Unsupervised Head-Modifier Detection in Search Queries

Zhongyuan Wang, Renmin University of China, Microsoft Research
Fang Wang, State Key Laboratory of Software Development Environment, Beihang University
Haixun Wang, Facebook Inc.
Zhirui Hu, Harvard University
Jun Yan, Microsoft Research
Fangtao Li, Google Research
Ji-Rong Wen, Beijing Key Laboratory of Big Data Management and Analysis Methods, Renmin University of China.
Zhoujun Li, State Key Laboratory of Software Development Environment, Beihang University

Interpreting the user intent in search queries is a key task in query understanding. Query intent classification has been widely studied. In this paper, we go one step further to understand the query from the view of head-modifier analysis. For example, given the query "popular iphone 5 smart cover," instead of using coarse-grained semantic classes (e.g., *find electronic product*), we interpret that "smart cover" is the head or the intent of the query and "iphone 5" is its modifier. Query head-modifier detection can help search engines to obtain particularly relevant content, which is also important for applications such as ads matching and query recommendation. We introduce an unsupervised semantic approach for query head-modifier detection. First, we mine 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-modifier detection. The developed mechanism has been used in production for search relevance and ads matching. We use extensive experiment results to demonstrate the effectiveness of our approach.

CCS Concepts: • **Information systems → Web log analysis; Query intent;**

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Query Intent, Head and Modifier, Concept Pattern, Knowledge Modeling

ACM Reference Format:
Zhongyuan Wang, Fang Wang, Haixun Wang, Zhirui Hu, Jun Yan, Fangtao Li, Ji-Rong Wen, and Zhoujun Li, 2016. Unsupervised Head-Modifier Detection in Search Queries. *ACM Trans. Knowl. Discov. Data.* 9, 4, Article 39 (March 2010), 26 pages.
DOI: 0000001.0000001

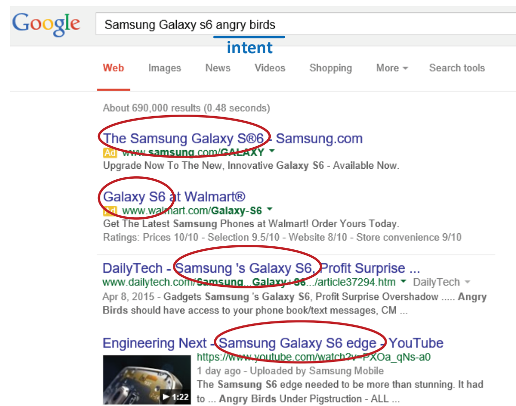
1. INTRODUCTION

Understanding a user's intent or information need that underlies a query has long been recognized as a crucial part of effective information retrieval [Li et al. 2011]. This is because search queries are usually short and do not observe the grammar of a written language. For example, to find out "where to buy the popular smart cover for iphone 5," a user may simply search "popular smart cover iphone 5." Most previous work in this area focuses on query intent classification [Shen et al. 2006; Li et al. 2008; Hu et al. 2009], which aims at interpreting queries in terms of predefined semantic intent classes. Indeed, the intent is crucial in determining whether a query can be answered by certain data

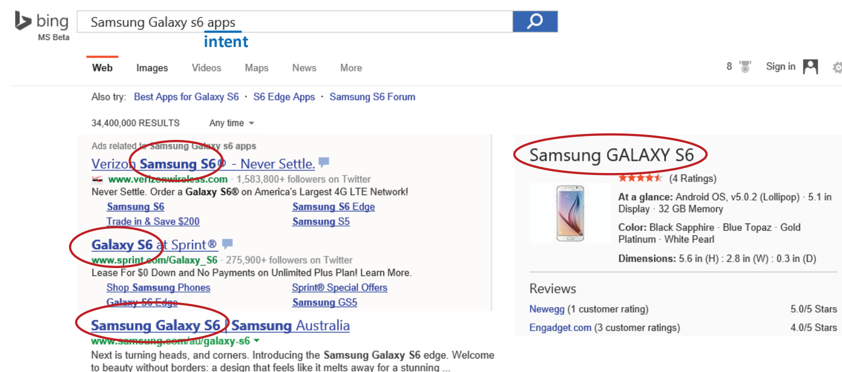
A preliminary version of this paper has been accepted for publication in the Proceeding of The IEEE International Conference on Data Engineering (ICDE'14), pp. 280-291, 2014.
Author's addresses: Z. Wang, Renmin University of China and Microsoft Research Asia; F. Wang and Z. Li, State Key Laboratory of Software Development Environment, Beihang University; H. Wang, Facebook Inc.; Z. Hu, Harvard University; J. Yan, Microsoft Research Asia; F. Li, Google Research; J. Wen, Beijing Key Laboratory of Big Data Management and Analysis Methods, Renmin University of China.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. 1556-4681/2015/-ART0 $15.00
DOI: 0000001.0000001

sources. For example, the query “smart cover iphone 5” should be determined to contain product intent, thereby triggering product search would return to users the relevant and essential results; whereas “Seattle hotel jobs” should not. However, assume we can get the semantic intent classes (e.g. *find electronic product*), search engines may still fail to understand that the user wants “smart cover,” rather than “iphone 5.” This is because: (1) current search engines are still keyword-based; (2) there may be not enough click-through signals indicating users’ real intent for this kind of queries.

Fig. 1(a) and Fig. 1(b) give two real examples in current search engines. Both Google and Bing do not understand that users are looking for angry birds and apps for Samsung Galaxy S6, instead of the smartphone itself.



(a) Search “Samsung Galaxy s6 angry birds” in Google

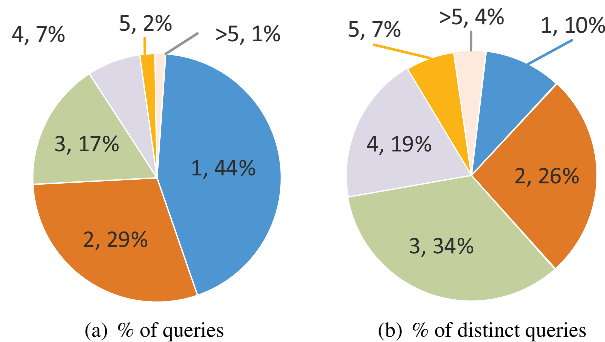


(b) Search “Samsung Galaxy s6 apps” in Bing

Fig. 1: Examples: search engines fail to understand users’ intent in semantics

In this work, we go one step further to detect the head-modifier structure in search queries. A query contains head and modifier components, where head represents the intent, and modifier limits the scope of the intent. Take the search query “popular iphone 5 smart cover” as an example. The query consists of three components: “popular,” “iphone 5,” 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 5,” “popular” modifier components. However, not all modifiers are equal. Compared with “popular,” which is more subjective, “iphone 5” 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 5” will introduce many irrelevant results. In this paper, we call modifiers such as “iphone 5” *constraint modifiers*, and modifiers such as “popular” *non-constraint modifiers* or *pure modifiers*.



