Using a Stack Decoder for Structured Search

Kien Tjin-Kam-Jet, Dolf Trieschnigg, and Djoerd Hiemstra

University of Twente, Enschede, The Netherlands
{tjinkamj,trieschn,hiemstra}@ewi.utwente.nl

Abstract. We describe a novel and flexible method that translates free-text queries to structured queries for filling out web forms. This can benefit searching in web databases which only allow access to their information through complex web forms. We introduce boosting and discounting heuristics, and use the constraints imposed by a web form to find a solution both efficiently and effectively. Our method is more efficient and shows improved performance over a baseline system.

1 Introduction

Many web pages contain structured information that cannot be indexed by general web search engines like Bing or Google [4]. Web search engines use crawlers to follow hyperlinks and download web pages in order to index these pages, which enables fast keyword search. This crawler architecture has three drawbacks [1, 15]. First, a large part of the web cannot be crawled by simply following hyperlinks. Many pages are hidden behind web forms which cannot be automatically filled out by a crawler. Second, the indices of crawler-based search engines are only a snapshot of the state of the web. Pages containing real-time or highly dynamic information like traffic information or stock information are outdated as soon as they are indexed. Third, most of this information resides in structured databases that allow structured queries, a powerful means of searching. In contrast, putting this information in indices of crawler-based search engines would only allow unstructured keyword queries, a less powerful means of searching.

In this work we alleviate these problems by providing a single free-text search box to search multiple websites through complex web forms. We address the problem of translating a free-text query into a structured query, i.e., key-value pairs accepted by web forms. For instance, the free-text query "acer travelmate at least 4gb" could be mapped to the fields 'brand', 'model' and 'minimum memory' of a shopping website. As results, our system would return forms containing such fields, filled out and ready to be submitted. Note that in order to return results, the system does not need to crawl the web pages behind the forms, it just needs to know how to fill out the form given the free-text query. The problem can be decomposed into a *segmentation* problem of cutting up the free-text query into parts (segments); and a *labeling* problem of actually assigning each segment to the right input field. Our work extends existing segmentation & labeling methods based on HMMs (Hidden Markov Models) [16]. Segmenting is based on whitespace and punctuation characters, and subsequent labeling is based

on a probabilistic model. Our contributions are as follows. We propose a novel method that incorporates constraint information (see Sect. 3) and segments, labels, and normalizes queries; thereby deriving structured queries. We show that it is beneficial to apply boosting and discounting heuristics; that our method can be applied to a multi-domain, multi-site per domain setting; and, that our method outperforms a well known baseline. *Paper outline:* In Sect. 2, we discuss and compare related work to this work. We then formalize the problem and describe our framework in Sects. 3 and 4. We describe our data in Sect. 5, and our evaluations in Sect. 6. Finally, we round up with our conclusions in Sect. 7.

2 Related Work

Correct **query segmentation in web IR** can substantially improve retrieval results, e.g., grouping 'new' and 'york' as 'new york' can make a big difference. Li et al. [14] argue that supervised methods require expensive labeled data and propose an unsupervised segmentation model that can be trained on click log data. Hagen et al. [7] show that their segmentation algorithm, which uses only raw web n-gram frequencies and Wikipedia titles, is faster than state-of-the-art techniques while having comparable segmentation accuracy. Lastly, Yu and Shi [19] train a CRF (Conditional Random Field [12]) with tokens from a database. They first predict labels for each word in the query, and then segment at each start (S-) label. For example, given the query *Green Mile Tom Hanks* and the predicted labels {[S-MOVIE],[R-MOVIE],[S-ACTOR],[R-ACTOR]}, it is segmented as "Green Mile" and "Tom Hanks".

