria.

Variety of Structured Data Domains: We desire our technology to be effective for a wide variety of structured data domains, e.g., products, people, books, movies or locations. One of the main tasks is to establish the relationships between the web documents and entities in these structured databases. In this paper, we focus on the *"mentions relationship"*, that is, a web document is related to an entity if the entity occurs the document. Our goal is to be able to establish this relationship, i.e., identify mentions of entities in web documents for a wide class of entities. This task is an instance of the *entity extraction* problem [15, 18]. However, current entity extraction techniques cannot be adopted in our scenario. Current entity extraction techniques use machine learning and natural language techniques to parse documents and break it into sentences, and assign parts of speech tags for extracting entities. These techniques can be quite resource-intensive. Even if entity extraction is performed at document indexing time in a web search engine, the additional overhead is typically unacceptable as it adversely affects the document crawl rates.

Our main insight for overcoming this limitation is that the structured database defines the universe of entities we need to extract from web documents. For any entity not in the structured database, the search engine cannot provide the rich experience to a user as the entity is not associated with any additional information. Such entities would not be returned as results. We therefore constrain the set of entities that need to be identified in a web document to be from a given structured database. By doing so, we can avoid the additional effort and time spent by current entity extraction techniques to extract entities not in the target structured database. We leverage this constraint to develop techniques (i) which can handle a *wide variety of structured data domains*, and (ii) which are also significantly more *efficient* than traditional entity extraction techniques.

Integration with Search Engine Architecture: The challenge here is to keep the space and time overheads, both at document indexing and query processing times, very low. In order to address these challenges, we leverage and adapt existing search engine components. We discuss the details in Section 2.

In contrast to our approach, prior entity search approaches [11, 6] do not effectively exploit the web search engines. These systems rely on sophisticated entity extraction and natural language parsing techniques and custom indexing structures in order to return entities relevant to a user's query. These approaches have significant overhead which makes it difficult to adopt them at web scale. Furthermore, we show in Section 5 that our techniques produce higher quality results compared with those prior approaches. We will discuss these approaches in detail in Section 6.

The remainder of the paper is organized as follows. In Section 2, we describe our approach and the integration with the search engine architecture. In Section 3, we discuss our techniques for entity extraction. In Section 4, we discuss our techniques for aggregating relevance of entities across the returned documents. In Section 5, we describe an ex-

perimental study illustrating quality and efficiency of our system. In Section 6, we discuss related work and conclude in Section 7.

2. ARCHITECTURAL OVERVIEW

In this section, we first describe the key insights of our search engine-integrated approach for entity search. Then we describe the adaptations to the search engine to efficiently implement it.

2.1 Key Insights

Our search engine-integrated approach is based on the following observation. Even though the query keywords may not match with any entity or their descriptions in the entity database, the web documents relevant to the query would typically mention the relevant entities.

Consider the example depicted in Figure 1. Here, we issue the query [light-weight gaming laptop] to the search engine and obtain the set of highly relevant documents. A large fraction of these documents will likely contain mentions of relevant entities: *laptops* that are *light-weight* and suitable for *gaming*. Furthermore, those mentions will typically occur in close proximity to the query keywords. By *aggregating* the occurrences of the entities in the top N (typically, 10) web search results, we can obtain the most relevant entities for the above query.[1]

In order to enable this functionality, our architecture needs to (a) establish the 'mention relationships' between entities in the structured database and the web search results and (b) aggregate the entities related to the top web search results and rank them. Given the response time requirements of search engines, it is not feasible to identify the required relationships at query time. Hence, we must be able to (i) identify the required relationships off-line (at document indexing time) and (ii) materialize and index them so that they can retrieved, aggregated and ranked at query time at a very low overhead. Our key insight here is to leverage suitable data structures and processing components already part of industrial search engines to accomplish the above goals. We will now describe this approach in detail.

2.2 Architecture

Figure 2 illustrates a high level overview of current search engine architecture as described in [8]. It has four main components: (i) The *crawlers* crawl documents on the web and store them in a repository. (ii) The *indexers* process each document from the crawled repository, and build an inverted index and a document index. For each keyword, the inverted index maintains the list of all its occurrences in all documents. The document index, referred to as the DocIndex in short, is indexed by document identifier and contains, for each document, the metadata associated with the document like URL, title, etc. as well as the raw content of the document. (iii) The *searchers* use the inverted index to return the best K document identifiers and then access the DocIndex to obtain the URLs, titles, and generate the

[1]Note that a product mention close to the query keywords does not guarantee that the product is relevant to the query. For example, the document might say "laptop X is a great gaming laptop, but look somewhere else if you are looking for light-weight". However, it is unlikely that multiple top results will contain such references and hence, after aggregation, such entities are unlikely to be ranked on top.

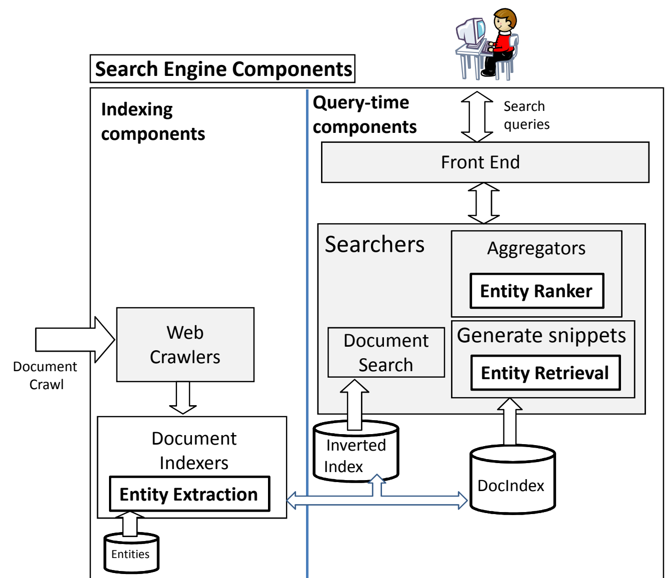

Figure 2: Architecture

(query dependent) snippets that are shown to the user. (iv) The *front end* implements the experience a search user is presented with.