(a) % of queries (b) % of distinct queries

Fig. 2: Components in Search Queries

Typically, a query contains at least one head, zero or more modifiers. We analyzed 1 week worth of search log (from 07/25/2012 to 07/31/2012) of BING, and used Freebase [Bollacker et al. 2008; Lee et al. 2011] and Probase [Wu et al. 2012; Song et al. 2011] as vocabularies to identify components in each query. As Fig. 2 shows, about 56% of queries have 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 and modifier is critical in understanding the search query. Queries may contain multiple heads. Generally, multiple heads in a query belong to the same category. For example, in the query “iphone 5s vs galaxy s5,” there are two heads: “iphone 5s” and “galaxy s5,” and both of them belong to the same category *smartphone*. Thus the case of multiple heads can be easily detected. In this paper, we mainly focus on single head detection. We use examples that contain one head and one modifier in our discussion.

We focus on noun phrase queries. Head and modifiers in noun phrases have long been found useful for information retrieval [Evans and Zhai 1996]. Identifying the head and modifiers from queries can also be beneficial to many IR applications [Evans and Zhai 1996; Li 2010]. For example, knowing the semantic role of each query component, we can reformulate the query into a structured form or re-weight different query components for structured data retrieval [Robertson et al. 2004; Kim et al. 2009; Paparizos et al. 2009]. Also, the knowledge of heads and modifiers can be used for short text matching applications such as advertisement matching in search engines.

However, head-modifier detection in queries is a challenging task, because search queries usually do not observe the grammar of a written language. Hence, simple linguistic methods for the detection may not be applicable. For example, a simple linguistic rule says in a noun phrase, the last noun is the head. However, for “popular smart cover iphone 5,” this is not true. Bendersky et al [Bendersky et al. 2010] develop a statistical approach to weight the terms in a query. But it needs a large labeled corpus. Besides, it is more concerned with weighting query terms rather than detecting the head and modifier for the query. Some other work tries to infer the intent of the input by fitting it into templates that are common in a specific domain [Li 2010; Agarwal et al. 2010]. There is also some work mining entity-attribute relationships but not specifically head-modifier relationships [Pasca and Van Durme 2007; Paşca and Van Durme 2008; Agichtein and Gravano 2000] and their performance depends on the seed entity-attribute pairs chosen for each domain.

In contrast, human beings are good at deriving meaning from noisy, ambiguous, and sparse input. We understand queries by leveraging the knowledge in our mind which enriches the input to produce meaning. For example, given the query "popular smart cover iphone 5," we know that "smart cover" is an *accessary* and there is a kind of "smart cover" designed for "iphone 5." Based on the above knowledge, we infer that "smart cover" is the query head, which is modified by "iphone 5." From this view, in order for machines to understand user queries, we need supply such knowledge to machines so that the gap between input and understanding can be bridged. Specifically, the open domain knowledge we need beyond the input includes:

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

Our approach of query head-modifier detection is to derive head-modifier patterns at the concept level, from a large number of instance level head-modifier pairs. The concept level head-modifier patterns are in the following form: $(concept_{[h]}, concept_{[m]}, score)$. One example might be $(accessary_{[h]}, device_{[m]}, 0.9)$, which indicates that when an *accessary* and a *device* appear together in a query, 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 knowledge base 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. The most important one is that conflicts may exist between the concept patterns, that is, there might be a pattern that says *device* is a head, *accessory* is a modifier, and another pattern that says the opposite. There are two possible causes, as shown in Tab. I. Conflict instance pairs can directly lead to conflict concept patterns. For example, "camera" is the head and "laptop" is the modifier in "camera *for* laptop," but in "laptop *with* camera," they are just the opposite. They will lead to conflict concept patterns such as $(accessary_{[h]}, device_{[m]})$ and $(device_{[h]}, accessary_{[m]})$. Besides, conflict can also be introduced by generalizing (also known as Conceptualization [Song et al. 2011]) the instance head-modifier pairs to the concept level, since different instances may belong to the same concepts. For instance, "cover" is the head in "cover *for* ipad" but shares the same concept *accessary* with the modifier "camera" in "laptop *with* camera," and the corresponding modifier "ipad" and head "laptop" also belong to the same concept *device*. This also leads to conflict concept patterns $(accessary_{[h]}, device_{[m]})$ and $(device_{[h]}, accessary_{[m]})$. We need to design a sophisticated detecting process to handle conflict patterns. Second, the knowledge we acquire must have enough coverage so that we can handle all possible input. Third, as we mentioned, we differentiate constraint modifier from non-constraint modifiers. The intuition is that non-constraint 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 knowledge base and use betweenness centrality to mine non-constraint modifiers.

Table I: Examples of the two conflict reasons

Conflict Type	Query	Concept pattern
conflict instance pairs	camera *for* laptop laptop *with* camera	$(accessary_{[h]}, device_{[m]})$ $(device_{[h]}, accessary_{[m]})$
conflict in conceptualization	cover *for* ipad laptop *with* camera	$(accessary_{[h]}, device_{[m]})$ $(device_{[h]}, accessary_{[m]})$

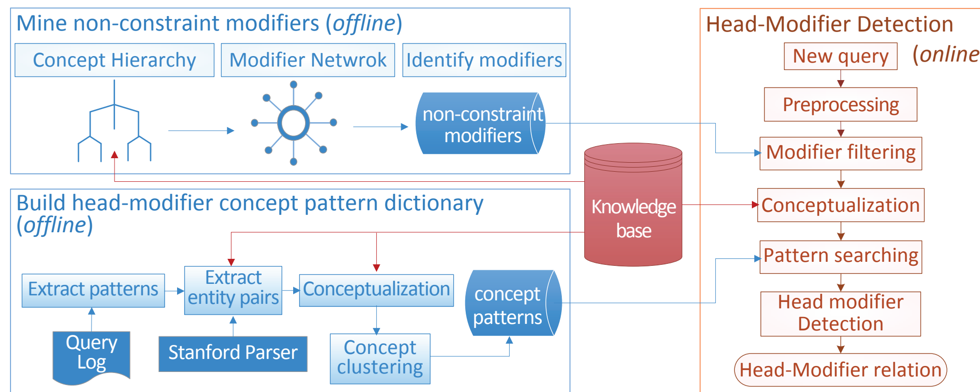


Fig. 3: Framework for head-modifier detection in search queries

Our contributions

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 detection.* In comparison, existing work usually requires labeled data and is domain specific.
— *We build a concept pattern knowledge base to model head-modifier relationship at the concept level.* We "lift" head-modifier relationships at the instance level to the concept level. We analyze the causes of conflict patterns and propose to model the head-modifier relationship in a probabilistic way. The knowledge base is small but has strong generalization power.
— *We design a sophisticated detecting approach for head-modifier detection in search queries.* For two-component queries, we detect the head-modifier relation by aggregating the supporting evidence from concept patterns. For multi-component queries, a combined detection method is proposed by leveraging the concept pattern knowledge and the statistical information learnt from search log. Extensive results show that the proposed approach achieves good performance (90% $accuracy$) in head-modifier detection.