Query segmentation & labeling. The previous example illustrates that CRFs can both indicate segment offsets (e.g., with start/rest labels) and assign entire segments to fields (e.g., ACTOR or MOVIE). However, CRFs need a lot of expensive (manually labeled) training data. To avoid the high costs of manually labeled data, Li et al. [13] used two data sources to train CRFs: a pool of 19K queries labeled by human annotators; and a pool of 70K queries, automatically generated by matching entries from click logs with information from a product listings database. However, the generated queries did not contain all possible labels. Still, the highest performance was obtained when combining the evidence of both sources. In contrast, Kiseleva et al. [10] train multiple CRFs solely on click log data. But unlike manually labeled data, click log data suffers from noise and sparsity. In a follow-up study [11], they did use some manual data (brand synonyms and abbreviations) and artificially expanded their training set aiming to reduce data sparsity. Sarkas et al. [17] propose an unsupervised approach to segment & label web queries. They train an open language model (LM) on tokens derived from a general web log, and attribute LMs on tokens from the structured data residing in tables. They score results using a generative model of the probability of choosing: a set of attributes $T.\mathcal{A}$ from table T, a set of tokens $\mathcal{AT}$ given $T.\mathcal{A}$, and a set of free tokens $\mathcal{FT}$ given the table T. Further, they decide whether a query is intended as a web keyword query, or as a structured search query. DATAMOLD, by Borkar et al. [3], uses nested HMMs to segment &

label short unformatted text into structured records. They modify the Viterbi algorithm [6] to include semantic constraints, restricting it from exploring invalid paths. Since this violates the independence assumption, they re-evaluate a path when some state transition is disallowed by the constraints. Zhang and Clark [20] describe a framework that uses the averaged perceptron algorithm for training and a beam search algorithm (which is essentially, a stack decoder with a small stack) for decoding, and apply it to various syntactic processing tasks, like joint segmentation and POS-tagging. Our approach differs from these approaches in that it uses a stack decoder [2] and incorporates constraint information to prune, boost and discount; it is not purely probabilistic and works without training; and while it does not require, it can benefit from training.

Conclusion. Probabilistic methods like HMMs or CRFs outperform other methods for segmentation & labeling, but require large amounts of expensive training data, while fully unsupervised methods suffer from noisy training data. As a general remark, there is no agreed upon test collection to compare these methods, which makes it hard to determine the best method. That is, if such a conclusion can be made at all, since each method was developed for very specific use cases.

3 Problem Description and Approach

Our query translation problem can be formalized as:

Given a web form and a free-text query, find the intended values and assign the values to their intended fields, under the constraints imposed by the web form.

A *web form* has input *fields*, it only accepts queries as *structured information needs* consisting of a set of field-value assignments, e.g., $F_i = v_i$, for $i = 1 \dots n$; and, a *free-text query* is an unstructured sequence of characters describing an intended structured information need. Next, we describe the types of contraints, how they can aid free-text to structured query translation, and our approach.

3.1 Hard Constraints

Web forms impose *constraints* that only allow certain combinations of fields and values. Queries satisfying these constraints are *valid*. Otherwise, they are *invalid*.

Mandatory fields. A web form may require certain fields to be filled out before it can be submitted. For example, it may require either the `make` field, or both the `min` and `max` price fields to be filled out before it can be submitted. Formally, *mandatory field constraints* are propositions of the form: $(F_i) \vee (F_j \wedge F_k) \vee \dots$, stating that at least one set of fields must be present in the query.

Conditional fields. While a field may not be mandatory, it may be required if some other field is used. For example, consider a query that contains the text *5 miles near*, which states a radius (near some place). A web form with fields `radius` and `place`, may require that if you fill out `radius`, you must also fill out `place`. Formally, *assertive conditional constraints* are implications of the form: $F_i \rightarrow F_j$, stating that if some field F_i is present, then so must F_j. *Negative*

conditional constraints are implications of the form: $F_i \rightarrow \neg F_j$, stating that if some field F_i is present, then F_j may and must not also be present.
Field frequency. We refer to fields that allow only one value as *single-valued* fields and to fields that allow more values as *multi-valued* fields. *Frequency* constraints state that if a field is single-valued, it can be used at most once.
Categories. A category defines a set of values. For example, the category `base color` defines 'red', 'green' and 'blue' as values, while `year` could define numbers between 1970 and 2015 as values. Closed categories have a limited set of values, which are typically stored in a dictionary. Open categories have a limitless set of values, such as the set of real numbers. These are typically modeled by regular expressions. An input field will only accept values of one specific category.
Dependencies. The values allowed for one field may depend on the value of another. For example, if a `make` field has value *Ford*, then `model` may have *Fiesta*, but not *Laguna*. Formally, *dependency constraints* are implications of the form: $F_i \wedge F_j \rightarrow f(\lambda(F_i), \lambda(F_j))$, stating that if two dependent fields F_i and F_j are used, then the function f applied on their values $\lambda(F_i)$ and $\lambda(F_j)$ must be true. Here, f can be any function that takes two values as input and returns a boolean.