To implement the entity search functionality within this framework, we require three modifications to the components outlined above.

Entity Extraction: The entity extraction component takes a text document (or a document tokenized into a sequence of tokens) as input, analyzes the document and outputs all the mentions of entities from the given structured database mentioned in the document. For each mention, it outputs the entity name, id, and the position in document it occurs at. As shown in Figure 2, we embed this component in the search engine's indexer which already processes and tokenizes documents for indexing.

When a crawled document is being tokenized and indexed by the indexer, we invoke the entity extraction component and obtain the list of entities mentioned in the document. We add the list of entity mentions to the document's metadata. The indexer stores this metadata along with the other metadata of the document (like URL, title) in the DocIndex. We describe this component further in Section 3.

Entity Retrieval: The task of entity retrieval is to lookup the DocIndex for a given document identifier and retrieve the entities extracted from the document (at document indexing time) along with their positions. A direct implementation of this approach would add an additional access to the DocIndex per document in the web search results. Because this cost is incurred at query processing time, the additional overhead could be excessive.

Our observation here is that current search engines access the DocIndex anyway to generate snippets for the top ranking documents. Given a document identifier, the query keywords and their positions in the document (which is all information present in the inverted index), they retrieve the document's metadata and context of the query keywords

within each document. This information is used to generate the snippet for the document. In our architecture, the extracted entities for each document are stored by the indexer in the DocIndex as metadata of the document. Therefore, we are able to piggyback on accesses to the DocIndex and modify the snippet generator to also return the entities extracted from the document along with their positions. Since the snippet generator is always invoked for the top 10 web search results, the entity retrieval component always retrieves the extracted entities for the top 10 documents. If the user requests for subsequent results, the snippet generator is invoked for subsequent documents as well. In that case, the entity retrieval component retrieves the extracted entities for those documents as well. Because we leverage existing structures and access patterns, our approach of retrieving entities adds very little overhead to a search engine's query processing time.

Entity Ranker: The entity ranker ranks the set of all entities returned by the entity retrieval component. As shown in Figure 2, we embed the entity ranker in the search engine's aggregators. The set of highest-ranked entities are then returned along with the web document results.

Observe that one key aspect of our architecture is the tight integration with and the re-use of existing search engine components. This approach allows us to leverage the high relevance of web search results and the scalability of the search engine architecture. But it also poses a challenge: our components must not impose a significant time and space overhead on the search engine. In the following two sections we will focus on the two critical parts of the proposed infrastructure – the entity extraction and the entity ranking – and show how we address this challenge in detail.

3. ENTITY EXTRACTION

The goal of the entity extraction task is to extract entities from the given structured database that are mentioned in each document. We need this component to be very efficient because it is invoked for every document inside the search engine's indexer. Traditionally, entity extraction techniques analyze the entire document: breaking it up into sentences, identifying noun phrases and then filtering them to identify certain classes of entities. Hence, these techniques are usually not very efficient.

Our main insight to address the above limitations exploits the constraint that we have to extract only the entities in the structured database. This constraint enables us to split the extraction into two steps, as shown in Figure 3. In the first *lookup driven extraction* step we efficiently recognize all phrases in a document which match with any entity in the database, using purely syntactic matching of entity strings. However, some of these phrases may not actually be referring to the entity in the structured database. For example, the phrase "Pi" in a document may not always be referring to the entity "Pi" in the movie database. In the second *entity mention classification* step, we analyze (using machine learning classifiers) the contexts in which the entity strings are mentioned in order to identify the mentions that actually refer to the entities in our database. We now describe these two steps.

