Head, Modifier, and Constraint Detection in Short Texts

Zhongyuan Wang [†,‡1], Haixun Wang [§2], Zhirui Hu [#3]

[†]*School of Information, Renmin University of China, Beijing, China*
[‡]*Microsoft Research, Beijing, China*
[1]wzhy@outlook.com

[§]*Google Research, Mountain View, CA, USA*
[2]haixun@google.com

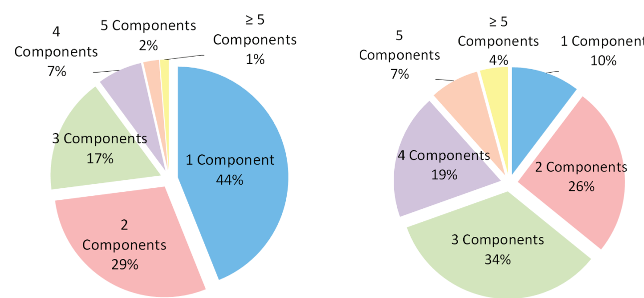
[#]*Department of Automation, Tsinghua University, Beijing, China*
[3]hzr07@mails.tsinghua.edu.cn

***Abstract*—Head and modifier detection is an important problem for applications that handle short texts such as search queries, ads keywords, titles, captions, etc. In many cases, short texts such as search queries do not follow grammar rules, and existing approaches for head and modifier detection are coarse-grained, domain specific, and/or require labeling of large amounts of training data. In this paper, we introduce a semantic approach for head and modifier detection. We first obtain a large number of instance level head-modifier pairs from search log. Then, we develop a conceptualization mechanism to generalize the instance level pairs to concept level. Finally, we derive weighted concept patterns that are concise, accurate, and have strong generalization power in head and modifier detection. Furthermore, we identify a subset of modifiers that we call constraints. Constraints are usually specific and not negligible as far as the intent of the short text is concerned, while non-constraint modifiers are more subjective. The mechanism we developed has been used in production for search relevance and ads matching. We use extensive experiment results to demonstrate the effectiveness of our approach.**

I. Introduction

A large variety of applications need to handle short texts such as search queries, ads keywords, tweets, image captions, etc. Understanding short texts is a big challenge for machines. Unlike long texts and documents, for which we can use "bag of words" based statistical approaches [1] to analyze, short texts do not contain enough information or statistical signals to make the analysis meaningful. Furthermore, short texts are usually not well-formed sentences. For example, queries submitted to search engines usually do not follow grammar rules. Consequently, approaches based on sentence structure analysis [2] do not work well either.

In this paper, we focus on head, modifier, and constraint detection for short texts. A short text contains head and modifier components, where head represents the intent, and modifier limits the scope of the intent of the short text. Take a search query "popular iphone 5s smart cover" as an example. The query consists of three components: "popular," "iphone 5s," and "smart cover." It is obvious that the intent of the query is to find "smart cover," which makes "smart cover" the head component, and "iphone 5s," "popular" modifier components. However, not all modifiers are equal. Compared with "popular," which is more subjective, "iphone 5s" limits the intent in a more specific way. For a search query, we may drop modifier "popular" without changing the query intent, while dropping "iphone 5s" will result in many irrelevant matches. In this paper, we call modifiers such as "iphone 5s" *constraints*, and modifiers such as "popular" *non-constraint modifiers* or *pure modifiers*. Clearly, differentiating constraints from non-constraint modifiers is important.



(a) # of queries

(b) # of distinct queries

Fig. 1. Components in Search Queries

Typically, a short text contains one or more heads, zero or more modifiers. We analyzed 1 week worth of search log (from 07/25/2012 to 07/31/2012) of BING, and used big dictionaries such as Freebase [3], [4] and Probase [5], [6] to identify components in each query. As Fig. 1 shows, about 56% of search has 2 or more than 2 components (each component may contain multiple words). If we consider distinct queries only, the ratio goes up to 90%. This means detecting components and identifying their roles as head, modifier, and constraint is critical in understanding the search query. In our discussion, we use examples that contain one head and one modifier, but the techniques we develop handle all cases, and we address all of them in the paper.

Head, modifier, and constraint detection for short texts is a challenging task for the following reasons.

- *Short texts such as search queries usually do not observe grammar rules*. Hence, simple linguistic methods for head and modifier detection may not apply. For example, a simple linguistic rule says in a noun phrase, the last noun is the head, and the other components to its left are modifiers. However, for "popular smart cover iphone 5s," this is not true. Other statistical approaches have been developed. Bendersky et al [7] assign weights to terms in a query, and introduce statistics features to train an MRF model for head modifier detection. It needs a large labeled corpus, and more importantly, the head-modifier relationships are not explictly detected.

- *Head, modifier, and constraint detection requires external knowledge.* Our goal is to devise a *general purpose* mechanism, rather than mechanisms for a specific domain. Some existing work classifies a query to a category in a predefined taxonomy [8], [9], [10], and the category can be considered as the head of the short text. However, the effectiveness of this approach is limited by the coverage and the granularity of the taxonomy. For example, queries such as "job search" and "job interview" are in the same category of "job" but have different intent. Some other work tries to infer the intent of the input by fitting it into templates that are common in a specific domain [11], [12]. There is also some work mining entity-attribute relationships but not specifically head-modifier relationships [13], [14], [15] and their performance depends on the seed entity-attribute pairs chosen for each domain.

In order to handle noisy, ambiguous, and sparse input, we need knowledge beyond the input:

1) (*Instance-level head-modifier knowledge*) We need to know, when "smart cover" and "iphone 5s" appear together, no matter in what order, "smart cover" is the head, and "iphone 5s" is the constraint.
2) (*Conceptual knowledge*) We need to know "smart cover" is an *accessary*, and "iphone 5s" is a *device*;
3) (*Concept-level head-modifier knowledge*) We need to know, when an *accessary* and a *device* appear together, the *device* is the constraint and the *accessary* is the head.

Our approach of head and modifier detection is to derive head-modifier patterns at the concept level of the following form:

$$(concept_{[head]}, concept_{[modifier]}, score) \tag{1}$$

One example might be

$$(accessary_{[head]}, device_{[modifier]}, 0.9)$$

which indicates that when an accessary and a device appear together in a short text, it is more likely (with score 0.9) that the accessary is the head and the device is the modifier. With such knowledge, for any input, we can decide which patterns in the knowledgebase match the input. Finally, using the patterns and their corresponding scores, we can infer the most likely head and the most likely modifiers in the input.

There are three major challenges. First, the knowledge we acquire must have enough coverage so that we can handle all possible input. For example, in (1), *head concept* and the *modifier concept* are in a predefined, fine-grain concept space that contains millions of concepts. Second, we should try to avoid deriving conflicting patterns, that is, the knowledge we acquire cannot contain a pattern that says *device* is a head, *accessory* is a modifier, and another pattern that says the opposite. However, since patterns are derived from independent instances, such inconsistency may arise. We need to design a sophisticated conceptualization process to reduce conflicting patterns. Third, as we mentioned, we differentiate constraints from non-constraint modifiers. It is important to tell them apart. The intuition is that pure modifiers are subjective terms such as "best," "top," "well-known," "popular," etc, and they are often used across all domains. Based on these observations, we build a modifier network from a knowledgebase and use betweenness centrality to mine pure modifiers.

Our contributions

To the best of our knowledge, this is the first unsupervised approach for head, modifier, and constraint detection for general, open-domain short texts. The technique we describe in this paper is in production for search relevance and ads matching. Following is a summary of our contributions:

- *We introduce an unsupervised, open domain mechanism for head, modifier, and constraint detection.* In comparison, existing work requires labeled data and are domain specific.
- *We build a concept pattern knowledgebase to model head-modifier relationship at the concept level.* We "lift" head-modifier relationships at the instance level (e.g., "smart cover" is a head and "iphone 5s" is a modifier) to the concept level (e.g., *accessory* is a head and *smart phone* is a modifier) so that they have more generalization power.
- *The mechanism for head, modifier, and constraint detection is light weight and highly efficient.* The concept pattern knowledgebase is small but has strong generalization power. This makes it possible to handle millions of open domain short texts instantly.