3.2 Soft Constraints

Soft constraints indicate which filled out form is more likely, given a valid query.
Patterns. A pattern determines when to assign values to a particular field by detecting field-specific hints that appear just before or after the values of a field's expected category. Formally, a pattern is defined as a 4-tuple {field name, prefixes, category, postfixes}. Prefixes and postfixes denote a set of words which may be empty. For example, consider the query *to New York from Dallas* and assume that *New York* and *Dallas* are values of the category `city`, which can be assigned to the fields: departure or destination. Then, a pattern for the destination field could for example be: {destination, [to], `city`, [ ]}.
Field order. Ideally, when a query contains a hint for some field F, followed by a value v of the category expected by F, then by all means, assign v to F. In practice however, queries may just contain values, like the query *New York Dallas*. The system would benefit from knowing that a particular field order is more likely than another, e.g., that P(departure, destination) $\geq$ P(destination, departure). We make the Markov assumption and model the probability of a sequence of fields as: $P(F_1, F_2, \ldots, F_n) = \prod_{i=1}^{n} P(F_i|F_{i-2}, F_{i-1})$.

3.3 Approach

Our approach consists of three steps: *a*) **segmenting**, i.e., splitting the free-text query into smaller *segments* ready to be assigned to some field—a segment is a subsequence of the characters of the free-text query. At each character position in the query, we search for known values which are defined by a regular expression or are contained in a dictionary. Our dictionary is based on a Bursttrie [8], but is modified to tolerate spelling errors as long as the first few characters of the search string are error free, and return search completions even if the string

being completed has a spelling error. Whenever a value is found, it is added to the segment in which it was found. This process yields a set of segments, each segment containing a list of values, e.g., the segment 'red' can contain the values '4' (a color), and 'red hat' (an operating system name); *b*) **labeling**, i.e., indicating what to do with a segment value. A label assigned to a segment indicates one of three roles, namely that the segment contains: 1) a value v that will be assigned to some field F; 2) a field name, hinting that the value of an adjacent segment must be assigned to F; or, 3) no useful information. During this process, we also determine an actual segmentation. A segmentation denotes a list of segments such that the whole query can be reconstructed by concatenating each segment from the list. This also implies that the chosen segments may not overlap each other. In Section 4, we discuss how we apply our stack decoder for this labeling task; and *c*) **normalizing**, i.e., (slightly) rewriting the field value into a format accepted by the form, if necessary. A field has a format in which a value must be specified. For example, a field may require that a time be entered as `hh:mm`, i.e., two digits for the hour, a colon, and two digits for the minutes. If the query contains a time as *ten to five am*, it should be normalized to `04:50`. For normalizing dates and times, we created a separate function. Other normalizations, like when the color *red* actualy has a value `4`, or when a word is misspelt, are dealt with using a dictionary.

4 Stack Decoding

Given a free-text query, we first segment it into a set of segments, each segment containing a list of values. Next, we initialize a sorted stack with an empty *path*. A path has a score and a list of labeled segments. We then iteratively decode the query as follows: 1) remove the best path from the stack; 2) look up all segments S that follow immediately after the last segment in the path; 3) for each value in each segment $s \in S$, determine the possible labels and label the segment; 4) for each labeled segment, create a new path and add it to the stack. The process iteratively extends partial paths to become complete paths. When a path is complete, it is removed from the stack and stored as a result for further processing. The decoding stops when the stack is empty, or when some stopping criterion is met (e.g., some max decoding time t has elapsed).

Scoring. A path's score is based on the field values, and on the field order which was discussed in Section 3.2. The score of a value v from some closed category C is initially modeled as a uniform probability of $\frac{1}{|C|}$ for observing v. The score of a numeric value from an open category is determined heuristically: based on the number of digits, it diminishes quadratically such that a 4-digit value gets the highest score, then 3-digit and 5-digit values, and so on. An important issue in stack decoders is the comparability of partial paths [2, 20]. We lower a partial path's score by the number of characters that must yet be processed. This basically estimates for any partial path what the score would be if the whole query was processed. Note that lowering the score too much causes the decoder to proceed in a depth-first search manner instead of best-first search manner.