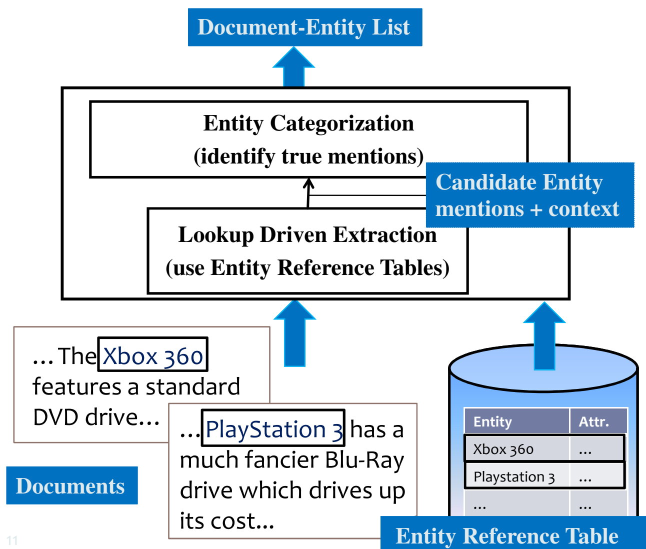

Figure 3: Components of Scalable Entity Extraction

3.1 Lookup-Driven Extraction

We now introduce some preliminary notation and define the lookup driven entity extraction problem. Let $\mathcal{E}$ be the reference set (from the structured database) of entities. We associate each entity e with a class $c = class(e)$ and denote the set of all entity classes as $\mathcal{C}$. Each entity $e \in \mathcal{E}$ is a sequence of words. An input document d is also a sequence of words. For any sequence $s = [w_1, \ldots, w_k]$ of words, we use $s[i, j]$ where $(1 \leq i \leq j \leq k)$ to denote the subsequence $[w_i, \ldots, w_j]$. The triplet $(d, e, start)$ is a *candidate mention* of an entity $e \in \mathcal{E}$ by a document d if the contiguous subsequence $d[start, start + |e|]$ of words in d is equal to the sequence of words in e.

Definition 1. *(Extraction from an input document) An extraction from a document d with respect to the reference set $\mathcal{E}$ is the set $\{m_1, \ldots, m_n\}$ of all candidate mentions in d of the entities in $\mathcal{E}$.*

Example 1. Consider the reference entity table and the documents in Figure 3. The extraction from $d1$ and $d2$ with respect to the reference table is {(d1, "Xbox 360", 2), (d3, "PlayStation 3", 1)}.

The lookup-driven entity extraction problem reduces to the well studied multi-pattern matching problem in the string matching literature [25]. The goal of multi-pattern matching is to find within a text string d all occurrences of patterns from a given set. In our scenario, if each entity is modeled as a pattern, the lookup-driven entity extraction problem reduces to the multi-pattern matching problem.

Popular techniques for solving the multi-pattern matching problem (e.g., Aho-Corasick algorithm) is to build a trie over all patterns. The trie is used to significantly reduce the number of comparisons between subsequences of words in an input document and patterns [19]. We adapt this technique and build a trie over the sequence of tokens in each entity of the reference table. We encode each distinct token by a unique integer thus representing each entity as a sequence of integers. The encoding allows us to reduce the overall memory required for storing the trie. During

the initialization step, we build a trie over all entities in the reference set. While processing a document, if a token sub-sequence starting at position p in a document does not match the prefix of any entity in the reference set, then the prefix match fails in the trie. At that point, we can start matching document sub-sequences starting at $p+1$ (or even later if we maintain additional information in the form of backlinks per internal node in the trie). Since most sub-sequences in documents do not match even short prefixes of any entity, we can skip significant parts of the document and hence can process the document efficiently.

Note that in typical search engines, the crawled documents are partitioned and each partition is potentially indexed by a different machine. In such cases, we need to initialize the trie once per partition or at least once per distinct machine. Since the initialization time (as shown in Section 5) is usually insignificant compared to the time required to process documents in a partition, the additional overhead due to initialization is acceptable.

Approximate Match: In many realistic scenarios, e.g. extraction of product names, the requirement that a sub-string in a document has to match exactly with an entity string in the reference table is very limiting. For example, consider the set of consumers and electronics devices. In many documents, users may just refer to the entity *Canon EOS Digital Rebel XTi SLR Camera* by writing "Canon XTI", or "EOS XTI SLR" and to the entity *Sony Vaio F150 Laptop* by writing "Vaio F150" or "Sony F150". Insisting that sub-strings in these documents match exactly with entity names in the reference table would cause these product mentions to be missed. Therefore, it is very important to consider approximate matches between sub-strings in a document and an entity name in a reference table [16, 10].

We overcome this limitation by first identifying a set of "synonyms" for each entity in the reference table. Each synonym for an entity e is an *identifying* set of tokens, which when mentioned contiguously in a document refer to e [14]. Once such synonyms are identified and the reference set enhanced with them, the task of identifying approximately matching entity mentions reduces to that of identifying exactly matching entity mentions. This allows us to leverage the efficient lookup driven extraction techniques discussed above. On typical reference sets such as consumer and electronics products, the synonym expansion adds on an average up to two synonyms per reference entity.

Alternatively, we can also adapt other approaches for efficiently enabling approximate match [16, 10]. These rely on known similarity functions which measure similarity between a sub-string in a document and the target entity string that it could match with [16, 10].

Memory Requirement: We assume that the trie over all entities in the reference set fits in main memory. In most scenarios, this is a fine assumption. Even if the number of entities (product names) runs up to 10 million, we observed that the memory consumption of the trie is still less than 100 MB. Given the main memory sizes of current commodity hardware, the trie easily fits in main memory. In those rare cases where the size of the trie may be larger than that available, we can consider sophisticated data structures such as compressed full text indices [24]. Alternatively, we can also consider approximate lookup structures such as bloom filters, and ensure that the entity mention classification filters out false positives. We leave the investigation into these extreme scenarios as future work.

3.2 Entity Mention Classification

After all *candidate mentions* within a document have been identified through lookup-driven extraction, we now have to classify these into candidate mentions that correspond to entities within our reference set $\mathcal{E}$ and those that do not. We refer to the former as *true mentions* or simply entity mentions and the latter as *false mentions*. It is important to note that this classification cannot be done based on the string of the candidate mention alone, given that many entity strings are highly ambiguous. For example, the string "Pi" can refer to either the movie or the mathematical constant. Similarly, an occurrence of the string "Pretty Woman" may or may not refer to the film of the same name.