Paper organization

The rest of the paper is organized as follows. Section II describes an overall framework. Section III derives concept level head-modifier patterns. Section IV conducts head, modifier, and constraint detection for short texts. Section V finds non-constraint modifiers. Section VI gives experiment results and compares our approach with other methods. Section VII introduces related work. Section VIII concludes our work.

II. Framework

Fig. 2 depicts the framework we use for head, modifier, and constraint detection. It contains two offline components, which acquire knowledge respectively for i) non-constraint modifiers, and ii) head-modifier concept patterns, and an online component, which performs head, modifier, and constraint detection using the knowledge acquired offline.

As we mentioned, we classify modifiers into two categories: constraints and non-constraint modifiers (or pure modifiers). To find terms that are often used as pure modifiers, we construct a modifier network. For example, from "large developed country," "developed country," and "country," we derive potential modifiers "large" and "developed." In the network, nodes denote either head concepts (e.g., "country")

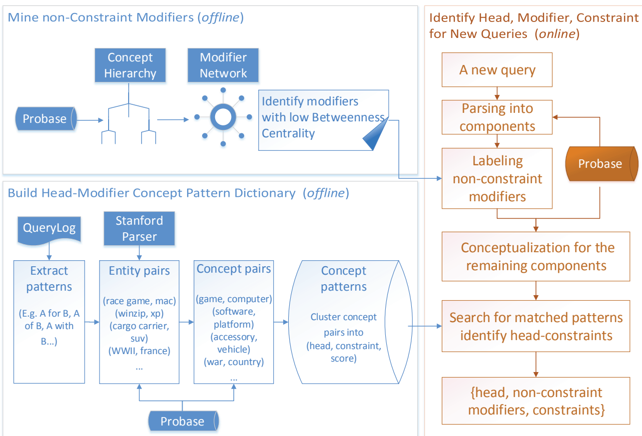


Fig. 2. Framework for head, modifier, and constraint detection

or modifiers (e.g., "large" and "developed"), and edges denote modifying relationships. We show that pure modifiers can be detected using a measure known as betweenness centrality in graph analysis. We give more details in Section V.

A more challenging task, which is also the major focus of this paper, is to identify head-modifier patterns in the concept space. We first acquire instance level head-modifier pairs such as (race game$_{[head]}$, mac$_{[modifier]}$). We then conceptualize them into the concept level. This which gives us head-modifier patterns such as (game$_{[head]}$, computer$_{[modifier]}$). We describe the above process in detail in Section III.

Using the acquired knowledge, we perform head, modifier, and constraint detection in a short text. We first identify and remove pure modifiers. Then, we form candidate head-modifier pairs. Finally, we match the candidates against the concept head-modifier patterns through conceptualization. This enables us to identify head-modifier we have never seen before. We describe the above process in detail in Section IV.

III. Mining Concept Patterns

In this section, we describe our method of finding concept-level head-modifier patterns.

A. Probase: A large scale IsA taxonomy

We use an isA taxonomy known as Probase[1] [5] to conceptualize instance level head-modifier pairs to concept level.

Probase is a large network of multi-word terms, where terms can be entities (e.g., "Barack Obama") and concepts (e.g., "USA President"). There are other types of nodes in the network, including attributes, verbs, adjectives, etc., but in this work we are not concerned with those types. The version of Probase we use contains 2.7 M concepts and 40 M entities. Probase thus provides a huge concept space that covers all concepts of worldly facts. We use a mechanism to efficiently recognize Probase concepts and entities in a short text. We omit the discussion here due to lack of space.

Terms in Probase are connected by a variety of relationships. Here, we focus on the *isA* relationship (although

[1]Probase data is publicly available at http://probase.msra.cn/dataset.aspx

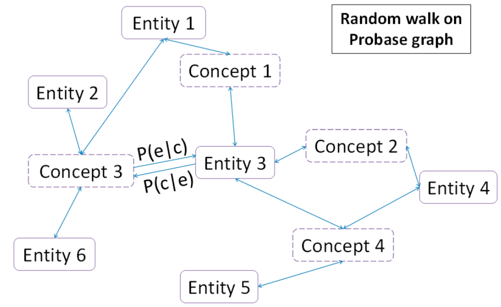


Fig. 3. The Probase Semantic Network

other relationships such as *isPropertyOf* are also important to conceptualization). The *isA* relationship exists between entities and concepts (e.g., "Barack Obama" *isA* "USA President"), or between sub-concepts and concepts (e.g., "USA President" *isA* "Celebrity"). The relationship between e and c, where e is an entity or a subconcept in concept c, is weighted as follows:

$$P(e|c) = \frac{n(e,c)}{n(c)}, \qquad P(c|e) = \frac{n(e,c)}{n(e)} \tag{2}$$

where $n(e,c)$, $n(c)$, and $n(e)$ denote the frequencies of e and c occur together, e occurs independently, and c occurs independently when they are observed in information extraction.

The weights have intuitive meanings. The probability $P(e|c)$ tells us how typical or popular e is as far as concept c is concerned, and $P(c|e)$ tells us how typical c is given e. For example, knowing that both "poodle" and "pug" are dogs is sometimes not enough. We may also need to know that "poodle" is a much more popular dog than "pug," that is, when people talk about dogs, listeners are more likely to think of the image of a "poodle" rather than that of a "pug." Such information is essential for understanding, and is captured by the fact that $P(poodle|dog) > P(pug|dog)$ in Probase.

B. Instance-level Head-Modifiers

In order to model head-modifier relationships at the concept level, we first obtain a large number of instance level head-modifier relationships. Our intuition is the following. Although it is difficult for machines to identify heads and modifiers from queries "iphone 5s smart cover" or "smart cover iphone 5s" directly, we know the same search intent is also expressed in other forms, for example, "smart cover **for** iphone 5s." In the latter form, it is clear that "smart cover" is the intent. This suggests that in queries where "smart cover" and "iphone 5s" occur together, "smart cover" is likely to be the head even if they are not connected by the preposition **for**.

Prepositions play an important role in identifying head and modifiers [16], [17]. We evaluate a set of prepositions, and find that when prepositions 'for,' 'of,' 'with,' 'in,' 'on,' 'at' are used to connect two known terms A and B (e.g. "A for B," "A of B," "A with B"), it is almost always true that A is the head and B is the constraint. We thus use the following syntactic pattern to extract pairs of (A, B) from the query log:

$$\{head\ [for|of|with|in|on|at]\ modifier\}$$

To ensure correct extraction, we use Probase as a dictionary, that is, the heads and constraints must be terms in Probase. Although Probase is big, certainly there are valid terms not included there. This is however not a problem because our goal is to find concept-level head-constraint pairs, and concepts can be derived from instances that are known to Probase.

C. Concept-level Head-Modifiers

From the instance-level head-modifier relationships obtained above, we derive concept-level relationships. This gives us a concise model that can be generalized to cover more instance-level occurrences. Previous work context-dependent conceptualization [18] incorporates LDA to conceptualization. Because it has coverage and scalability issues, it is not suitable for our scenarios. In general, the "goodness" of a model is measured by a trade-off between its complexity and accuracy. We show that the semantic approach we adopt offers a concise model with great generalization power.

1) Levels of Conceptualization: An instance maps to many concepts, some very specific and others very general. We can map a pair of instances, say, (smart cover, iphone 5s) to a pair of concepts in two extreme ways. First, we can map it to itself, i.e., we treat "smart cover" and "iphone 5s" as concepts on their own. But such a mapping does not have generalization power, as it covers nothing else except itself. Second, we can map it to (object, object), where "object" is the root concept that all instances belong to. But such a mapping is not useful, as it does not have power of telling heads and constraints apart.

A more challenging problem is the following. It may seem alright to map "skype for windows phone" to $(company_{[head]}, device_{[modifier]})$ and "iphone 5s for verizon" to $(device_{[head]}, company_{[modifier]})$. But then, the two resulting patterns are in head-on conflict: When *company* and *device* appear together, the first pattern says *device* is the head, while the second says *company* is the head. Clearly, the mapping is too general or too coarse grained.