Pruning. With enough time and memory resources, we could theoretically examine all possible paths, including invalid ones. In practice however, we have little time and resources and need to reduce the time spent on processing invalid paths. Therefore, we prune partial paths that violate the dependency, field frequency, or negative conditional constraints defined in Section 3.1.
Boosting & discounting. The speed of a stack decoder depend on it repeatedly choosing and expanding the best partial path until it finds the best complete path. The choice is based on fields and values seen so far, without regard for possible further fields and values. This is not always desirable. For example, consider the query *BMW 2000 euro* and a form with three fields: `make`, `year` and `price`. The segment 'BMW' is labeled as `make` and we must now label the segment '2000'. If we only considered segments up to and including '2000', then both labels `year` and `price` would seem fine. However, if we would have looked ahead when labeling '2000' as `year`, we would have known that this label is not likely, therefore we would have lowered the position of this path in the stack. The process of looking ahead and deciding to raise or lower a path's position in the stack is referred to as *boosting* or *discounting*, respectively. We can rank the complete paths by their original scores or by the boosted & discounted scores.

5 Data used for Evaluation

Our aim was to obtain realistic queries under three conditions: *a*) **multi-domain** search environment. Participants should be able to search in different domains, like travel planning or second hand cars, and get real-time query suggestions; *b*) **multi-site** domains. Each domain should have different sites that may or may not offer the same search functionality. For example, in travel planning, one site might offer bus travel results, while other sites offer train or flight results; and, *c*) **minimal query bias**. Participants should not be persuaded to any kind of information need nor to any structure in which the they can phrase a query.

5.1 Data Acquisition

We setup an online search system covering 3 multi-site domains, and instructed the participants that they could search these domains. We briefly describe the domains and instructions for the participants. The **travel planning** domain has 3 sites, each providing either bus, train, or flight travel information. Instruction: *Find travel advice (for example, a traintrip to someone you know) and rate the result.* The **second hand cars** domain has 5 sites, each having a web form with fields for at least minimum price, maximum price, make, and model. Instruction: *Find cars with specific characteristics (for example, find cars with characteristics like your own car or a car of someone you know) and rate the result.* The **currency exchange** domain has 3 sites, each with a form that has three input fields (*from* currency, *to* currency, and *amount*). Instruction: *Find the exchange rate (of currencies of your choice) and rate the result.*

Participants started with a training session in which they could issue multiple queries in each of the three domains. Whenever a result was clicked on, a box

appeared asking to rate the result as either: 'completely wrong', 'iffy', or 'completely right'. After rating a result, the system prompted for the next domain. It is natural to rephrase the query if a system returns no or unsatisfying results. However, if a participant believed that the query could have been answered corectly by the system, he/she could indicate this and optionally describe what kind of results should have been returned. During the training session, participants got acquainted with the system and discovered the search functionality by themselves. After introducing all domains, the participant was asked to conduct 10 different searches and rate at least one result of each search request. As an incentive to continue with the experiment, a score was shown based on, amongst others: the number of queries issued, the number of results rated, and the search functionality[1] discovered so far. Participants could quit whenever they wanted.

5.2 Manual Analysis and Labeling

We manually analysed all submitted queries and specified which forms could return relevant results and how the forms should be filled out. For each form, we compiled a testcorpus specifying the set of field-value assignments for the queries that make sense to the web form. We then measured how much our judgments agreed with those of the participants using the *overlap* between our manually assigned query-result pairs and those of the participants. Overlap is defined as the size of the intersection of the sets of relevant results divided by the size of their union, and has been used by several studies for quantifying the agreement among different annotators [18, 9, 5]. We needed to compile the testcorpora ourselves because: first, participants did not (and were not expected to) find and label all correct results. Second, the system may not have returned any correct results, making it impossible for participants to label all correct results.