Paper organization

The rest of the paper is organized as follows. Section 2 describes an overall framework and the taxonomy we use. Section 3 finds non-constraint modifiers. Section 4 derives concept level head-modifier patterns. Section 5 conducts our query head modifier detection. Section 6 gives experiment results and compares our approach with other methods. Section 7 introduces related work. Section 8 concludes our work.

2. OVERVIEW

In this section, we describe the framework of our approach, and the taxonomy we used in the framework.

2.1. Framework

Fig. 3 depicts the framework we use for head-modifier 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 query head-modifier detection using the knowledge acquired offline.

As we mentioned, we classify modifiers into two categories: constraint modifiers and non-constraint modifiers. To find terms that are often used as non-constraint modifiers, we construct

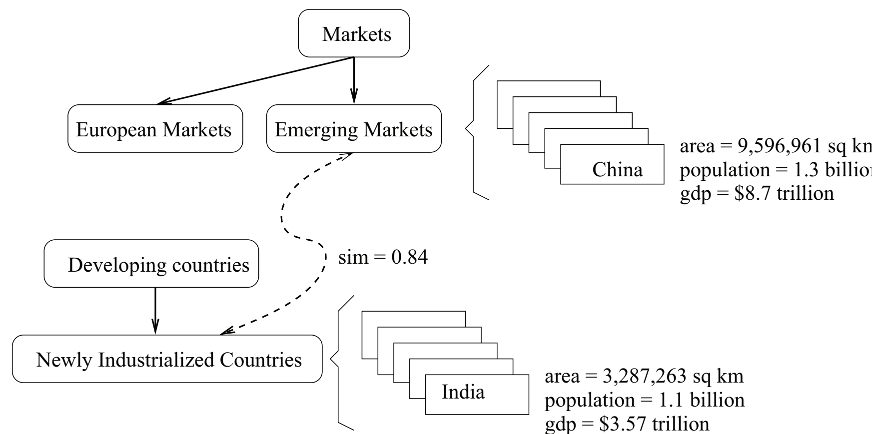


Fig. 4: A Snippet of Probases core taxonomy

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") or modifiers (e.g., "large" and "developed"), and edges denote modifying relationships. We show that non-constraint modifiers can be detected using a measure known as betweenness centrality in graph analysis. We give more details in Section 3.

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"$_{[h]}$, "mac"$_{[m]}$). We then conceptualize them into the concept level, which gives us head-modifier patterns such as $(game_{[h]}, computer_{[m]})$. This process involves how to handle the conflict pairs. We describe this process in detail in Section 4.

Given a new query, we perform online head-modifier detection using the acquired knowledge. We first check if there is any preposition in the query, since preposition could tell us the head-modifier structure directly. For example, given "smart cover *for* ipad", it is clear that "smart cover" is the head. When there is no preposition in the query, we first identify and remove non-constraint modifiers. Then, we form candidate head-modifier pairs. Finally, we match the candidates against the concept head-modifier patterns through conceptualization and detect the head-modifier by leveraging the acquired concept patterns. We describe the above process in detail in Section 5.

2.2. Probase: A large scale IsA taxonomy

We need a taxonomy knowledgebase to "lift" the instance level head-modifier relationships to concept level. In this paper, we use an isA taxonomy known as Probase[1] [Wu et al. 2012] to do this work.

Probase is a large network of multi-word terms. Figure 4 is a snippet of Probase, which consists of concepts (e.g. emerging markets), instances (e.g., China), attributes and values (e.g., China's population is 1.3 billion), and relationships (e.g., emerging markets, as a concept, is closely related to newly industrialized countries). It provides a huge concept space that covers all concepts of worldly facts. The version of Probase we use contains 2.7 M concepts and 40 M entities. We use a mechanism to efficiently recognize Probase concepts and entities in a query. We omit the discussion here due to lack of space.

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

2.2.1. Typicality. Another feature of Probase is that it is probabilistic, which means every claim in Probase is associated with some probabilities that model the claims correctness, typicality, ambiguity, and other characteristics. Typicality measures the probabilities between concepts and entities. Probase contains the following probabilities for it:

— $P(c = company|e = apple)$: How likely people will think of the concept “company” when they see the entity “apple.”
— $P(e = stevejobs|c = ceo)$: How likely “steve jobs” comes into mind when people think about the concept “ceo.”

The probabilities are derived from evidences found in web data, search log data, and other existing taxonomies. The typicality between e and c, where e is an entity or a subconcept of concept c, is weighted as follows:

$$P(e|c) = \frac{n(e,c)}{n(c)}, \quad P(c|e) = \frac{n(e,c)}{n(e)} \tag{1}$$

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. 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 $P(poodle|dog) > P(pug|dog)$.

2.2.2. Concept Cluster. Among the large amount of concepts in Probase, many concepts are similar to each other, such as “country” and “nation,” “music star” and “pop star,” etc. We use Concept Clusters to gather similar concepts together, by using a k-Medoids clustering algorithm proposed by Li et al. [Li et al. 2013]. One concept cluster can represent a general topic domain, recognized with its center concept. For example, for the cluster centered around country, most of its members are highly related to country, such as nation, asian country, developing country, region etc. In this paper, we use the concept cluster in multiple-ways including sense detection in non-constraint modifier mining and concept pattern mining.

3. MINING NON-CONSTRAINT MODIFIERS

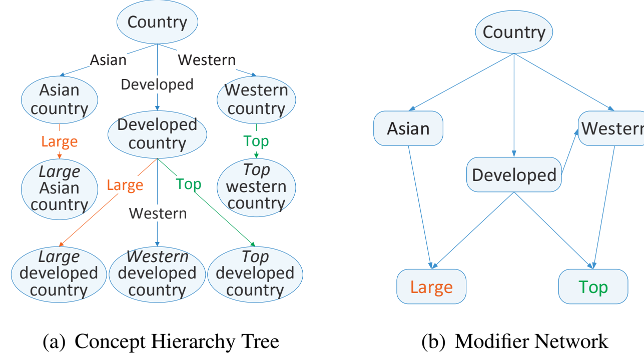
We now describe how to find terms that are often used as non-constraint modifiers. In the query “top Seattle hotels,” there is a difference between modifier “top” and “Seattle” in the sense that “Seattle” is a specific modifier, while “top” is subjective. In some applications, such as search, non-constraint modifiers can be and should be ignored.

Based on the head-modifier principle [Hippisley et al. 2005], given “large developed country” and “developed country,” we can deduce that “large” is a potential pure modifier. But this is not always the case For example, “hot” is not a pure modifier for “hot dog.” We solve this problem by noticing “dog” belong to the *animal* concept, while “hot dog” to the *snack* or *quick food* concept. Thus, we perform pure modifier detection within each concept domain or concept cluster [Li et al. 2013]. 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.”