The principle of conceptualization is thus two-fold. First, we must avoid concepts that are too specific, because specific concepts have poor generalization power. Besides, we end up with a large number of concept-level head-modifier patterns. Second, we must avoid concepts that are too general. Over generalization leads to conflicting patterns, as each claims territory that it does not own.

2) Conceptualizing Instances: We now show how to properly map a single instance to a set of concepts. As Fig. 3 shows, from an instance e, we can reach e's concepts $C = \{c_1, \cdots, c_n\}$ by following the weighted isA links that denote the typicality of c_i given e, and the typicality of e given c_i.

The criterion of selecting concepts for a term e must take into account both generality and specificity. Consider the following four possible ways of mapping e to c:

1) Map e to c_i if $P(c_i|e)$ is among the top k;
2) Map e to c_i if $P(e|c_i)$ is among the top k;
3) Map e to c_i if $P(c_i|e) \cdot P(e|c_i)$ is among the top k;
4) Map e to itself if e is itself a concept.

The first two options are less desirable because concepts with high $P(c|e)$ values have the risk of being too general. E.g. the top concepts of "iphone" ranked by $P(c|e)$ are *product, device*, etc. This is because general concepts occur more often. On the other hand, concepts with high $P(e|c)$ values have the risk of being too specific as $P(e|c)$ is more likely larger when concept c contains a smaller number of instances. E.g. the top concepts of "iphone" ranked by $P(e|c)$ are *finger-friendly touchscreen phone, apple's mobile device*, etc. To take both generality and specificity into consideration, we choose option 3, that is, we map e to those concepts c_i if $P(c_i|e) \cdot P(e|c_i)$ is large. Intuitively, the score is basically the probability of a 2-step random walk starting from e and coming back to e that goes through c_i. A larger probability indicates stronger corpus evidence of the closeness between c_i and e.

Option 4 is a special case worth careful consideration. An instance e can be a concept itself, and sometimes it is already the most appropriate concept. For example, if e="company," option 3 may lead to concepts such as "organization" or even "object," which are too vague. On the other hand, if e is very specific, say e ="small IT company," then it makes sense to use option 3 to map e to concepts such as "IT company" or "company." Ideally, we want to keep relatively popular concepts that cover a reasonable number of instances, so in our work, we use a concept's entropy as an indicator:

$$H(c) = -\sum_{\text{e is an instance of c}} P(e|c)\log P(e|c) \tag{3}$$

Intuitively, a concept has large entropy if it contains many equally popular instances. For such a concept, we prefer mapping to itself. As an example, a concept such as "device" will have large entropy (7.54), while a concept such as "recording device" will have a small one (1.67). Specifically, we map e to itself if i) e is a concept; ii) $H(e) > H(c)$ for any super concept c of e; and iii) the frequency of e is above a threshold (if e is rare, $H(e)$ may not be meaningful).

In summary, we map an instance e to a set of k concepts C. If e is a qualified concept on its own, then $C = \{e\} \cup top_{k-1}(e)$, otherwise $C = top_k(e)$, where $top_k(e)$ is the top-k concepts obtained using option 3 described above. Furthermore, for any $c_i \in C$, we also give it a score $CS(e, c_i)$:

$$CS(e, c_i) = \begin{cases} 1 & c_i = e \\ P(c_i|e) \cdot P(e|c_i) & c_i \neq e \end{cases} \tag{4}$$

3) Conceptualizing Pairs: To map a set of head-modifier pairs to a (much smaller) set of concept level head-modifier patterns, we first conceptualize the head and the modifier independently, and then combine them to generate concept level head-modifier patterns.

The task of combination, however, is not trivial. For example, the term "apple" conceptualizes to "fruit" or "company." Therefore, "CEO for apple" results in two possible concept pairs: (corporate officer, company) or (corporate officer, fruit). Obviously, (corporate officer, fruit) is wrong. This indicates that we should not conceptualize each head-constraint pair by itself. For the above case, there are similar queries such as "CEO for Microsoft," "CEO for IBM." Both support the (corporate officer, company) pair but not the (corporate officer, fruit) pair. In other words, aggregating different queries has the power of disambiguation.

Specifically, for each instance level head-modifier pair, we conceptualize its head and modifier independently. We combine them in all possible ways to concept pairs. After we obtain all concept pairs (c_i, c_j), where c_i is a head concept and c_j is a modifier concept, from all instance level head-modifier pairs, we rank the pairs by a scoring function:

$$Score(c_i, c_j) = \sum_{u,v} CS(e_u, c_i) \cdot CS(e_v, c_j) \cdot \log N(e_u, e_v) \tag{5}$$

where $CS(e, c)$ is the score of e mapping to c as defined in Eq (4), and $N(e_u, e_v)$ the frequency of the pair (e_u, e_v). We take the logarithm of $N(e_u, e_v)$ to prevent large frequency value having too strong inference on the final score. This ensures that concept pairs supported by a variety of entity pairs rank higher. Wrong concept pairs introduced by ambiguous senses will have a low score and can be filtered out.

Besides ambiguity, there is also the issue of similar concept pairs, as Probase contains concepts that are similar to each other, such as "country" and "nation." Li et al [19] proposed a k-Medoids clustering algorithm to cluster these concepts. We leverage their clustering results directly for clustering concept pairs.

IV. Head and Modifier Detection

We detect head and modifiers in a short text using the acquired concept patterns.

A. Parsing

Given a short text, we first identify all the terms in the short text that we can recognize. We do this by using Probase as a lexicon of terms. During the parsing, if one term is a substring of another term (e.g., New York and New York Times), we choose the longest term[2]. We then remove non-constraint modifiers (Section V describes how we detect them). For the remaining terms, we cluster terms semantically to form components, such that each component is a group that contains one or more semantically similar terms. We do this for two reasons. First, some short texts such as "apple ipad microsoft surface" contain more than one head (e.g., the user wants to compare two products). Second, we want to reduce the number of concept pair candidates for conceptualization. In the above example, it contains 4 terms: "apple," "ipad," "microsoft," and "surface," but only 2 components: {apple, microsoft} and {ipad, surface}, the first of which is related to "*company*," and the second "*device*." This is achieved with a simple co-clustering of concepts and terms by identifying the disjoint cliques [6].

Assume there are k components left. If $k = 1$, we return the component as the head of the short text. In the following, we discuss cases for $k = 2$ and $k > 2$, respectively. In most cases, a component contains a single term only. Thus, we sometimes use a single term to represent a component.

[2] If the longest term is a very rare term, we also consider short terms. We omit the details due to lack of space.

B. Head-modifier detection for 2 components

Consider a short text with two components "smart cover" and "iphone 5s." Fig. 4 demonstrates the process of head-modifier detection.

We first conceptualize "iphone 5s" to {*mobile phone, smart phone, phone, device,* $\cdots$}, and "smart cover" to {*mobile accessory, accessory, part,* $\cdots$}. Each (term, concept) pair (e, c_i) is associated with a score, $CS(e, c_i)$, given by Eq (4).

Then, we search the concept pattern knowledgebase, and find matches, for example, *(accessory, device),* $\cdots$, each of which is associated with a score given by Eq (5).