To address this issue, we leverage the fact that $\mathcal{E}$ is a set of entities from known entity classes such as products, actors and movies. Therefore, determining whether a candidate mention refers an entity e in the reference set can be cast as a classification problem: if the candidate mention belongs to the corresponding class $class(e)$, it is a true mention of e, otherwise it is a false mention.

We use linear *Support Vector Machines* (SVM) [27] to classify candidate mentions as they are known to be accurate for this type of classification tasks [23, 22]. We now discuss two important issues for training and applying these classifiers. The first is that of obtaining training data and the second is that of which feature sets we use as part of the classifier.

Generating training data: We require a corpus of labeled training data for true as well as false mentions. Since obtaining these via human annotation is costly, we propose to use *Wikipedia* [4] to automatically generate such examples for many classes of entities. We now describe the process we adopt.

For many classes of entities (such as *movies*, *actors*, *painters*, *writers*), Wikipedia maintains manually curated pages listing the most important instances of each class (e.g., `http://en.wikipedia.org/wiki/-List_of_painters_by_name`). The pages contain both the entity string as well as a link to the Wikipedia page of that entity. We can use the above information to automatically label candidate mentions of entities in one of the above lists in Wikipedia documents. If the candidate mention is linked to the Wikipedia page of the entity, then it is a true mention. If it is linked to a different page within Wikipedia, it is a false mention. Because of the size of the overall Wikipedia corpus, this gives us a significant number of examples of candidate mentions as well as the contexts they occur in to train our classifier on.

Feature sets: We use the following 4 types of features:

Document Level Features: The information about the document where the candidate mention occurs is important for mention classification. For example, movies tend to occur in documents about entertainment, and writers in documents about literature. To capture this information, we identify important phrases (n-grams, for $n = 1, 2, 3$) in a document which encapsulate this information. In our implementation, we trained the classifier using the 30K

n-gram features occurring most frequently in our training set.

Entity Context Features: The context a candidate mention occurs in is important for classification where the context is the sequence of words immediately preceding or following a candidate mention. For example, a mention of a movie is often preceded by the words '*starred in*'. We select the 10K most frequent n-grams within preceding contexts of entity mentions from the training data. Presence of these n-grams in candidate mention contexts are indicative of true entity mentions, so we can increase the accuracy of our classifiers by using these features. We use a second set of 10K n-gram features for contexts directly following the candidate mentions.

Entity Token Frequency Features: The frequency with which we observe a candidate mention in a random text document is an important feature for entity mention classification. For example, the movie title '*The Others*' is a common English phrase, likely to appear in documents without referring to the movie of the same name. Hence, we are likely to see many false mentions for such phrases. As a consequence, we include the frequency of each entity string in a *background corpus* of English text as a feature.

Entity Token Length Features: Short strings and strings with few words tend to be more frequent as generic phrases that do not refer to entities than longer ones. Therefore, we include the number of words in an entity string and its length in characters as features.

Feature Extraction using Lookup-Driven Extraction:

One crucial factor for the features we have selected is that the value of each of these features can be determined using (a variant of) lookup-driven extraction during classification time. In each case, we need to tokenize the input document a candidate mention occurs in and then lookup the tokens and n-grams of tokens in lookup tables containing the individual features, their weights and additional information (such as their frequency in the background corpus). Therefore, the classifier can benefit from the speed of lookup driven extraction. In Section 5, we will demonstrate that this, in combination with the fact that linear SVM-classifiers require us to only sum up the weights associated with each feature for classification, leads to a very low overhead for this component.

4. ENTITY RANKING

The task of the entity ranker is to rank the set of entities occurring in the top N (typically 10) pages returned by search engine. It takes as input the ranked list of top N URLs, the set of entity mentions and the positions they occur in for each URL, the positions of the query keywords for each URL and the number K of entities desired. It computes the "score" of each entity in the input set using a scoring function. Subsequently, it returns the K (or less) entities with scores above a specified threshold, ranked in decreasing order of their scores. Prior approaches have proposed to learn the scoring function using a supervised machine learning model [13]. However, these approaches require large amounts of labeled training data. Since we do not have learning datasets for many entity domains, we initially adopt an unsupervised approach. We note that when sufficient training data is available, existing techniques for learning ranking functions can be leveraged. However, because we are exploiting highly relevant documents returned by a search engine, we observe that even our unsupervised scoring function produces high quality results (as shown in Section 5). Our unsupervised scoring function is based on 3 main observations.

- **Aggregation**: The number of times an entity is mentioned in the top N documents is crucial in determining how relevant the entity is. The higher the number of mentions, the more relevant the entity.
- **Proximity**: The proximity of the mentions of an entity to the query keywords is also very important for determining the relevance of the entity. An entity mentioned near the query keywords is likely to be more relevant to the query than an entity mentioned farther away from the query keywords.
- **Document importance**: Among the top N documents, some documents are more important than others in determining the relevance of an entity. An entity mentioned in a document with higher relevance to the query and/or higher static rank is likely to be more relevant than one mentioned in a less relevant and/or lower static rank document.