5.3 Data Obtained

In total, 47 participants interacted with the system and 23 opted to state their age and gender, resulting in 17 males (age: 19–81, avg. 39) and 6 females (age: 25–41, avg. 30). We analyzed 363 queries, but nearly half were invalid, either missing mandatory fields or asking information that was out of scope. Examples of invalid queries are: *to Amsterdam*; *how long is the Golden Gate bridge*; *kg to pound*; and, *for sale: 15 year old mercedes*. In total, we labeled 194 valid queries containing enough information to fill out a form in our experiment. When multiple forms could be filled out for a given query, we chose the ones in which we could specify most key-value pairs of the query. A summary of the results for the travel planning, currency exchange, and second hand cars domains is shown in Table 1. The rows 'A' to 'K' each correspond to a form in the specified domain and shows: *Q:* the number of queries submitted in that form; *Max:* the maximum number of different ways to fill out that form for a single query; *Avg:* the average

[1] Search functionality here means the number of different fields in all clicked results, divided by the total number of fields from all web forms configured in the system.

number of filled out forms per query; and, *Std.dev:* the standard deviation from this average. The row 'All' shows the results when aggregating all forms, and should be interpreted as: 194 queries were submitted in this aggregated form; there was a query that could be filled out in 19 different ways; there were 2.99 filled out forms per query on average, with a standard deviation of 2.43.

Table 1. Manual labeling results.

Travel	Q.	Max.	Avg.	Std.dev.
A	52	8	1.19	0.99
B	5	5	1.80	1.79
C	12	3	1.25	0.62
Currency				
D	61	1	1.00	0.00
E	61	2	1.03	0.18
F	62	1	1.00	0.00

Cars	Q.	Max.	Avg.	Std.dev.
G	24	2	1.04	0.20
H	59	7	1.39	1.16
I	61	9	1.38	1.29
J	52	4	1.12	0.51
K	49	3	1.20	0.58
Merged				
All	194	19	2.99	2.43

A result (i.e., a filled out form) denotes a set of field-value pairs. On a result level, the agreement of our judgments and those of the participants is 0.33, which is consistent with the "key" agreement reported in [5]. Though it might seem low, it is a direct result of the strict comparison: one slightly different field value causes results to disagree completely. If we considered field-value pairs instead, and averaged the field-value agreement per result, the agreement is 0.68.

6 Evaluating the Stack Decoder

We evaluated our system using the data described in Section 5.3. We investigated how different stopping criteria, boosting, discounting, and ranking on original or on boosted scores, affected the decoding time and retrieval performance—which was measured using MAP (Mean Average Precision [18]). Table 2 lists the 6 stopping criteria that we used. The decoding stopped when: a maximum of r results was found; or, more than t time elapsed during decoding; or, the next result's score was lower than some absolute minimum $abs.min$; or, when it was lower than some mimimum $rel.min$ relative to the best result. Further, we tested two settings for pruning probably irrelevant paths based on the percentage of the query that was ignored. A path was discarded if more than j% was ignored (e.g., due to unknown words). One (fairly strict) setting required the system to interpret at least 60% of the query, while the other required only 20%.

Table 2. Stopping criteria, sorted by number of results and "strictness".

	Results	Time	Abs. min.	Rel. min.	Ignore %
A	10	0.5	-200	-150	40
B	10	0.5	-200	-150	80
C	10	0.5	-600	-550	80
D	50	45	-200	-150	40
E	50	45	-200	-150	80
F	50	45	-600	-550	80

Table 3. Results obtained without training. The headers A–F denote stopping criteria (see Table 2). The leftmost letters B, D, and R denote *boosting*, *discounting*, and *ranking* by original score, respectively. *Time* is the average query decoding time. *Map1* and *Map2* are the MAP of filled out forms, and of segmentation & labeling, respectively.

(a) Evaluation results, averaged over the individual tests per form.