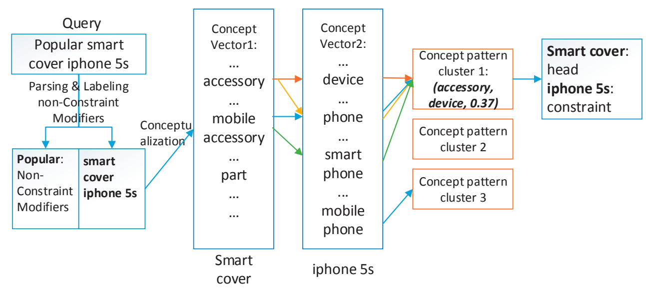


Fig. 4. Head-modifier detection short texts with two components

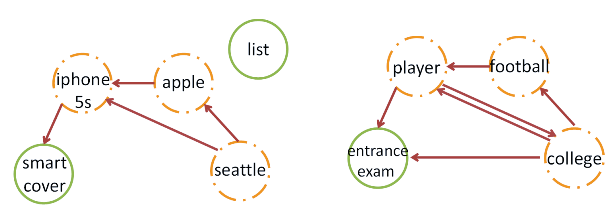
We aggregate the scores to identify the head and the modifier. For two components t_1 and t_2, if $f(t_1, t_2) > f(t_2, t_1)$ then we conclude t_1 is the head and t_2 is the modifier, where $f(t_1, t_2)$ is defined as:

$$f(t_1, t_2) = \sum_{c_1, c_2} CS(t_1, c_1) \cdot CS(t_2, c_2) \cdot Score(c_1, c_2)$$
$$\text{where} \quad CS(t, c) = \sum_{e_i \in comp} CS(e_i, c) \tag{6}$$

Conceptually, the above function is to aggregate the supporting evidence from concept patterns, and decide which component is more likely as the head component.

C. Head-modifier detection for more than 2 components

As we have shown in Fig. 1(b), a large number of search queries have more than two components. To solve this problem, we first use the above process to detect the head-modifier relationship between any 2 components. Then, we represent a query using a directed graph, where nodes represent components and directed edges represent head-modifier relationships between the components. The direction of an edge is from the modifier to the head. Fig. 5 gives examples.



(a) no cycle (b) with cycle

Fig. 5. Components connected by head-modifier relations

It is clear that nodes with no outgoing links represent heads. If the graph is acyclic, we can find a set of paths from a sequence of modifiers to heads. In Fig. 5(a), there are 2 paths: $smart\ cover \leftarrow iphone\ 5s \leftarrow apple \leftarrow seattle$ and $smart\ cover \leftarrow iphone\ 5s \leftarrow seattle$, each of which describes a sequence of modifying relationships among the terms. We can rank a modifier by its closeness to the head in the path. The lower the rank, the less important the modifier. There also can be isolated components in the query, such as the component $list$, which does not associate with any other component. It may be because no head-modifier relationships exist between that component and others, or because we miss some concept patterns in mining the search log. We also leave these isolated components as heads since we do not have evidence saying they are modifiers.

In some (rare) cases, the graph contains cycles, as shown in Fig. 5(b). Cycles are often introduced by ambiguous head-modifier relationships. As an example, for "college football player," we have $player \leftarrow football \leftarrow college$. However, it comes to the pair (*college, player*) and we may have both $player \leftarrow college$ and $college \leftarrow player$, the former of which for the intent of finding "player," and the latter for finding "college." We remove cycles using methods below:

- As edges are associated with weights given by Eq. 6, we break the cycle by removing the weakest edge.
- If there is an obvious head connected to the cycle, the entire cycle becomes a modifier of the head.

Therefore, if the query "college football player" comes alone, we remove the likely weakest edge $college \leftarrow player$. After removing the cycle, we know "player" is the head, while "college" and "football" are modifiers. On the other hand, if the cycle comes with "entrance exam" (as shown in Fig 5(b)), then the head is "entrance exam", and the whole cycle is its modifier.

It is clear that our head-modifier detection approach does not solely depend on words' relative position in a phrase. This makes it useful for short texts such as queries that do not strictly follow the grammar.

V. Mining Non-Constraint Modifiers

We now describe how to find terms that are often used as non-constraint modifiers (pure modifiers). As we mentioned, there is a difference between the two modifiers in "top Seattle hotels," in the sense that "Seattle" is a specific constraint, while "top" is subjective. In some applications, such as search, non-constraint modifiers are often ignored. Furthermore, we observe that non-constraint modifiers are universal and open domain. E.g. "top" can occur in "top movies," "top books," and "top hotels."

Based on the head-modifier principle [16], given "large developed country" and "developed country," we can deduce that "large" is a potential modifier[3]. Besides, we observe that the modifier on the left is more likely to be a non-constraint modifier than the one to its right. For example, people usually say "cheap red shoe" instead of "red cheap shoe."

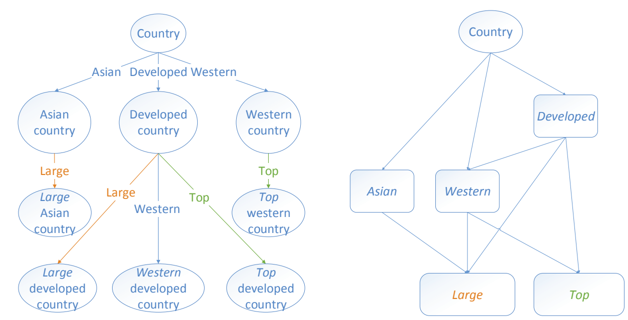
[3]But this is not always the case, for example, "hot" is not a modifier for "hot dog." We solve this problem by noticing "dog" belongs to the animal concept, while "hot dog" to the "snack" or "quick food" concept. Thus, we perform non-constraint modifier detection within each concept domain or concept cluster [19].

Therefore, we consider using a mass of phrases or concepts to mine non-constraint modifiers. Probase is a good choice here because: (1) it contains 2.7 million concepts, including many tail concepts such as "large developing country" that contain non-constraint modifiers; (2) it is across domains.

Our mining process is as follows:

1) Construct modifier networks based on above observations.
2) Calculate the score of each node as a non-constraint modifier in modifier networks.

We use an example to illustrate the process of mining non-constraint modifiers. Consider the concept hierarchy in Fig. 6(a), which is for the concept domain of country. Here, each node is a concept, and each edge is labeled with the modifier on the superconcept. We then transform Fig. 6(a) to Fig. 6(b). This is done by first keeping the root concept unchanged, and coverting edges to nodes. Edges with the same label are mapped to one node. We call the new graph the "modifier network." With this transformation, we construct many modifier networks based on concept clusters.



(a) Concept Hierarchy Tree (b) Modifier Network

Fig. 6. Mining non-Constraint Modifiers

Then we score nodes as non-constraint modifiers in these modifier networks. According to our definitions, if a term is a non-constraint modifier, it is always a non-constraint modifier, regardless of its context. In contrast, heads and constraints depend on their context, which means a term can be a head sometimes, and be a constraint in some other cases. Consequently, head and constraints can be central nodes in some modifier networks, while non-constraint modifiers should always not be central nodes. This makes non-constraint modifiers have low centrality. Both degree and betweenness are measures of a node's centrality in a network. Compared with degree based approaches, betweenness centrality can measure the centrality of a node based on path through globally. Therefore, we use betweenness centrality to decide if a modifier is a non-constraint modifier.

The definition of betweenness centrality of node v is:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{7}$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v. We normalize this betweenness centrality score in each modifier network:

$$NL(g(v)) = \log \frac{g(v) - min(g)}{max(g) - min(g)} \tag{8}$$

Then we aggregate all modifier networks to get the pure modifier score $PMS(t)$ for a term t:

$$PMS(t) = \sum NL(g(t)) \tag{9}$$

Finally, we can calculate a pure modifier score for each term in all modifier networks, and rank them by this score. The smaller a term's score is, the more probable it is a non-constraint modifier.

VI. EXPERIMENTS

We present a comprehensive experimental study of the head, modifier, and constraint detection technique we introduced. We compare it with previous approaches and discuss applications such as sponsored search that benefit from the technique.

A. Mining Non-Constraint Modifiers

We mine non-constraint modifiers from the large concept space in Probase. From the 2.7 million concepts, we build 4,819 concept hierarchies, which are then converted to 4,819 modifier networks. We then calculate the logarithmic normalized betweenness centrality in each modifier network and aggregate them to get the score for each modifier. The top ranked modifiers are shown in Table I.