Let D be the set of top N documents returned by the search engine. Let D_e be the set of documents mentioning an entity e. The score of an entity e is computed as follows.

$$score(e) = \sum_{d \in D_e} imp(d) * score(d, e)$$

where $imp(d)$ is the importance of a document d, and $score(d, e)$ is e's score from the document d. Note that if an entity is not mentioned in any document, its score is 0 by this definition. We now define the two components of the score:

Importance of a Document $imp(d)$: Since the search engine takes both relevance to the query and static rank into account in order to compute the search results, we rely on the search engine to obtain the importance of a document. We use the document's rank among the web search results to compute the importance of the document. [2]

We partition the list of top N documents into ranges of equal size B, where $1 \leq B \leq N$. For each document d, we compute the range $range(d)$ the document d is in. $range(d)$ is $\lceil \frac{rank(d)}{B} \rceil$ where $rank(d)$ denotes the document's rank among the web search results. Following the normalized discount cumulative gain (NDCG) function popularly used for assigning importance to a ranked list of web documents, we define the importance $imp(d)$ of d to be $\frac{log(2)}{log(1+range(d))}$ [21].

Score from a Document $score(d, e)$: The score $score(d, e)$ of an entity e gets from a single document d depends on (i) whether e is mentioned in the document d, and (ii) on the proximity between the query keywords and the mentions of e in the document d.

Both the query keywords and the mentions of an entity may occur in several positions in a document. Aggregating

[2] We can alternatively use the search engine's relevance score to compute the importance of the document. However, since the search engine relevance scores are designed for comparison among themselves and not meant for aggregation, they need to be suitably transformed for our use. Finding the appropriate transformation function is hard.

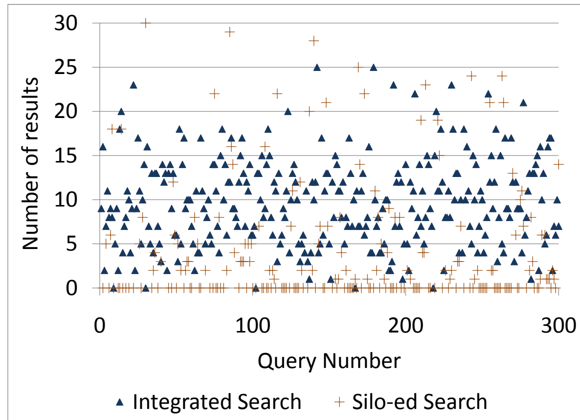

Figure 4: Comparison of proposed approach with silo-ed search

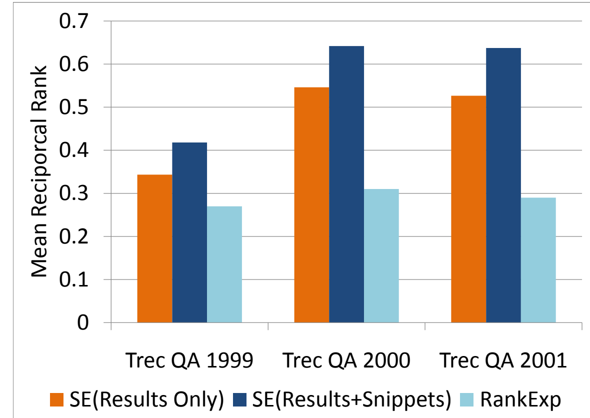

Figure 5: Quality of entity ranking on Trec Question Answering benchmark

the proximities among the individual keyword and entity occurrences cleanly is hard. However, search engines already employ sophisticated snippet generation techniques. These techniques linguistically analyze the contexts of the query keyword occurrences in the document and piece together the contexts to generate the snippet. Therefore, we observe an entity appearing in the snippet is a good indicator that it is "close" to the query keywords in the document.

Note that while detecting the mention of an entity in the snippet, we may not require the entire string (or its synonym) to occur in the snippet. We can detect partial mentions as long as it unambiguously refers to a single entity mentioned in the document. That is, the partial match is contained in only one entity mentioned in the document. For example, consider the entity 'Michael Jordan' mentioned in a document. If the document's snippet contains only 'Jordan', we detect it as a mention of 'Michael Jordan' if he is the only entity mentioned in the document that contains the token 'Jordan'. Note that we do not require 'Jordan' to unambiguously refer to 'Michael Jordan' across all documents.

We define the $score(d, e)$ as follows.

$$\begin{aligned} score(d, e) &= 0, \text{ if e is not mentioned in d} \\ &= w_a, \text{ if e occurs in snippet} \\ &= w_b, \text{ otherwise} \end{aligned}$$

where w_a and w_b are two constants. The value w_a is the score an entity gets from a document when it occurs in the snippet, and w_b is the score when it does not. In our implementation, we set $B = 5$, $w_a = 1.0$, and $w_b = 0.3$.

5. EXPERIMENTAL EVALUATION

We now present the results of an extensive empirical study to evaluate the approach proposed in this paper. The major findings of our study can be summarized as follows.

• **Superiority over silo-ed search**: Our integrated entity search approach overcomes the limitation of silo-ed search over entity databases. It returns relevant results when silo-ed search fails to return any relevant results.

• **Effectiveness of entity ranking**: Our entity ranking based on aggregation and query term proximity produces high quality results.

• **Low overhead**: Our architecture imposes much lower overhead on the query time compared to other architectures proposed for entity search. Furthermore, our extraction techniques imposes low overhead on the search engine indexer component.

• **Accuracy of entity extraction**: Our entity classifier is accurate in identifying the true entity mentions in web pages.

All experiments reported in this section were conducted on an Intel 6600 PC (2.4 GHz) with a single core and 4GB RAM, running Windows 2003 Server.