Therefore, we consider using a mass of phrases or concepts to mine non-constraint modifiers. We use Probase for mining non-constraint modifiers. Probase is suitable for this purpose as it contains 2.7 million concepts, including many tail concepts such as “large developing country” that contain non-constraint modifiers. Furthermore, Probase is an open domain knowledgebase. Because non-constraint modifiers often work in all domains (e.g., “top” can occur in “top movies,” “top books,” and “top hotels”), this feature of Probase is important for mining non-constraint modifiers.

We use an example to illustrate the process of mining non-constraint modifiers. Consider the concept hierarchy in Fig. 5(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 5(a) to Fig. 5(b). This is done by first keeping the root concept unchanged, and converting edges to nodes. Edges with the same label are mapped to one node. We call the new graph the "modifier network."



(a) Concept Hierarchy Tree (b) Modifier Network

Fig. 5: Mining non-constraint modifiers

Generally, a non-constraint modifier is independent with its context, or it is always a non-constraint modifier regardless of its context. In contrast, heads and constraint modifiers 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 tend to be nodes not in the central part of the hierarchy. Therefore, we can leverage the node centrality to find the non-constraint modifier. The smaller centrality score a node has, the more likely it is a non-constraint modifier. 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:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2}$$

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 build a modifier network for each concept domain or concept cluster [Li et al. 2013], and normalize this betweenness centrality score in each modifier network:

$$NL(g(v)) = \log \frac{g(v) - min(g)}{max(g) - min(g)} \tag{3}$$

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

$$PMS(t) = \sum NL(g(t)) \tag{4}$$

Finally, we can calculate a non-constraint 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.

4. MINING CONCEPT PATTERNS

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

4.1. 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 such as "iphone 5 smart cover" and "smart cover iphone 5" directly, we know the same search intent is also expressed in other forms, for example, "smart cover **for** iphone 5." From the latter expression, it is clear that "smart cover" is the intent. This suggests that in queries where "smart cover" and "iphone 5" 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 for identifying head and modifiers [Hippisley et al. 2005; Soderland et al. 1995]. 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," and "A with B"), it is almost always true that A is the head and B is the modifier. 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, i.e., the heads and modifiers must be terms in Probase. Although Probase is big, there still are valid terms not included there. This is however not a problem because our goal is to find concept-level head-modifier pairs, and concepts can be derived from instances that exist in Probase.

4.1.1. Conflict at the instance level. There is a serious issue in using syntactic patterns described above directly to identify head and modifiers. In particular, conflicts may exist between the identified instance level head-modifier pairs, due to two factors: First, there are exceptions for the preposition-based identification due to the arbitrariness of user input. For example, both "cover **for** ipad" and "ipad **for** cover" exist in query logs, which leads to a conflict pair ($cover_{[h]}$, $ipad_{[m]}$) and ($ipad_{[h]}$, $cover_{[m]}$). We define the conflict that appears in the pairs identified by the same preposition (e.g., "for") as *Inner Preposition Conflict*. Second, the same two instances can be connected by different prepositions to express different search intents. For example, "built-in camera **on** laptop" and "laptop **with** built-in camera" generate conflicting instance pairs, one of which is to search "camera" while the other looks for "laptop." This kind of conflicts that occur in the pairs identified by different prepositions are referred to *Outer Preposition Conflict* in this paper, such as the conflict pair ($camera_{[h]}$, $laptop_{[m]}$) and ($laptop_{[h]}$, $camera_{[m]}$) derived from preposition "on" and "with" respectively.

We investigate the conflict ratio ($r_c = \frac{\text{number of conflict pairs}}{\text{total number of pairs}}$) at the instance level based on 6 month of query logs from BING. Table II shows the results. We can see that the ratio of instance level conflict is very low, even for the outer preposition conflict. Removing these conflict pairs is an intuitive way to reduce the conflict. However, this will increase the bias of the concept pattern and damage its general usage.

Table II: Conflict ratio at the instance level, with statistics based on 6 months of BING query logs

Conflict Type	Preposition	Ratio (%)
inner preposition conflict	*"for"*	≈ 0.049
	"with"	≈ 0.009
outer preposition conflict	*"for"* vs. *"with"*	≤ 0.011

As can be seen from Table III(a), the frequency gap in the inner preposition conflict is very large. The query "ipad **for** cover" seldom appears in the query log, neither does "laptop **for** camera." We can see that these rare queries are logically improbable. Thus, we can omit these rare queries to resolve the inner preposition conflict. As for the outer preposition conflict, as shown in Table III(b), we cannot simply ignore one of the conflict pairs and keep the other, since both of them are legal and valid queries.

Table III: Examples of frequency statistic for instance level conflict, based on 6 month of query logs from BING.

(a) Inner preposition conflict

conflict queries	Freq.
cover *for* ipad	**2381**
ipad *for* cover	1
camera *for* laptop	**308**
laptop *for* camera	12

(b) Outer preposition conflict

conflict queries	Freq.
camera *for* laptop	**308**
laptop *with* camera	**302**
smartboard *for* ipad	31
ipad *with* smartboard	91

Based on the above observations, we propose to keep these outer preposition conflicts and model the head-modifier relationships in a probabilistic way by using all the identified instance pairs (including conflicts). This is reasonable, because given a query with preposition we can easily identify its search intent, such as "built-in camera on laptop" and "laptop with built-in camera," but only given "laptop camera," even we human can hardly know exactly what the user wants. Although we don't know exactly what the user wants, we are able to learn that it is more likely (with score 0.9) that the *accessary* is the head and the *device* is the modifier.

4.2. Concept-level Head-Modifiers

From the instance-level head-modifier relationships obtained above, we derive concept-level relationships. The motivations are the following:

(1) The number of head-modifier relationships at the instance level is huge, which is unfavorable for online applications. *Lifting* the instance pairs to concept level can largely reduce the number, since one concept can represent a set of instances that belong to the same category. For instance, both "iphone 5" and "samsung galaxy" belong to *device*.
(2) The extracted instance-level relationships do not cover everything. A generalization mechanism is required. *Lifting* the instance pairs to concept level gives us a concise model that can be generalized to cover more instance-level occurrences. For example, we derive the concept pattern $(accessary_{[h]}, device_{[m]})$ from "smart cover for iphone 5." Then given a new query "samsung galaxy smart cover," we detect that "samsung galaxy" is a modifier, since "samsung galaxy" is a *device*.

4.2.1. Levels of Conceptualization. An instance maps to many concepts, some very specific and others very general. We can map a pair of instances such as ("smart cover," "iphone 5") to a pair of concepts in two extreme ways. First, we can map it to itself, i.e., we treat "smart cover" and "iphone 5" 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 useless, as it does not have the power of telling heads and modifiers apart.