TABLE I. TOP RANKED MODIFIERS

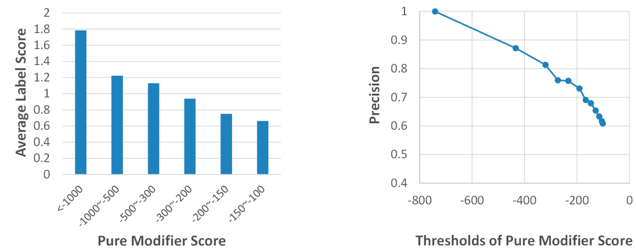
Rank	Modifier	Score	Rank	Modifier	Score
1	good	-4120.1	11	popular	-3607.97
2	traditional	-4037.08	12	conventional	-3586.87
3	common	-3995.69	13	standard	-3487.89
4	typical	-3794.6	14	local	-3287.59
5	great	-3779.18	15	regular	-3094.87
6	small	-3717.43	16	basic	-3046.72
7	large	-3664.69	17	big	-3041.43
8	modern	-3622.93	18	classic	-3013.82
9	simple	-3612.86	19	real	-2997.07
10	well-known	-3611.13	20	key	-2983.46

To measure the quality of our method and deterimine an optimal threshold for qualifying as a non-constraint modifier, we randomly select and manually label 300 terms with score 0, 1, or 2 using the criteria listed in Table II.

TABLE II. MANUAL LABEL CRITERIA

Score	Explanation	Examples
2	non-constraint modifier in any case	top, best
1	non-constraint modifier in some cases	electronic, thermally
0	in most cases, they are not negligible	South African, library

Then we bin the labeled terms in intervals of pure modifier scores (e.g. terms whose PMS is from -1000 and -500 are in one bin). For terms in each interval, we calculate their average labeled score (0$\sim$2). The result is shown in Fig. 7(a), where the x-axis represents the pure modifier score intervals, and the y-axis represents the average labeled score for terms in each interval. We also use different thresholds to test the precision of predicting on the test data. We assume that non-constraint modifiers with score 2 or 1 are correct, and score 0 wrong. The



(a) Manually Labeled Score vs. Modifier Score

(b) Precision vs. Modifier Score

Fig. 7. Performance of Modifiers Detection

results, shown in Fig. 7(b), are good for top ranked modifiers (x-axis represents the threshold we choose). We select a score cutoff with precision greater than 90%, which produce a total of 800 pure modifiers.

B. Mining Instance-Level Patterns

We mine instance-level head-modifier pairs from 6 month (2012/07-2012/12) worth of search log of BING (queries whose frequency $\leq$ 3 are filtered). We use the syntactic patterns described in Section III-B for mining. Take the 'A **for** B' pattern as an example. From 308,183,923 unique queries, we obtain 13,368,405 unique matches following the 'A **for** B' pattern. Among these matches, there are 3,892,152 unique queries whose A and B are in Probase. However, in many cases, the preposition **for** does not indicate head-modifier relationships. For example, patterns such as "* for sale," "search for *" appear frequently in queries, creating a lot of noise in the extracted head-modifier pairs. Although conceptualizing is able to filter out noise automatically, we perform some simple cleaning by removing patterns that are apparently unrelated (such as "* for sale") to save the cost of conceptualizing. Also, we remove non-constraint modifiers in the query. We finally obtain 3,336,475 unique queries for the 'A **for** B' pattern, and the total number of unique instance level head-modifier pairs is 14,144,235. Table III breaks down the number across different syntactic patterns. We can see that patterns with 'of', 'in', 'for' have similar proportion in all of the matches while patterns with 'with', 'on', 'at' account for a smaller proportion.

C. Building Concept Pattern Knowledgebases

Using the method described in Section III, we conceptualize instance pairs to concept level patterns, summing up the score for each concept pair and clustering similar concept pairs into concept patterns. Thus, we build a concept pattern knowledgebase for head-modifier relationships with each concept pattern comprising a number of concept pairs.

TABLE IV. CONCEPT-LEVEL HEAD-MODIFIER PATTERNS

# of instance level pairs	14,144,235
# of concept level pairs	84,207,802
# of concept level pairs after filtering	386,283
# of concept patterns after clustering	169,966

As shown in Table IV, from 14,144,235 instance level pairs we obtain 84 million concept level pairs (considering Probase can define as many as 7.29 trillion concept level pairs, this is

TABLE III. STATISTICS OF DATA SET

6 month query log (freq>3)	A **FOR** B	A **OF** B	A **WITH** B	A **IN** B	A **ON** B	A **AT** B
total # of matches	205,527,614	287,473,958	46,441,556	220,571,952	63,145,259	18,895,978
unique matches	13,368,405	14,189,485	3,420,457	17,276,873	5,032,200	1,458,932
filtered # of matches	120,345,688	211,852,494	29,865,611	151,359,122	40,286,256	11,978,341
filtered unique matches	3,336,475	3,398,661	1,031,810	4,627,184	1,539,772	508,981
unique heads	235,797	149,544	125,149	337,605	167,085	83,560
unique modifiers	253,919	327,001	121,499	166,086	143,703	61,341

a very small number). In some cases, both a concept pair and its reverse (head-modifier reversed) are included, but almost always at least one of them has very low score. In fact, in our system, we only keep concept pairs whose score is above a threshold of 3. This gives us 386,283 concept-level pairs, which are only 2.73% of the instance-level pairs mined from the log. Then, by leveraging the concept clustering results using k-Medoids [19] (available in the download link with Probase dataset), we further reduce to 169,966 pairs. Thus, we obtain a concise model for head-constraint relationships. Table V shown some top concept level patterns.

We can see concept patterns in many different domains in Table V, including “Game,” “Car,” “Health,” “Food,” etc. Previous work on understanding head-modifier relationships is domain specific. Our method, on the other hand, models head-modifier relationships for the open domain.

TABLE VI. TOP CONCEPT LEVEL PATTERNS

for		**of**		**with**	
home	city	map	city	movie	celebrity
recipe	dish	picture	celebrity	problem	vehicle
pet	state	university	city	people	disease
cheat	game	cast	movie	state	city
game	platform	song	film	interview	celebrity
coupon	store	player	team	container	lid
antibiotic	infection	diagnosis	illness	job	state
lyrics	song	skill	professional	sport	injury
code	symptom	episode	show	dessert	topping
shoe	women	symptom	disease	food	nutrient

Note: the first column is head, the second is modifier for each Prep.

Table VI shows concept patterns obtained from different syntactic patterns. We find that different syntactic patterns lead to concept patterns of different distribution. Concept patterns obtained from the ‘for’ syntactic pattern are most diverse (# of unique head concepts) and have the best coverage, and are more consistent with our manually labeled head and modifier datasets. Concept patterns from ‘of’ is less diverse, while patterns from ‘with’ contain valuable patterns missed by ‘for’. An interesting thing we can do with the patterns is that, given two terms, we can predict the preposition between them. In predicting, we are coming up with the missing semantics to make a meaningful phrase out of two terms.

D. Accuracy of Head-Modifier Detection

We measure the accuracy of head-modifier detection using labeled data. To avoid biases in labeling, we use the ’for’ syntactic pattern to generate the labeled data. We find queries matching the ‘for’ pattern in the search log from a separate 6 month interval (2013/01-2013/06), and label the instance before ‘for’ as the head and the one after ‘for’ as the modifier. We remove conflicting pairs in the labeled data (i.e., if ($A_{[head]}$, $B_{[modifier]}$) and ($B_{[head]}$, $A_{[modifier]}$) exist in the data, we remove both of them). This gives us a high quality, automatically labeled head/modifier pairs: *($A_{[head]}$, $B_{[modifier]}$)*. We then create two types of testing datasets from all queries (no matter it contains “for” or not): (1) for 2 components, the query in this dataset should contain A and B, where (A, B) is an entry in the labeled set; (2) for 3 components, the query in this dataset should contain A, B, and C, where (A, B) and (A, C) are two entries in the labeled set. Finally, we get 6 testing datasets for 2-component queries (from 6-month query log with frequency>5, and five monthly query log with frequency>5), and 1 testing dataset for 3-component queries (from 6-month query log with frequency>5).