5.1 Comparison with Silo-ed Search

We compared our integrated entity search approach with silo-ed search on a product database containing names and descriptions of 187,606 Consumer and Electronics products. We implemented silo-ed search on the above database by storing the product information in a table in Microsoft SQL Server 2008 (one row per product), concatenating the text attributes, viz. product name and description, into a single attribute and building a full text index on that concatenated column using SQL Server Integrated Full Text Search engine (iFTS). Subsequently, given a keyword query, we issue the corresponding AND query on that column using iFTS and return the results. For example, for the keyword query [lightweight gaming laptop], we issue the iFTS query ('lightweight' AND 'gaming' AND 'laptop') on the concatenated column. We issue AND queries (instead of OR queries or freetext queries) since users typically expect conjunctive semantics in web search, i.e., a relevant entity is expected to contain all (or most) of the keywords in the name and description fields.

We implemented the integrated entity search approach for the above database (consisting of the product names and their synonyms) using the Live Search infrastructure. In all our experiments, we use the top 10 documents returned

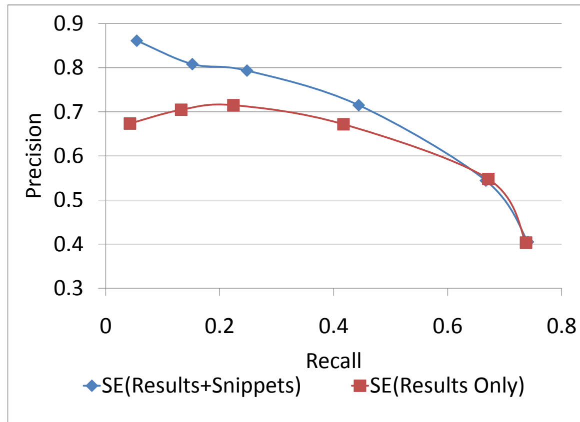

Figure 6: Quality of entity ranking on IMDB dataset

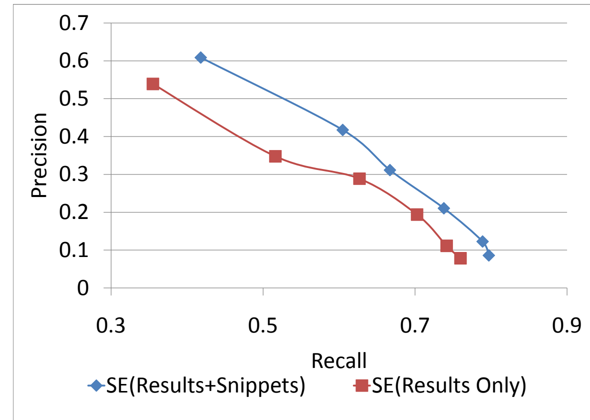

Figure 7: Quality of entity ranking on Wikipedia dataset

by Live Search engine to aggregate and rank the entities. This is because the snippet generator is always invoked for these documents. It might be possible to aggregate over more documents if the user requests for subsequent results. Aggregating over more documents (i.e., 20 or 30) typically improves the quality of entity results. We restrict our evaluation to top 10 documents in this paper.

We took a random sample of 316 Consumer and Electronics queries [3] from the Live search query log. Figure 4 shows the number of results returned by the two approaches for the 316 queries. Integrated entity search returned answers for most queries (for 98.4% of the queries) whereas silo-ed search often returned empty results (for about 36.4% of the queries). This is because the name and description of a product often does not have enough information to answer a query but web pages mentioning the product name do.

5.2 Entity Ranking

We evaluate the quality of our entity ranking on 3 datasets: TREC Question Answering benchmark, Wikipedia data and IMDB data. We focus on queries with person entities as answers. We used an entity reference table containing *2.04 million* person names (and their synonyms [4]) obtained from IMDB, Wikipedia, DBLP, ACM as well as several internal databases. Note that our approach is designed to return only the entities in the structured database. We therefore ensure that, for all the 3 datasets, the relevant entities (along with their synonyms) for all the queries are present in the entity reference table.

TREC Question Answering benchmark: We used the factoid questions from the TREC Question Answering track for years 1999, 2000 and 2001. We focussed on the 160 queries with person entities as answers. We evaluated the quality of our entity ranking using the "mean reciprocal rank" (MRR) used in the TREC community: if the first answer to query q is at rank r_q, award a score of $\frac{1}{r_q}$ and average over the query set Q to get MRR $\frac{1}{Q}\Sigma_{q\in Q}\frac{1}{r_q}$. Figure 5 shows our entity ranking has high quality ($MRR = 0.6$), i.e., the actual answer appeared among the top 1-2 results. Recall that our entity ranking depends on two main insights, viz., aggregation and proximity, we evaluate the individual impact of the two insights on quality by measuring the quality of entity ranking without using snippets (i.e., taking aggregation but not proximity into account) and with snippets (i.e, taking both aggregation and proximity into account). Figure 5 shows that using snippets improved the MRR significantly (e.g., from 0.53 to 0.64 for the TREC 2001 queries). We also compare our technique with the *RankExp* entity ranking technique that does not leverage the search engine results [13]. Since our technique is based on the web search engine and RankExp is difficult to implement on the web corpus, we compare with the results of RankExp for the same queries (on the ACQUAINT corpus) as reported in [13]. Our entity ranking significantly outperforms the MRR of RankExp as shown in Figure 5.