A more challenging problem is the following. It may seem alright to map "skype for windows phone" to $(company_{[h]}, device_{[m]})$ and "iphone 5 for verizon" to $(device_{[h]}, company_{[m]})$. 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. Second, we must avoid concepts that are too general. Over generalization leads to conflict patterns, as each claims territory that it does not own.

4.2.2. Conceptualizing Instances. We now show how to properly map a single instance to a set of concepts. The mapping criterion for a term e must take into account both generality and specificity. Denote $C = \{c_1, \cdots, c_n\}$ is the set of e's concepts. We map e to those concepts c_i if $P(c_i|e) \cdot$

$P(e|c_i)$ is large. A larger probability indicates stronger corpus evidence of the closeness between c_i and e in the semantic network of Probase.

There is a problem for using the above method to directly conceptualize instances. An instance e can be a concept itself, and sometimes it is already the most appropriate concept. For example, if e="company," the above method may lead to concepts such as "organization" or even "object," which are too vague.

Ideally, we want to keep relatively popular concepts that cover a reasonable number of instances. Thus, in our work, we use a concept's entropy as an indicator:

$$H(c) = - \sum_{\text{e is an instance of c}} P(e|c) \log P(e|c) \tag{5}$$

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 ranked by the probability $P(c_i|e) \cdot P(e|c_i)$. Furthermore, for any $c_i \in C$, we give it a score $CS(e, c_i)$:

$$CS(e, c_i) = \begin{cases} 1 & c_i = e \\ P(c_i|e) \cdot P(e|c_i) & c_i \neq e \end{cases} \tag{6}$$

4.2.3. 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 patterns.

The task of combination, however, is not trivial due to the following two facts:

(1) **Ambiguous** entities may lead to wrong concept pairs. For example, the term "apple" can be conceptualized to *fruit* or *company*. Therefore, "CEO for apple" results in two possible concept pairs: (*corporate officer*, *company*) or (*corporate officer*, *fruit*). Obviously, the latter is wrong.
(2) **Conflict** concept pairs may exist during the combination. As we mentioned in section 1, both the conflict instance pairs and the conceptualization can lead to conflict concept patterns. We investigate the concept level conflict ratio based on 1% of query logs from BING collected in 6 months, as shown in Tab. IV. Considering all kinds of conflicts introduced by six prepositions, the conflict ratio at instance level goes up to about 1%. Then we select the top 1 concept for each instance using the proposed conceptualization method. Finally, the conflict ratio at concept level is about 10%, going up by **10** times after the conceptualization. This ratio may go up further if we set a larger k when conceptualizing instances.

Table IV: Conflict ratio goes up in concept level

Conflict Level	# total pairs	# of conflicts	Conflict Ratio
Instance level	575,367	6,019	$\approx 1.04\%$
Concept level	367,009	37,165	$\approx 10.12\%$

We propose to aggregate all kinds of instance-level head-modifier pairs for each concept pair. For the first ambiguity issue, aggregating different queries has the power of disambiguation. For example, there are similar queries such as "CEO for Microsoft" and "CEO for IBM." Both support the (*corporate officer*, *company*) pair but not the (*corporate officer*, *fruit*) pair. After the aggregation, wrong concept pairs introduced by ambiguous senses will have a low *support* and can be filtered out. For the second conflict issue, aggregating helps compute the proportion of each

conflicting concept pair in the data set. One example might be $(accessary_{[h]}, device_{[m]}, 0.9)$ and $(device_{[h]}, accessary_{[m]}, 0.1)$.

Specifically, for each instance level head-modifier pair, we conceptualize its head and modifier independently. We first 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 score each concept pair by Eq. 7:

$$Score(c_i, c_j) = \sum_{u,v} CS(e_u, c_i) \cdot CS(e_v, c_j) \cdot \log N(e_u, e_v) \tag{7}$$

where $CS(e, c)$ is the score of e mapping to c as defined in Eq (6), and $N(e_u, e_v)$ is 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 can ensure that concept pairs supported by a variety of entity pairs have higher scores.

Finally, there is another issue of similar concept pairs, as Probase contains concepts that are similar to each other, such as "country" and "nation." Li et al [Li et al. 2013] proposed a k-Medoids clustering algorithm to cluster these concepts. We leverage their clustering results directly for clustering concept pairs.

5. HEAD AND MODIFIER DETECTION

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

5.1. Parsing

Given a query, we first identify all the terms in the query 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 3 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 queries 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 disjoints cliques [Song et al. 2011].

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.

5.2. Head-modifier detection for 2 components

Consider a query with two components "smart cover" and "iphone 5." Fig. 6 demonstrates the process of head-modifier detection.

We first conceptualize "iphone 5" 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, $CS(e, c_i)$, which is given by Eq (6).

Then, we search the concept pattern knowledgebase, and find matches such as *(accessory, device)*, each of which is associated with a score $Score(c_1, c_2)$ given by Eq (7).

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

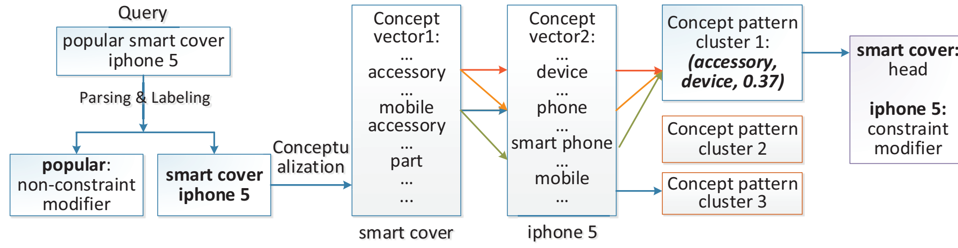


Fig. 6: Detection flow for 2-component queries

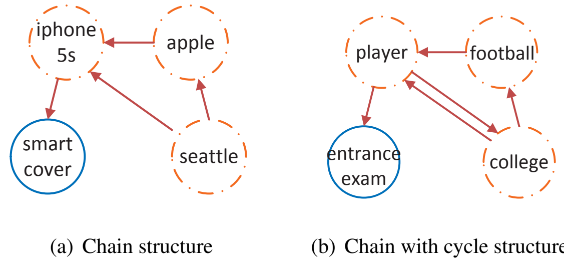
We aggregate the scores to identify the head and the modifier by Eq. 8. For two components t_1 and t_2, if $f(t_1, t_2) > f(t_2, t_1)$ then we conclude t_1 is the head and t_2 is the modifier.

$$f(t_1, t_2) = \sum_{c_1, c_2} CS(t_1, c_1) \cdot CS(t_2, c_2) \cdot Score(c_1, c_2)$$
$$\text{where} \quad CS(t, c) = \sum_{e_i \in comp} CS(e_i, c) \tag{8}$$

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

5.3. Head-modifier detection for multiple components

As we have shown in Fig. 2(b), a large number of search queries have more than two components. To detect the head, we first use the above process to detect head-modifier relationships 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, as shown in Fig. 7. The direction of an edge is from the modifier to the head.