Table VII shows the accuracy of our method on the 6 testing dataset. As we can see, our method achieves 90+% accuracy in all 6 data sets. Table VIII shows results of head-modifier dection on 2-component queries, and Table IX shows results on queries with 3 components. The accuracy of head detection in queries with 3 components is 92.4%.

TABLE VII. STATISTICS OF TESTING DATA SETS

Query Log frequency>5	# queries	# unique queries	Accuracy
6 month (2013/01-2013/06)	434,516,723	3,640,441	90.44%
1 month (2013/02)	36,373,640	408,270	91.90%
1 month (2013/03)	35,384,389	422,397	91.35%
1 month (2013/04)	28,457,571	373,530	91.93%
1 month (2013/05)	27,233,436	374,446	91.74%
1 month (2013/06)	31,013,819	403,792	91.42%

TABLE VIII. RESULTS OF 2-COMPONENT QUERY DETECTION

Query	Frequency
zip codes by *city*	379248
barbie **dress up games**	359766
jcpenney **coupons**	309890
dell **drivers**	227722
craigslist *ny*	218955
walmart **job application**	138745
seattle **weather**	112318
harry potter **reviews**	100833
lyrics to *songs*	58808
where is **notepad** in *windows 7*	63
blackberry **google calendar**	31
chords of a *guitar*	31
skinny jeans for older *women*	31

[a] **Head** is in bold; *modifier* is in italics and underlined.

We also verify one important point in head-modifier detection for short texts. Usually, in a phrase, the head appears as the last noun, and words appearing before the head is its constraint. However, this rule does not work for short texts such as search queries that don’t observe grammars of natural language. Using the 2 components queries in the 6 month testing data set, we calculate how many times the head appears last after the constraint. We find that head appears last in only 41.45% unique queries.

TABLE V. EXAMPLES OF CONCEPT PATTERN

Index	Cluster Size	Concept Pattern[a] (head,modifier)	Examples of Concept Pairs
1	615	pet, state	dog, state;pet, southern state;pet, team
2	192	home, city	home, city;home, town;home, place
3	143	cheat, game	cheat, title;cheat, video game;cheat, online game
4	124	weather, city	weather, county;weather, urban area;weather, town;
5	110	recipe, dish	recipe, food;recipe, appetizer;recipe, favorite
6	89	coupon, store	coupon, store;coupon, retailer;coupon, business
7	136	antibiotic, infection	drug, infection;antibiotic, illness;antibiotic, virus
8	296	game, platform	game, computer;video game, platform;game, console game pad
9	100	treatment, disease	treatment, disease;treatment, autoimmune disease; treatment, medical problem
10	153	accessory, vehicle	accessory, car;pump, vehicle;optional attachment, truck

[a] we choose the concept pair with the largest score as the representative of the concept pattern. Cluster size is the number of concept pairs in each concept pattern.

TABLE IX. RESULTS OF 3-COMPONENT QUERY DETECTION

Query	Frequency
6th grade *math* **worksheets**	856
dress up games for *girls* and *kids*	740
kids *cooking* **games**	1565
business **grants** for *women*	364
quotes about *life* and *love*	347
free **music** your *ipod* *itunes*	241
women seeking *men* **marriage**	154
zucchini **recipes** *bread*	95
adobe flash player *64 bit* *windows 7*	65
seattle **jobs** *craigslist*	11
cheats for *borderlands* *xbox 360*	11

[a] **Head** is in bold; *modifier* is in italics and underlined.

E. A Comparison with Other Methods

We compare the performance of our method with an existing approach [7], and 3 alternative approaches based on our approach, which we describe briefly below.

*Entity-oriented modifier detection (**EOMT**):* In this alternative approach, we create a dictionary with two columns: *(entity, score)*. The score is computed as a frequency difference, that is, we calculate how often the entity serves as a head, and how often as a modifier, and the score is the difference in the frequency. We consider an entity serves as a head when it appears before prepositions, and as a modifier when it appears after prepositions in search logs. We rank entities by their scores and make sure the entity dictionary is of roughly the same size as the concept pattern knowledgebase. For a 2-component query, we check each component to see whether it is a head (score>0) or a modifier (score<0). If the scores of both components are greater than 0, less than 0, or equal to 0, we cannot decide which is head and which is modifier in the query, and we classify these queries as "Not Identifiable."

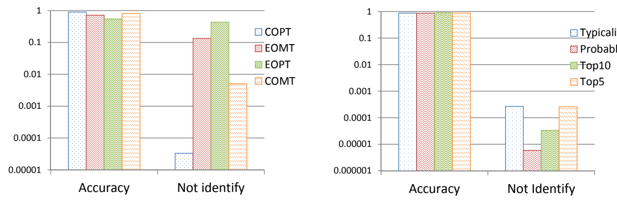
*Entity-oriented pattern detection (**EOPT**):* In this alternative approach, we create a dictionary of three columns: *(*$entity_{[head]}$*,* $entity_{[modifier]}$*, score)*. The score is calculated from the frequency difference between 'A for B' and 'B for A'. Similarly, we still keep the same size of this dictionary as other dictionaries' sizes to make a fair comparison. For a new query, we check whether it has matched pair in the entity pair dictionary. If not, we classify the query as "Not Identifiable."

*Concept-oriented modifier detection (**COMT**):* In this alternative approach, we create a dictionary with only the modifier concepts and their scores, with two columns: *(concept, score)*. For a given concept, first we collect its entities which serve as heads in the training data set, and then get their conceptualization scores CS_i^H by Equation (4). Similarly, we can get CS_j^M. The score of this concept in the dictionary will be $\sum CS_i^M - \sum CS_j^H$. For a new query, we conceptualize each entity as Equation (4) and get the modifier score of each entity by $\sum_c CS(e,c) \cdot Score(c)$. The entity with higher modifier score is the modifier. If the score of both entities are the same (both 0), the query is classified as "Not Identifiable."

We evaluate all of these methods and our method (which we call ***COPT*** or *concept-oriented pattern detection*) using the same testing data set. As Fig. 8 shows, the accuracy of entity-based method is much lower than concept-based method (EOMT is about 70% and EOPT is only 50%∼60%). At the same time, they cannot identify head and modifiers in a much larger number of queries than the concept-based methods. Non-identifiable rate of EOMT is about 10%, EOPT is 40% while COPT is near 0 ("Not identifiable" in COPT means that all the concept pairs formed by the two components are not in the concept pattern dictionary). For the two concept-based methods, the accuracy of COPT is about 90%, while the accuracy of COMT is about 85%. The non-identifiable rates are both close to 0, indicating that concept-based method achieves much better coverage.

We also conduct a 5-fold cross validation of our method on a 6 month test data set. We split the query log into 5 parts, generate the concept pattern dictionary on 4 parts and test our method on the remaining queries. The accuracy is about 87% and the non-identifiable rate is near 0. For comparison, we do 5-fold cross validation for EOMT, and the accuracy is about 64% and the non-identifiable rate is about 33%. These results show that our method has better extendibility to unknown queries than entity-based method.