			A			B			C			D			E			F		
B	D	R	MAP1	Time	MAP2	MAP1	Time	MAP2	MAP1	Time	MAP2	MAP1	Time	MAP2	MAP1	Time	MAP2	MAP1	Time	MAP2
-	-	-	0.485	0.05	0.549	0.551	0.04	0.608	0.622	0.07	0.625	0.485	0.05	0.548	0.551	0.05	0.608	0.629	0.31	0.627
0	1	0	0.502	0.04	0.576	0.568	0.04	0.636	0.641	0.07	0.656	0.501	0.05	0.575	0.568	0.05	0.635	0.647	0.30	0.653
1	0	0	0.503	0.04	0.567	0.569	0.04	0.627	0.639	0.07	0.641	0.503	0.05	0.566	0.569	0.05	0.626	0.647	0.16	0.645
1	1	0	0.504	0.04	0.579	0.570	0.04	0.638	0.640	0.07	0.652	0.504	0.04	0.577	0.570	0.04	0.637	0.649	0.15	0.656
0	1	1	0.519	0.04	0.583	0.582	0.04	0.644	0.642	0.07	0.664	0.521	0.05	0.586	0.583	0.04	0.647	0.649	0.31	0.666
1	0	1	0.521	0.04	0.580	0.583	0.04	0.642	0.642	0.07	0.656	0.522	0.05	0.583	0.585	0.05	0.645	0.649	0.16	0.664
1	1	1	0.522	0.04	0.585	0.584	0.04	0.646	0.643	0.07	0.659	0.523	0.04	0.588	0.586	0.04	0.649	0.650	0.16	0.668

(b) Evaluation results of the aggregated web forms.

			A			B			C			D			E			F		
B	D	R	MAP1	Time	MAP2	MAP1	Time	MAP2	MAP1	Time	MAP2	MAP1	Time	MAP2	MAP1	Time	MAP2	MAP1	Time	MAP2
-	-	-	0.414	0.09	0.523	0.444	0.08	0.547	0.442	0.23	0.539	0.446	0.11	0.556	0.475	0.10	0.581	0.504	1.64	0.597
0	1	0	0.424	0.08	0.539	0.454	0.08	0.562	0.453	0.22	0.554	0.455	0.11	0.571	0.484	0.10	0.596	0.516	1.66	0.611
1	0	0	0.427	0.08	0.539	0.463	0.08	0.570	0.461	0.22	0.553	0.462	0.14	0.573	0.495	0.13	0.603	0.514	1.09	0.608
1	1	0	0.428	0.08	0.545	0.464	0.08	0.576	0.464	0.22	0.558	0.461	0.13	0.576	0.493	0.12	0.606	0.517	1.16	0.613
0	1	1	0.402	0.08	0.509	0.434	0.08	0.540	0.439	0.22	0.540	0.417	0.10	0.527	0.452	0.10	0.558	0.479	1.88	0.579
1	0	1	0.407	0.08	0.511	0.440	0.08	0.548	0.447	0.22	0.538	0.424	0.13	0.533	0.457	0.12	0.567	0.477	1.23	0.579
1	1	1	0.408	0.08	0.514	0.441	0.08	0.550	0.449	0.22	0.537	0.422	0.12	0.532	0.456	0.12	0.567	0.478	1.25	0.582

6.1 Untrained and Individual, "per Form" Evaluation

One at a time, we loaded a form's dictionary and constraints and ran its tests. We did not train the system but used a uniform field order distribution[2]. Table 3(a) shows the averaged results of the individual tests, weighted by the number queries per form. The results show that we should not prune "improbable" paths beforehand, i.e., paths with low scores and in which up to 80% of the query is ignored. It also shows that boosting and discounting affects MAP, especially with relatively strict stopping criteria; and that as the criteria relaxes, the effect decreases. This is due to the relatively small search space in the individual tests. The stopping criteria limit the part of the search space can be inspected, and the boosting and discounting try to sneak in as many relevant paths to this limited space as possible. Thus when the stopping criteria are sufficiently relaxed, the effects of boosting and discounting will naturally decrease. For the individual tests, we can conclude that boosting reduces decoding time, and that boosting, discounting, and ranking on original scores yields the best retrieval performance.

6.2 Untrained and Collective, "Aggregated Forms" Evaluation

We collectively loaded all forms into our system. This causes the search space to be much larger, and aside from determining how to fill out a form, the system must also determine which forms to return in the first place. We also aggregated the tests, specifying for each query all forms that should be returned and

[2] Except in one form where we manually specified that "departure" fields were more likely followed by "destination" fields, instead of other fields. However, this was done before going online and gathering data, so before we had even seen the test data.

all ways of filling out a form for that query. From the collective evaluation results in Table 3(b), we can conclude that: we should not prune "improbable" paths beforehand, which agrees with the results of the individual tests; Boosting, discounting, and ranking on the boosted & discounted scores yields the best retrieval performance, which contrasts with the individual tests where you should rank on the original scores; Finally, our system effectively brokers over different sites across different domains (e.g., travel planning, currency, second hand cars).