(a) Chain structure (b) Chain with cycle structure

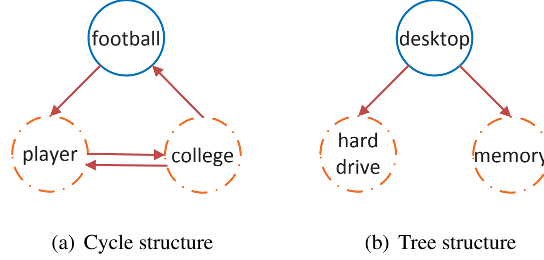
Fig. 7: Components connected by head-modifier relationships

There are two typical graph structures:

(1) *Chain structure*: The graph is acyclic, and we can find a set of *chains* from a sequence of modifiers to the head. In Fig. 7(a), there are 2 paths: $seattle \rightarrow apple \rightarrow iphone\ 5s \rightarrow smart\ cover$ and $seattle \rightarrow iphone\ 5s \rightarrow smart\ cover$, each of which describes a sequence of modifying relationships.

(2) *Chain with cycle structure*: There are chains and cycles in the graph, as shown in Fig. 7(b). Cycles are often introduced by ambiguous head-modifier relationships. In Fig. 7(b), we may have both player→college and college→player, the former of which for the intent of finding "player," and the latter for finding "college."

For the *chain structure* graph, we can identify the head easily as the terminal node of the chains. As for the *chain with cycle structure*, we break it down into two basic cases: cycle structure and tree structure, as shown in Fig. 8. We analyze them separately.



(a) Cycle structure (b) Tree structure

Fig. 8: Two basic cases of the chain with cycle structure

First, for *cycle structure*, we propose to remove the cycle using the following method. If there is an obvious head connected to the cycle, as in the example shown in Fig 7(b), the entire cycle becomes a modifier of the head. Otherwise, we break the cycle by removing the weakest edge, as each edge is associated with a weight given by Eq. 8. For example, the college $\rightarrow$ player link in Fig. 8(a) is likely the weakest (it matches conflicting patterns and the two words are not close to each other in the short text). After removing the cycle, we know "player" is the head, while "college" and "football" are modifiers.

Second, in the *tree structure*, chains may have different terminations (leaf nodes). It implies that there are more than one heads, which conflicts with the one-head assumption. We call this conflict as **multi-head conflict**. It is often introduced by the 2-component head-modifier detection. As shown in Fig. 8(b), "hard drive" and "memory" are detected as the heads for the query "1TB hard drive 128GB memory desktop." However, it is more likely that we should go with the "desktop with ..." semantics, i.e., the "desktop" is the real head. In the following subsection, we discus our solution for multi-head conflict in detail.

5.3.1. Conflict Resolution. **Intuition**: *In multi-component queries, the head are more likely to be modified by all the modifiers, and tends to have certain relationship with all the other components.* Taking "1TB hard drive 128GB memory desktop" for example, "desktop" is the real head, and it has the *Attribute-Of* relation with "hard drive" and "memory." Removing one modifier such as "hard drive," there still is a head-modifier relationship in the query "128GB memory desktop." However, if removing the head "desktop," there won't be a clear head-modifier relation in the query "1TB hard drive 128GB memory," which leads to a confusing query.

Based on the above intuition, we propose to select the component that has the maximum probability as the head. We define the probability of a component being the head as the product of the probability of it being *"modified"* by all other components. Here we use a strong assumption that the head-modifier relationships are independent from each other. Formally, given a query $Q = \{t_1, t_2, ..., t_n\}$, we detect its head $h \in Q$ by Eq. 9, where $pm(t_i, t_j)$ denotes the probability of t_i being modified by t_j.

$$h = \arg\max_{t_i \in Q} \prod_{t_j \in Q, t_j \neq t_i} pm(t_i, t_j) \tag{9}$$

Estimating $pm(t_i, t_j)$ is a challenging task. Only using the proposed head-modifier score $f(t_i, t_j)$ (Eq. 8) is insufficient, which may cause multi-head conflict as shown in Fig. 8(b). This is because usually semantics in multi-component queries are more complicated. For example, the "hard drive"

and the "memory" are attributes of the "desktop." These are not covered by the learnt concept patterns. The head-modifier score $f(t_1, t_2)$ (Eq. 8) designed for 2 components is not enough for multi-component queries. Therefore, we propose to leverage extra statistical information to estimate $pm(t_i, t_j)$.

We first learn $pr(t_i, t_j)$, the prior probability of two components having a certain relationship, from the search log. Our intuition is as follows:

Intuition: *For queries containing two instances, usually the two instances have a certain relationship (e.g., head-modifier relationship).* For example, there are many queries that are composed of "desktop" and "hard drive" in search log. But we hardly find a query that is composed of only "memory" and "hard drive."

Then we combine the prior probability with the head-modifier score to detect the head from multiple components, since the learnt prior probability $pr(t_i, t_j)$ only reflects the probability of the two components having a certain relationship, which cannot capture the head-modifier relation explicitly. Formally, we estimate $pm(t_i, t_j)$ by Eq. 10:

$$\begin{aligned} pm(t_i, t_j) &\approx f(t_i, t_j) \times pr(t_i, t_j) \\ &\approx f(t_i, t_j) \times \frac{c(t_i, t_j)}{\mathcal{N}} \end{aligned} \tag{10}$$

where $c(t_i, t_j)$ is the number of times that component t_i and t_j compose a query and $\mathcal{N}$ denotes the total number of queries in the search log.

A smoothing method is needed when $c(t_i, t_j) = 0$, which will lead to a zero score in Eq 10. In this case, we estimate $pr(t_i, t_j) = pr(t_i) \times pr(t_j)$, where $pr(t) \approx \frac{c(t)}{\mathcal{N}}$ is the probability of component t appearing in the search log. To sum up, we estimate $pm(t_i, t_j)$ by Eq. 11. Note $\mathcal{N}$ is uniform to each component and so do not affect the ranking in Eq. 9.

$$\begin{aligned} pm(t_i, t_j) &\approx \begin{cases} f(t_i, t_j) \times \frac{c(t_i, t_j)}{\mathcal{N}} & : \ c(t_i, t_j) > 0 \\ f(t_i, t_j) \times \frac{c(t_i)c(t_j)}{\mathcal{N}^2} & : \ c(t_i, t_j) = 0 \end{cases} \\ &\propto \begin{cases} f(t_i, t_j) \times c(t_i, t_j) & : \ c(t_i, t_j) > 0 \\ f(t_i, t_j) \times \frac{c(t_i)c(t_j)}{\mathcal{N}} & : \ c(t_i, t_j) = 0 \end{cases} \end{aligned} \tag{11}$$

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 that do not strictly follow the grammar. For applications such as sponsored search that require to match two short texts (e.g., query and ads bid keywords), we may match heads first and then modifiers, and use the weight of each component to quantify the match.