IMDB Data: We obtained the movie names and the people (actors, director, producer, etc.) associated with them from the MSN Movie database. For example, for the movie "The Godfather", the associated people include Francis Ford Coppola, Mario Puzo, Marlon Brando, etc. We treat the movie name as the query and the associated people as ground truth for the query. We used 250 such queries along with the ground truth. The ground truth contained, on average, 16.3 answers per query. Figure 6 shows the precision-recall tradeoff for our integrated entity search technique. We vary the precision-recall tradeoff by varying the number k of top-ranked answers returned by the integrated entity search approach (K = 1, 3, 5, 10, 20 and 30). The integrated entity search technique produces high quality answers: the precision is 86%, 81% and 79% for the top 1, 3 and 5 answers respectively. Using the snippet (i.e., taking proximity into account) improves the quality significantly: the precision improved from 67% to 86%, from 70% to 81% and from 71% to 79% for $K = 1, 3$ and 5 respectively. Furthermore, the

[3]These are the queries classified as Consumer and Electronics queries by the Live Search query classifier.

[4]We generated the synonyms of person names by using common first name synonyms ('Michael' ≡ 'Mike') and dropping/abbreviating middle names and initials.

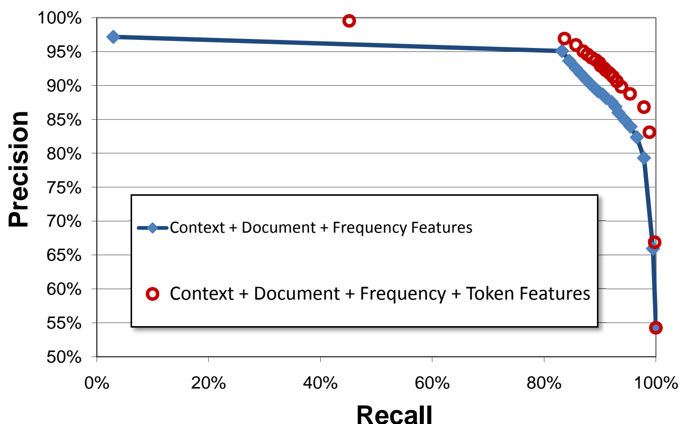

Figure 8: Precision/Recall tradeoff for Entity Classification

MRR improves from 0.8 to 0.91.

Wikipedia Data: We extracted 115 queries and their ground truth from Wikipedia. We focussed on queries whose ground truth can be extracted easily from Wikipedia pages using automated scripts. For example, we extracted ceo queries (e.g., [general electric ceo]) whose ground truth (the corresponding ceo names) can be extracted from the list page `http://en.wikipedia.org/wiki/List_of_chief_executive_officers`. We extracted nobel prize winner queries (e.g., [nobel prize 2006]) whose ground truth (the names of the winners) can be easily extracted from the year pages (e.g., `http://en.wikipedia.org/wiki/2006`). The ground truth contained, on average, 3.6 answers per query. Figure 7 shows the precision-recall tradeoff for our integrated entity search technique for various values of K (K = 1, 3, 5, 10, 20 and 30). The integrated entity search approach produces high quality results for this collection as well: the precision is 61% and 42% for the top 1 and 3 answers respectively. Again, using the snippets significantly improves quality: the precision improved from 54% to 61% and from 35% to 42% for $k = 1$ and 3 respectively. Furthermore, the MRR improves from 0.67 to 0.74.

5.3 Overhead of our Approach

Query Time Overhead: Since we piggyback on snippet generator for entity retrieval, the only overhead of our approach at query time is that of aggregating the retrieved entities from top 10 documents and ranking them. We measured the time taken for aggregating and ranking the entities for two different workloads of 1000 queries each for various entity databases like movie, product (Consumer and Electronics), and people databases. In the first workload we used 1000 head queries from the search query logs. In the second workload, we used 1000 commerce search queries. Table 1 shows the query time overhead of our approach per query for the two workloads. In all cases, the overhead is less than 0.1 milliseconds which is negligible compared to end-to-end query times of web search engines (several tens of milliseconds). We observe that even for the commerce queries on the product database where the aggregation and ranking costs are relatively high (since most of the documents returned for these queries contain large number of product mentions), the overhead remains below 0.1 ms. In summary, our architecture imposes negligible query time overhead on existing search engines.

Structured database	Database size	Average overhead per query for head query workload	Average overhead per query for commerce query workload
Movies	800K	0.022 ms	0.017 ms
Products	200K	0.019 ms	0.070 ms
People	2 million	0.029 ms	0.026 ms

Table 1: Query time overhead of our approach

Extraction Overhead: Since the extraction takes place in the indexer component of the search engine, it is important for the extraction overhead to be low so that the search engine crawl rate is not affected. The extraction time consists of 3 components: the time to scan and decompress the crawled documents (performed in chunks of documents), the time to identify all candidate mentions using lookup-driven extraction and the time to perform entity mention classification using the classifier described in Section 3.2. We measured the above times for a web sample of 30310 documents (250MB in document text). The times for each of the individual components are 18306 ms, 51495 ms and 20723 ms respectively. The total extraction time is the sum of the above three times: 90524 ms. Hence, the average extraction overhead at less than 3ms per web document, which is acceptable in our architecture.

5.4 Accuracy of Entity Classification

We now evaluate the accuracy of the classifier described in Section 3.2 to identify true mentions from the candidate mentions identified by the lookup driven extractor. For this purpose, we use separate training and test corpora of a total 100K instances of movie mentions found within Wikipedia documents and generated automatically via the link-analysis method described in Section 3.2. We use a linear *Support Vector Machine* (SVM) [27] trained using *Sequential Minimal Optimization* [26] as the classifier. This means that once all features for a given candidate mention have been obtained, the computation of the corresponding classifier output corresponds to a summation of the corresponding feature weights, which can be performed efficiently.