6.3 Baseline Evaluation

To our knowledge, no other system translates free-text queries to filled out forms, normalizes values, and checks against constraints. However, LingPipe[3] is a suitable baseline, as it recognizes named entities by segmenting & labeling text, and is a widely used text processing toolkit. We manually segmented the queries and labeled each segment. Filled out forms naturally correlate with segmented & labeled queries. However, due to normalization and constraint checking, there may not be a valid filled out form even if the query is correctly segmented.

We evaluated both systems on their prediction of which query segments contained field values and what label to assign to each segment. We used 3 data sets to simulate "untrained" up to "fully trained" systems: *set A* contains uniform field transitions and uniform token counts; *set B* contains field transitions from the queries, but uniform token counts; and, *set C* contains both field transitions and token counts taken from the queries. We cross-validated LingPipe using out-of-the-box settings for named entity recognition. In each test, we loaded the dictionary but no regular expressions because they cannot be used together (at least, not out-of-the-box). We cross-validated our system using the parameters from Table 3 that gave the best filled out forms (i.e., with the highest *MAP1*, and lowest time if *MAP1* is equal). So, for the individual tests we used {criteria=C; B,D,R=1,1,1}, and for the collective tests {criteria=B; B,D,R=1,1,0}.

The segmentation & labeling results are shown in Table 4. Row A denotes results of untrained systems (i.e., they are only "trained" on uniform distributions). Rows B and C denote 5-fold cross validation results of the systems. The collective cross-validations tests are stratified, i.e., 1/5-th of the queries of each form is used in each fold. As expected with no training (row A), LingPipe performs poorly, which constrasts with our untrained system. For now, our system

Table 4. Segmentation & labeling results. Training set A involves no training. In B we train on field transitions, and in C on both field transitions and token counts.

(a) Averaged individual tests.

LingPipe			Our system		
Training set	MAP	Time	Training set	MAP	Time
A	0.302	0.27	A	0.659	0.07
B	0.459	0.04	B	0.717	0.05
C	0.708	0.04	-	-	-

(b) Collective tests.

LingPipe			Our system		
Training set	MAP	Time	Training set	MAP	Time
A	0.117	66.88	A	0.576	0.08
B	0.207	5.16	B	0.629	0.07
C	0.289	5.11	-	-	-

[3] Alias-i. 2013. LingPipe 4.1.0. http://alias-i.com/lingpipe (accessed March 1, 2013)

can only train on field transitions (row B), and this already improves performance. Training LingPipe on only field transitions also improves performance; but training on both transitions and token counts (for which it was designed) gives the biggest improvement. Since LingPipe does not know that once it uses labels of one form it cannot use labels of others, it performs very poorly in the collective tests. Then again, it was not developed for such a task.

6.4 Further Discussion

The problem of converting non-structured queries to structured queries goes back as far as 30 years, and solutions were proposed based on heuristics, grammars, and graphs. Due to space limits however, we focussed on probabilistic, state-of-the-art approaches to segmentation & labeling in Sect. 2. In Sect. 3, the form's constraints must be specified manually. Automatic detection of such constraints would be beneficial and warrants further research. Regarding the results, after inspecting a sample of the results we noted that OOV (out-of-vocabulary) words were lowering retrieval performance. Some OOV words can easily be added (e.g., new car models), but others consitute natural language phrases that must be interpreted in context and cannot easily be added. The problem of OOV words must be further researched. Online learning using click log data is potentially the cheapest solution, but comes with several challenges (see Section 5.2). We also noticed that few labels were used for numerical tokens, e.g., a number was often intended as a price, but never as the engine displacement. This makes it easier for LingPipe to guess the right label, as it is ignorant of the actual possible labels for each numerical token and just considers the labels seen during training. Finally, we will extend our system to train on token counts as well (i.e., data from *set C*), which should further improve retrieval results.