6. EXPERIMENTS

We present a comprehensive experimental study of our query head-modifier detection mechanism. We compare it with previous approaches and discuss applications such as sponsored search that benefit from the technique.

6.1. 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 VI.

To measure the quality of our method and determine 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 VII.

Then we bin the labeled terms in intervals of non-constraint 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

Table V: 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
Total unique queries	308,183,923					
Total traffic	12,871,641,724					

Table VI: Top Ranked Non-Constraint Modifiers

Rank	Modifier	Rank	Modifier	Rank	Modifier	Rank	Modifier
1	good	11	popular	21	single	31	high
2	traditional	12	conventional	22	normal	32	suitable
3	common	13	standard	23	second	33	specialty
4	typical	14	local	24	complex	34	so-called
5	great	15	regular	25	famous	35	powerful
6	small	16	basic	26	true	36	minor
7	large	17	big	27	first	37	natural
8	modern	18	classic	28	commercial	38	non
9	simple	19	real	29	strong	39	external
10	well-known	20	key	30	sometimes	40	ordinary

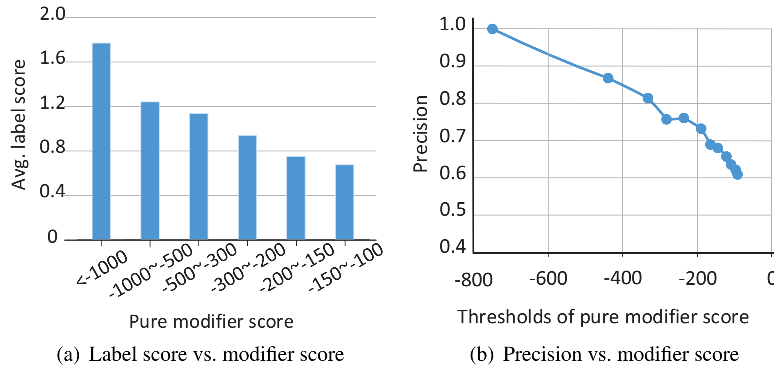
Table VII: Manual Label Criteria for Non-Constraint Modifier

Score	Explanation	Examples
2	always be non-constraint modifier	top, best
1	be non-constraint modifier sometimes	electronic, thermally
0	generally can't be negligible	American, library

labeled score (0$\sim$2). The result is shown in Fig. 9(a), where the x-axis represents the non-constraint 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 results, shown in Fig. 9(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 non-constraint modifiers.

6.2. 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 4.1 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" and "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 (e.g., "* 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 V breaks down the number across different syntactic patterns. Patterns with 'of', 'in', 'for' have similar proportion in all of the matches while patterns with 'with,' 'on,' 'at' account for a smaller proportion.



(a) Label score vs. modifier score (b) Precision vs. modifier score

Fig. 9: Performance of modifier detection

6.3. Building Concept Pattern Knowledge base

Using the method described in Section 4, 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 VIII: 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 VIII, from 14,144,235 instance level pairs we obtain 84 million concept level pairs. 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 [Li et al. 2013] (available in the download link with Probase dataset), we

Table IX: 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.

further reduce to 169,966 pairs. Thus, we obtain a concise model for head-modifier relationships. Table IX shown some top concept level patterns.

We can see concept patterns in many different domains in Table IX, 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 X: 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 X 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 dataset. 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.

6.4. Accuracy of Head-Modifier Detection

6.4.1. Data. To evaluate our proposed detection method, we collect queries from real search logs. 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 ($A_{[h]}$, $B_{[m]}$) and ($B_{[h]}$, $A_{[m]}$) exist in the data, we remove both of them). This gives us a high quality, automatically labeled head/modifier pairs: ($A_{[h]}$, $B_{[m]}$). 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. 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).

6.4.2. Metrics. We measure accuracy of the analysis. Specifically, let us assume that the true heads and modifiers of a query are $Q_h = \{t_1^h, t_2^h, ..., t_N^h\}$ and $Q_m = \{t_1^m, t_2^m, ..., t_M^m\}$. We say that a component t is a true positive if it is detected as a head and $t \in Q_h$ or a modifier and $t \in Q_m$. The detection is correct if all components are true positives. Query accuracy is measured by the total number of correct queries divided by the total number of queries.

6.4.3. Detection accuracy for 2 components. Table XI shows the accuracy of our method on the 6 testing data sets. As we can see, our method achieves 90+% accuracy in all testing data sets, which demonstrates that the mechanism we proposed for query head-modifier detection is effective.

Table XI: Statistics of Testing Data Sets and Detection Accuracy

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

To investigate the performance of our method on conflicting queries, we select ten such cases from the search log and recruit five human judges to label the head and modifier for each query. To avoid biases in labeling, we also use another statistical method for the labeling by leveraging query click log. Intuitively, queries with the same intent tend to have same clicked URLs. Thus, for each conflicting query such as "laptop built-in camera," we collect its related queries through co-clicked URLs. Then, we find queries matching the 'for' pattern from its related queries (e.g., "camera for laptop"). Finally, the role of each component is voted by their pattern-based related queries. For example, ("camera"$_h$, "laptop"$_m$) will get one vote from "camera for laptop." As shown in Table XII, there is a general consensus among the annotators about the most likely head in each conflicting query. And so does the query click log, which further proves the fairness of the annotators' judgement. Although it is hard to know exactly which one is the head in conflicting queries, our method is able to detect the heads consistently with human thinking.

Table XII: Results of conflicting query detection

Query	Human Votes	Click-log Votes
clothes *women*	5/5	788/790
kids **video**	4/5	360/405
toys *kids*	5/5	366/370
frosting **cupcakes**	4/5	266/315
pasta **sauce**	3/5	87/128
smart cover *ipad*	5/5	71/77
laptop built-in **camera**	5/5	17/27
built-in **cabinets** *kitchen*	5/5	23/26
camera *laptop*	4/5	11/15
500 G **hd drive** *desktop*	5/5	8/11

Note: **Head** is in bold; *modifier* is in italics and underlined.

6.4.4. Detection performance on multi-components. Unlike 2-component queries, it is hard to automatically collect large-scale high quality labeled data for multi-component queries. Thus, we investigate the performance of the combined detection approach for multi-components through a qualitative evaluation. We select 10 queries from the testing data set for 3-component queries. The selected queries are then labeled by the same 5 annotators as the above conflict test. Table XIII shows the results on queries with multiple components. As can be seen from Table XIII, it achieves good performance in most cases. But it performs a little worse on multi-component queries than 2-component queries, with one error detection for the query "beachfront lodging myrtle beach." This is reasonable because in the detection, we rely on 2-component detection results where errors may exist, in addition to the estimation error on prior probability of two components by Eq. 11.