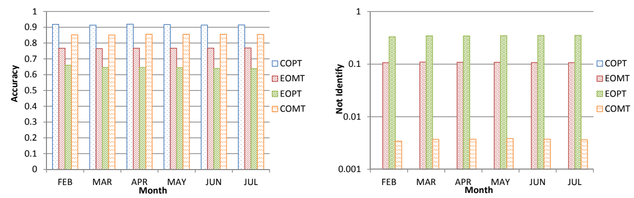
Comparing the results of different methods by month, neither the accuracy nor the not-identifiable-rate changes much (Fig. 9). Comparing the pattern-based methods (including entity-oriented pattern-based EOPT and concept-oriented pattern-based COPT) and non-pattern-based methods (EOMT and COMT), we find that the pattern-based methods resulted in fewer errors. The error rate of EOPT is near zero since it directly stores the head and modifier instance pairs. Besides, the error rate of COPT (pattern-based) is about 10% lower than that of COMT (non-pattern-based). This also indicates the advantage of mining patterns. The essence is that a term



(a) Different Methods on 6 month query log

(b) Different Parameters on 6 month query log

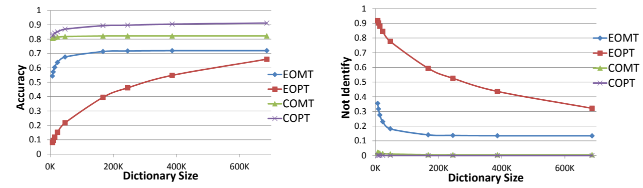
Fig. 8. Comparison of Methods and Parameters



(a) Accuracy on each month query log

(b) Not Identifiable rate on each month query log

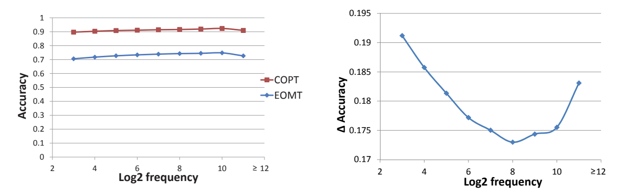
Fig. 9. Test on each Month Query Log



(a) Accuracy vs. Dictionary Size

(b) Not Identifiable rate vs. Dictionary Size

Fig. 10. Accuracy and Identifiability on Different Dictionary Size



(a) Accuracy vs. Query Frequency

(b) Accuracy difference vs. Query Frequency

Fig. 11. The variation of Accuracy against Query Frequency

(or concept) can be a head or a modifier in different context, so we can decide whether it is a head or a modifier only when its context is given. For example, When the concept "game" accompanies "phone, platform, technology," it is the head. E.g. in queries like "*angry birds* for windows phone 7" and "*juice defender* for android," "*angry birds*"(game) and '*juice defender*"(game) are the heads. While in queries such as "angry birds *walkthrough*," "game" is a modifier while "walkthrough" is the head. Also, in queries like "*zombie mod* for minecraft," "*deadly boss mods* for wow," "game" is a modifier concept while "mod" is the head concept.

Comparison with an existing method: We implemented the method of term weighting [7]. The features used in that work include features of uni- or the N-gram count (Google uni/bi-gram); how many times the term appears as a query and within a query (1-month query log); how many times the term appears as a Wikipedia Title or within a Wikipedia Title. They also consider the ratio of bi-gram and the product of its two unigrams for each feature (e.g. $\frac{s(q_i,q_j)}{s(q_i)\cdot s(q_j)}$). They use a linear combination of these features to assign the weight to each term in the query.

After collecting the features as they do, we use SVM to classify the head and the modifier. We use 'A for B' queries in 6 month query log as the training set as our approach and another 6-month query log for test as before. We compute the feature vectors for each uni-/bi- grams in head and constrain entity, and add up these vectors for head ($\vec{h}$) and modifier ($\vec{m}$) respectively. We use libSVM package [20] with linear kernel to train and classify. However, the performance of this approach is poor and the accuracy is about 55%. The reasons may be the features they chose don't have direct relationship with our head-modifier relationship. By this experiment, we can also conclude that traditional term weighting approach can't resolve the head-modifier detection problem.

F. Impact of Scoring Functions and Parameters

We evaluate the influence of dictionary sizes (e.g. the number of concept pairs) on accuracy. The results are shown in Fig. 10. For all of the four methods, it shows that the accuracy increases and the "not identifiable" rate decreases when the dictionary becomes larger, but different methods reach saturation at different dictionary size. EOPT is not saturated even at 0.7M while COMT saturates at about 0.1M. It is not surprising that the coverage of entity-based method is much less than the concept-based method. On the other hand, non-pattern-based dictionary contains less diverse entities or concepts than the pattern-based dictionary. For our method, when the dictionary size reach 0.3M, the accuracy tends to be saturated. It means that about 0.3M concept pairs can cover most entity pairs with modifying relationship in queries.

Then, we test whether our conceptualization score function is suitable. We do the experiments on 6 month test data set using $P(c|e)$ ("Probability") and $P(e|c)$ ("Typicality") as conceptualization score functions separately. The accuracy using the former is 86% and the latter one is 87%. Both are less than our method ("Top10" in Fig. 8(b)). On the other hand, the "not identify" rate of "Probability" is less than that of "Typicality" while our method ("Top10") is in the middle (in Fig. 8(b)). As described above, using $P(c|e)$ as the conceptualization score function often results in general concepts while $P(e|c)$ results in more specific concepts. Thus, the coverage of the former is larger. Our score function is better than both methods, since our score function takes into account both typicality and generality of a concept, and chooses the most suitable concept for each entity. Our concept pattern better summarizes the modifying relationship between entities, better clusters similar entities together. For a new query, it can easily refer to the correct concept pattern it belongs to. We also test the performance of mapping to top 5 concepts for each entity instead of top 10. The accuracy is 87.5% and the "not identifiable" rate is larger than that of "Top 10" (Fig. 8(b)). Thus, mapping to 10 concepts achieves better coverage and accuracy.

G. Performance on Tail Queries

Furthermore, we test whether the performance of our method deteriorates for tail queries. We divide queries into

TABLE X. EXAMPLES OF QUERY AND BID KEYWORD PAIRS

Query	Query Substitute	Bid Keywords
benny hill wiki	benny hill	benny hill videos
all Samsung Galaxy phone cases	phone cases	case Samsung Galaxy; phone covers Samsung Galaxy
appropriate preteen girl books	preteen books	books preteens;teen book;
Asturias guitar music	guitar music	classical guitar lessons; free guitar music
best ion ceramic hair dryer	hair dryer	best hair dryer; ionic hair dryer
BA 194 flight status	flight status	airline status; jet airways flight status
berkshire natural resources	natural resources	list natural resources
any good not inappropriate jokes	joke	clean jokes; good jokes tell
are benefits exercise concerning heart health	benefits exercise	exercise tips;10 benefits exercise;

different bins by their logarithmic frequencies. The average accuracy of each bin is shown in Fig. 11(a). It is shown that for tail queries, the accuracy of our COPT method does not suffer much while the accuracy of entity-based method (e.g. EOMT) suffers to some extent. To show more clearly, we compute the accuracy difference between these two methods in each frequency bin, shown in Fig. 11(b). The accuracy difference is large for tail queries and ultra-high frequency queries. For ultra-high frequency queries, the terms can appear both as heads or modifiers frequently, so a dictionary without pair information is not so accurate as the pattern dictionary. On the other hand, the entity-based dictionary may not contain entities in tail queries. Even if it contains these entities, it may not be correctly labeled as they are poorly supported. While for concept-based method, a concept pair is supported by lots of entities including both frequent queries and tail queries, so it will not suffer much from the noise in tail queries. Tail queries, though may be missed in the training data set, can find matched concept patterns in the dictionary as discovered by other queries. Thus, concept-based method performs much better than entity-based method especially for tail queries.

We manually label 200 tail queries and score the identified head or intent by three grades: 0 means that the intent detected by our method is wrong; 1 means that the detected intent is fair; 2 means that the detected intent is good. There are 133 queries labeled '2' and 50 queries labeled '1' while only 17 queries labeled '0'. Queries with label '1' often have general heads, for example, queries "police hat template" and "propane prices per gallon oklahoma" returns "template" and "price" as head, which can be viewed as heads to some extent but are too general to retrieve useful information. The manual labeling results are consistent with the results on our automatically generated test data sets.

H. Application on Sponsored Search

It is a challenging task to find matching ads for tail queries, which do not have click through data nor enough context to suggest the best match. We use our method to calculate a similarity score between a tail query and an ad bid keyword. We first remove the non-constraint modifiers from both queries and ads bid keywords. We then identify the head and modifier components, and generate their concept representation (a concept vector for the head and a concept vector for the modifier). We put more weight on the concept representation of the head, and less on that of the modifier. Then, we use cosine similarity of the weighted vectors to measure the similarity between queries and ad keywords. We randomly select and manually label 100 tail query and bid keyword matching pairs: if the query and bid keywords are matched, the pair is labeled '2'; if the query and bid keywords share the same main component but with different non-negligible constraints, the pair is labeled '1'; pairs with no matching components are labeled '0'. There are 75 pairs labeled '2' and 11 queries labeled '1' while 14 queries labeled '0'. Some examples of matched query and bid keyword pairs are shown in Table X. This mechanism we developed has been used in production.