We measured the accuracy of the classifier with regards to differentiating true mentions from the false ones among the set of all candidates. 54.2% of all entity candidates in the test data are true mentions. Our classifier achieves an accuracy of 97.2%. Furthermore, by varying the threshold (the distance from the separating hyperplane in case of this classifier), we can tradeoff additional precision against lower recall for the extracted true mentions. Figure 8 shows the resulting precision/recall tradeoff.

6. RELATED WORK

The problem of entity search has been studied recently [13, 12, 6]. In [13], the authors propose a specialized indexing scheme that maintains an inverted index called the "atype index" in addition to the standard word index. The aytpe index maintains, for each entity type, the list of document IDs that contains occurrences of one or more entities of that type

along with the positions of those occurrences. Given a keyword query and the desired entity type, the system performs a mutli-way intersection between the keyword posting lists and the atype posting lists, computes the score of the entities in the atype posting list and return the K entities with the highest scores. This architecture results in a bloating of the inverted index (by a factor of 5). To reduce the bloating, the authors propose to maintain the posting lists for only a small number of entity types. At query time, they find the best generalization of the desired entity type for which the posting list exists. The query processor performs the above intersection with the posting list of the generalized entity type and then uses the "reachability index" to remove the false positives. This technique increases the query time by a factor of 1.9 due to additional intersections and the lookups in the reachability index involved. This overhead makes it difficult to integrate this architecture in commercial search engines.

In [6], the authors present another entity search technique that creates a concordance document for each entity, consisting of all the sentences in the corpus containing that entity. They then index and search these documents using a standard IR engine like Lucene. The result of the search returns a ranked list of entities. The advantage of this approach is that it can leverage an existing search engine without major modifications. However, this approach may produce large concordance documents and hence a big index, resulting in high search times. Furthermore, the above querying over the entity index has to be performed in addition to the traditional querying over document indexes. Hence, this architectures can add significant overhead to web search engines in terms of query processing time and is hence difficult to adopt at web scale. Note that neither of the above two approaches exploit the results produced by the search engine.

Systems like KnowItAll and TextRunner [17, 9] focus on extracting entities and relationships from web pages and populating structured relations. In [9], the authors propose to index the tuples in the structured relations using an inverted index and support keyword search on them. Like the entity search approaches discussed above, this approach does not leverage the search engine.

Several techniques have been proposed for keyword search on databases (e.g., [5, 20, 7]). All these techniques represent implementations of silo-ed search. As mentioned before, siloed search often results in empty or incomplete results due to the limited amount of information in the database. The search engine integrated approach proposed in this paper overcomes the above limitation.

7. CONCLUSION

In this paper, we showed that establishing and exploiting relationships between web search results and structured entity databases significantly enhances the effectiveness of search on these structured databases. We proposed an architecture which effectively leverages existing search engine components in order to efficiently implement the integrated entity search functionality. Specifically, our techniques add very little space and time overheads to current search engines while returning high quality results from the given structured databases.

8. REFERENCES

[1] http://search.live.com/products/.

[2] http://search.live.com/xrank?form=xrank1.

[3] http://www.google.com/products.

[4] www.wikipedia.org.

[5] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *IEEE ICDE Conference*, 2002.

[6] M. Bautin and S. Skiena. Concordance-Based Entity-Oriented Search. In *Web Intelligence*, pages 586–592, 2007.

[7] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE Conf.*, 2002.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7), 1998.

[9] M. Cafarella, M. Banko, and O. Etzioni. Relational Web Search. In *WWW Conference*, 2006.

[10] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *ACM SIGMOD Conference*, 2008.

[11] S. Chakrabarti. Breaking Through the Syntax Barrier: Searching with Entities and Relations. In *PKDD Conference*, pages 9–16, 2004.

[12] S. Chakrabarti. Dynamic Personalized Pagerank in Entity-Relation Graphs. In *WWW Conference*, pages 571–580, 2007.

[13] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing Scoring Functions and Indexes for Proximity Search in Type-annotated Corpora. In *WWW Conference*, 2006.

[14] S. Chaudhuri, V. Ganti, and D. Xin. Exploiting web search to generate synonyms for entities. In *WWW Conference*, 2009.

[15] W. Cohen and A. McCallum. Information Extraction and Integration: an Overview. In *SIGKDD Conference*, 2004.

[16] W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-markov extraction processes and data integration methods. In *ACM SIGKDD Conference*, 2004.

[17] O. et. al. Web-scale information extraction in knowitall. In *WWW Conference*, 2004.

[18] R. Grishman. Information Extraction: Techniques and Challenges. In *SCIE*, 1997.

[19] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[20] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *28th VLDB Conf.*, 2002.

[21] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4), 2002.

[22] T. Joachims. Text Categorization with Support Vector Machines: Learning with many Relevant Features. In *EMNLP Conference*, 1998.

[23] T. Joachims. Training Linear SVMs in Linear Time. In *ACM SIGKDD Conference*, pages 217 – 226, 2006.

[24] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[25] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambride University Press, 2002.

[26] J. Platt. Fast Training of SVM's Using Sequential Minimal Optimization. In *Advances in Kernel Methods: Support Vector Machine Learning*, pages 185–209. MIT Press, 1999.

[27] V. Vapnik. *Statistical Learning Theory*. Whiley, 2000.