7 Conclusion

We introduced a novel and flexible method for translating free-text queries to structured queries for filling out web forms. This enables users to search structured content using free-text queries. In contrast, web search engines struggle to index structured content from web databases, and users cannot enter structured queries in a typical web search engine. Our method consists of three steps: segmenting, labeling, and normalizing. We use the constraints imposed by web forms to prune the search space and apply boosting & discounting heuristics. Our results confirm that our heuristics are effective, reducing decoding time and raising retrieval performance. We also showed that without training, our system outperforms an untrained baseline on the individual and the collective tests. Compared to a trained baseline, our trained system is still better on the individual tests and outperforms the baseline on the collective tests.

Acknowledgment

We thank the anonymous FQAS'13 reviewers for their comments. This research was funded by the Netherlands Organization for Scientific Research, NWO, grant 639.022.809.

Bibliography

[1] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE '07*, pp. 6–20, april 2007.

[2] L. R. Bahl, F. Jelinek, and R. L. Mercer. A maximum likelihood approach to continuous speech recognition. In *Readings in speech recognition*, pp. 308–319. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[3] V. Borkar, K. Deshmukh, and S. Sarawagi. Automatic segmentation of text into structured records. In *SIGMOD '01*, pp. 175–186, New York, USA, 2001. ACM.

[4] K. C.-C. Chang, B. He, C. Li, M. Patel, and Z. Zhang. Structured databases on the web: observations and implications. *SIGMOD Record*, 33(3):61–70, 2004.

[5] T. Demeester, D.-P. Nguyen, R. B. Trieschnigg, C. Develder, and D. Hiemstra. What snippets say about pages in federated web search. In *AIRS 2012, Tianjin, China*, Lecture Notes in Computer Science. Springer, 2012.

[6] J. Forney, G.D. The viterbi algorithm. *Proc. of the IEEE*, 61(3):268–278.

[7] M. Hagen, M. Potthast, B. Stein, and C. Braeutigam. Query segmentation revisited. In *WWW '11*, pp. 97–106, New York, NY, USA, 2011. ACM.

[8] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. In *TOIS '02*, 20(2):192–223, Apr. 2002.

[9] D. Hiemstra and D. A. van Leeuwen. Creating an information retrieval test corpus for dutch. In *CLIN 2001*, volume 45 of *Language and Computers - Studies in Practical Linguistics*, pp. 133–147, Amsterdam, The Netherlands, 2002. Rodopi.

[10] J. Kiseleva, Q. Guo, E. Agichtein, D. Billsus, and W. Chai. Unsupervised query segmentation using click data: preliminary results. In *WWW '10*, pp. 1131–1132, New York, NY, USA, 2010. ACM.

[11] J. Kiseleva, E. Agichtein, and D. Billsus. Mining query structure from click data: a case study of product queries. In *CIKM '11*, pp. 2217–2220, New York, NY, USA, 2011. ACM.

[12] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML '01*, pp. 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[13] X. Li, Y.-Y. Wang, and A. Acero. Extracting structured information from user queries with semi-supervised conditional random fields. In *SIGIR '09*, pp. 572–579, New York, NY, USA, 2009. ACM.

[14] Y. Li, B.-J. P. Hsu, C. Zhai, and K. Wang. Unsupervised query segmentation using clickthrough for information retrieval. In *SIGIR '11*, pp. 285–294, NY, USA, 2011. ACM.

[15] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. *Proc. VLDB Endow.*, 1(2):1241–1252, 2008.

[16] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. of the IEEE*, 77(2):257–286, 1989.

[17] N. Sarkas, S. Paparizos, and P. Tsaparas. Structured annotations of web queries. In *SIGMOD '10*, pp. 771–782, New York, NY, USA, 2010. ACM.

[18] E. M. Voorhees. Variations in relevance judgments and the measurement of retrieval effectiveness. *Inf. Processing and Management*, 36(5):697–716, 2000.

[19] X. Yu and H. Shi. Query segmentation using conditional random fields. In *KEYS '09*, pp. 21–26, New York, NY, USA, 2009. ACM.

[20] Y. Zhang and S. Clark. Syntactic processing using the generalized perceptron and beam search. *Computational Linguistics*, 37(1):105–151, 2011.