VII. RELATED WORK

Most query intent detection methods are based on query topic classification [8], [9], [10]. The task of KDD Cup 2005 was to classify queries into 67 categories [21]. These methods typically do not have good coverage, as they are constrained by existing taxonomies. Another problem is that the taxonomies usually do not have appropriate granularity for intent detection. For example, "job search" and "job interview" have different intent but both are classified into the category of "job."

Bendersky et al [7] worked on the problem of assigning different weights to different terms in a query. Query rewriting is very relevant to our work on query intent detection. Kumaran et al [22] turned long queries into short ones by dropping less significant terms in the query. The rationale of reducing long queries to equivalent short queries is the following. A short query is usually less ambiguous and more frequent, and web search or sponsored search is good at handling such queries. On the other hand, a long query is usually rare, more likely to contain misleading terms, and is more difficult for web search. Both of these two methods defined some features to weight the terms or rank the sub-queries based on the terms statistics in collections. In the former one, the authors defined query terms and bi-grams as concepts and gathered the concept frequency in documents, Wikipedia title, and Google n-grams as features. They used the linear combination of these features as weight or importance of each concept in the query. Then, they built a weighted dependence model based on the concept weights for information retrieval. However, in their work, "concepts" are just terms in the query. In the latter paper, the authors implemented several predictors for query quality, such as mutual information between two terms, query clarity (i.e. the KL divergence of query model and collection model), and so on. Then, the authors used these features to train a classifier by RankSVM and learn a ranking function for sub-queries. These two methods learned the weight of terms in the query by term statistics features and need large amount of corpus and labeling data. However, these features are not related to the meaning of the term directly and thus it is hard to explain how these features decide the head-modifier relationship. Instead, our method uses semantic features (concept patterns) directly. Our features are interpretable as they explicitly reflect the head-modifier relationships.

Instead of using term statistics, recent works derive query intent by fitting queries into templates [11], [23], [12]. Li et al. [11] used semantic and syntactic features to decompose queries into intent head and intent modifier. They considered attribute names as heads of the intent and attribute values as values of the intent. However, they needed a head and modifier lexicon and knowledge about attributes and their values for a specific domain. Cheung et al. [23] clustered queries and constructed patterns for each domain, aiming at domain-dependent structured search. Chang et al. [12] developed a sophisticated probabilistic inferencing framework based on both forward and backward random walks to construct query templates in each domain. Although all of these work and our method are to find the relationship between terms in the query, they focused on queries that fit certain templates in a specific domain. Instead, our work aims at finding general head-modifier relationships, dealing with all queries without assuming underlying common structures, and are not confined to a specific domain.

There are some existing works [13], [14] on attribute extraction that takes advantage of syntactic patterns with prepositions such as 'for', 'of', etc. Certainly, attributes can be used to define head-modifier relationships. But head-modifier relationships are not confined by entity-attribute relationships. For example, in "movie *review*,""*side effect* for drug," reviews and side effects are not really attributes for movies and drugs. Head-modifier relationships are more general, such as in "*game* for girls," "*accessory* for vehicle." In our work, we model head-modifier relationships as relationships between two concepts. There is much work on mining all instance pairs with certain relationships. For example, Agichtein et al [15] fould templates such as *ORGANIZATION* 's headquarters in *LOCATION* for a specific relationship. These templates are generated by bootstrapping from seed instance pairs. One big difference between such work and our work is that we are modeling relationships at the concept level.

VIII. Conclusion

Short text understanding is an important and challenging task to a large variety of applications including search relevance, ads keyword selection, query expansion/reformulation/classification, etc. One essential part of understanding the intent in a short text is to correctly identify the head and the modifiers in the short text. In this paper, we introduce a semantic approach for head, modifier, and constraint detection. We use an unsupervised learning approach to obtain a large amount of instance level head-modifier relationships, then we "lift" them into the concept level to derive a general and concise model for head-modifier relationships in all domains. Extensive results have shown that our method achieves good performance in identifying the head and modifiers in short text. The technique described in this paper can directly benefit semantic similarity calculation between two short texts as it is able to assign different weights to different components in the short texts based on their head and modifier status.

There is also much future work. For non-constraint modifiers, we leverage the concept space of Probase in this paper. However, this may also bring in bias from Probase. A more sophisticated approach for non-constraint modifier mining and usage (instead of simply ignoring non-constraint modifiers) is an interesting research topic. Besides, how to smartly handle conflicting instance pairs and concept patterns is also important for our framework. Last but not least, recognizing those unseen entities and mapping them to appropriate concept patterns can further improve the coverage and performance of our approach.

Acknowledgments

This work was partially supported by the National Key Basic Research Program (973 Program) of China under grant No. 2014CB340403, the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China.

References

[1] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[2] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Prentice Hall PTR, 2000.

[3] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *SIGMOD*, 2008.

[4] T. Lee, Z. Wang, H. Wang, and S. Hwang, "Web scale taxonomy cleansing," in *VLDB*, 2011.

[5] W. Wu, H. Li, H. Wang, and K. Q. Zhu, "Probase: a probabilistic taxonomy for text understanding," in *SIGMOD*, 2012.

[6] Y. Song, H. Wang, Z. Wang, H. Li, and W. Chen, "Short text conceptualization using a probabilistic knowledgebase," in *IJCAI*, 2011.

[7] M. Bendersky, D. Metzler, and W. B. Croft, "Learning concept importance using a weighted dependence model," in *WSDM '10*, 2010.

[8] D. Shen, J.-T. Sun, Q. Yang, and Z. Chen, "Building bridges for web query classification," in *SIGIR*, 2006.

[9] X. Li, Y.-Y. Wang, and A. Acero, "Learning query intent from regularized click graphs," in *SIGIR*, 2008.

[10] J. Hu, G. Wang, F. Lochovsky, J.-t. Sun, and Z. Chen, "Understanding user's query intent with wikipedia," in *WWW*, 2009.

[11] X. Li, "Understanding the semantic structure of noun phrase queries," in *ACL*, 2010, pp. 1337–1345.

[12] G. Agarwal, G. Kabra, and K. C.-C. Chang, "Towards rich query interpretation: walking back and forth for mining query templates," in *WWW*, 2010.

[13] M. Pasca and B. Van Durme, "What you seek is what you get: extraction of class attributes from query logs," in *IJCAI*, 2007, pp. 2832–2837.

[14] M. Paşca and B. Van Durme, "Weakly-supervised acquisition of open-domain classes and class attributes from web documents and query logs," in *Proceedings of ACL-08: HLT*.

[15] E. Agichtein and L. Gravano, "Snowball: extracting relations from large plain-text collections," in *DL*, 2000.

[16] A. Hippisley, D. Cheng, and K. Ahmad, "The head-modifier principle and multilingual term extraction," *Nat. Lang. Eng.*, vol. 11, no. 2, Jun. 2005.

[17] S. Soderland, D. Fisher, J. Aseltine, and W. Lehnert, "Crystal: Inducing a conceptual dictionary," in *IJCAI*, 1995.

[18] D. Kim, H. Wang, and A. Oh, "Context-dependent conceptualization," in *IJCAI*, 2013.

[19] P. Li, H. Wang, K. Zhu, Z. Wang, and X. Wu, "Computing term similarity by large probabilistic isa knowledge," in *CIKM*, 2013.

[20] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, May 2011.

[21] Y. Li, Z. Zheng, and H. K. Dai, "Kdd cup-2005 report: facing a great challenge," *SIGKDD Explor. Newsl.*, vol. 7, no. 2, Dec. 2005.

[22] G. Kumaran and V. R. Carvalho, "Reducing long queries using query quality predictors," in *SIGIR*, 2009.

[23] J. C. K. Cheung and X. Li, "Sequence clustering and labeling for unsupervised query intent discovery," in *WSDM*, 2012.