6.4.5. A comparison with a position-based method. We also verify one important point in head-modifier detection for queries. Usually, in a phrase, the head appears as the last noun, and words appearing before the head is its modifier. However, this rule does not work for search queries that do not observe grammars of a natural language. We conduct an experiment to validate it. Using the 2 components queries in the 6 month testing data set, we calculate how many times the head appears

Table XIII: Results of 3-component query detection

Query	Human Votes
abercrombie *winter* **cloth**	5/5
red hat *server* **hardware**	5/5
business **grants** for *women*	5/5
seattle **jobs** *craigslist*	5/5
kids *cooking* **games**	5/5
beachfront *lodging* *myrtle beach*	1/5
women seeking *men* **marriage**	5/5
1TB *hd drive* 128GB *memory* **desktop**	5/5
enlistment *texas* *ww2*	4/5
toyota *headlight cover* **cleaner**	5/5

Note: **Head** is in bold; *modifier* is in italics and underlined.

last after the modifier and find that head appears last in only *41.45%* unique queries. This suggests that only using the position to detect the head does not work well.

6.5. A Comparison with Other Methods

We compare the performance of our method with an existing approach [Bendersky et al. 2010], and 3 alternative approaches based on our approach.

6.5.1. Comparison with an existing method. Bendersky et al. [Bendersky et al. 2010] proposed a weighting mechanism to measure the concept importance, which is a classical way of query intent detection. In a query, different terms will be assigned different weights. The terms with the highest weight could be seen as the head of this query. We implemented the method of term weighting. 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 constraint entity, and add up these vectors for head ($\vec{h}$) and modifier ($\vec{m}$) respectively. We use libSVM package [Chang and Lin 2011] with linear kernel to train and classify. However, the performance of this approach is poor and the accuracy is only about *55%*. The reasons may be the features they chose do not have direct relationship with our head-modifier relationship. By this experiment, we can also conclude that traditional term weighting approach cannot resolve the head-modifier detection problem.

6.5.2. Comparison with 3 alternative approaches. To assess the effectiveness of the proposed method, we compare it with 3 possible alternative approaches based on our approach. We first briefly describe the alternative approaches as follows:

— *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, i.e., the difference between the frequency of the entity serving as a head and the frequency of it as a modifier. 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 detect the head and modifier in this way, and we classify these queries as *Not Identifiable*.

— *Entity-oriented pattern detection (**EOPT**)*: In this approach, we create a dictionary of three columns: *($entity_{[h]}$, $entity_{[m]}$, 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 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 (6). Similarly, we can get CS_j^M. The score of this concept in the dictionary will be $Score(c) = \sum CS_i^M - \sum CS_j^H$. For a new query, we conceptualize each entity as Equation (6) 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 these methods and our method (which we call ***COPT*** or *concept-oriented pattern detection*) using the same testing data set. Two evaluation metrics Accuracy and Not Identifiable Rate are used in this evaluation. Table XIV shows the results, $*$ denotes the improvement over the baseline is statistically significant (sign-test, p value < 0.05). From this table, we draw the following observations:

First, the concept based methods (COMT and COPT) perform much better than entity-based methods (EOMT and EOPT). Compared with the best entity-based method EMOT, the accuracy was significantly improved by 14.29% (COMT) and 20.06% (COPT) (in Table XIV(a)). It indicates that lifting the instances to a higher level of concepts is effective for the head-modifier detection task. As can be seen from Table XIV(b), the not identifiable rate of concept-based methods decreased significantly. This shows that concept-based method achieves much better coverage.

Second, our method COPT is superior to COMT. The accuracy was improved by 5.77% and the not identifiable rate declined markedly. COMT handles each entity independently and ignores the relationships between entities in a query. In contrast, our COPT takes the head-modifier relationship into consideration. This shows that the mined concept pattern knowledge works well on this task.

Third, EOPT is inferior to EOMT in terms of accuracy. This is reasonable. EOPT considered the head-modifier relationship between entities. However, it also suffered a lot from the low coverage of the entity pattern dictionary, as can be seen from its high not identifiable rate (40% in Table XIV(b)). From another angle, it shows the importance of the conceptual knowledge in the task of head-modifier detection.

Table XIV: Comparison of different methods on 6 month query log

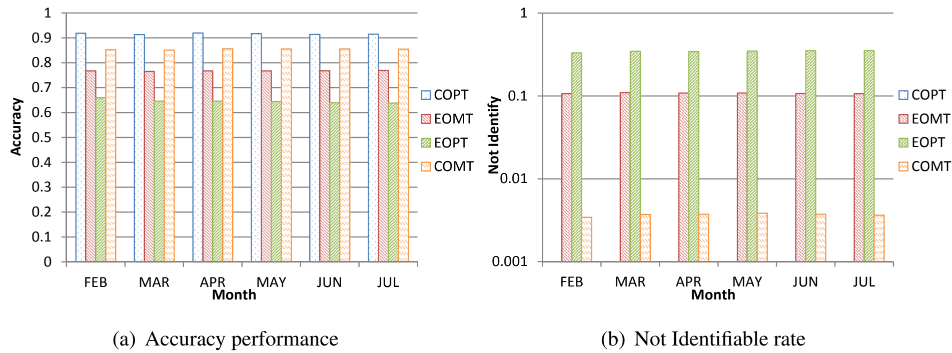
(a) Accuracy comparison

Different methods	Accuracy (%)	Impro. (%)
EOMT	70.38	-
EOPT	56.85	-13.53
COMT	84.67*	+14.29
COPT	90.44*	+20.06

(b) Non-identifiable comparison

Different methods	Not-identifiable
EOMT	0.10252
EOPT	0.40370
COMT	0.00733
COPT	0.00005

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 only about 64% and the non-identifiable rate is about 33%. These results show that our concept-oriented pattern detection approach has better extendibility to unknown queries than entity-based method.



(a) Accuracy performance (b) Not Identifiable rate

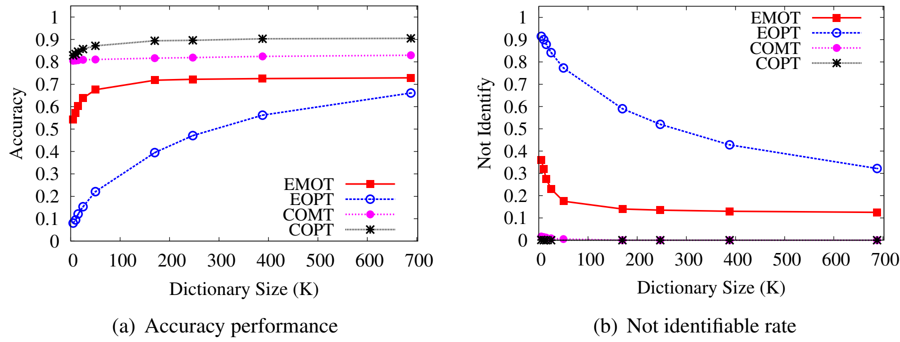
Fig. 10: Test on each month query log

Comparing the results of different methods by month, neither the accuracy nor the not-identifiable-rate changes much (Fig. 10). 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 (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.

6.6. Impact of Scoring Functions and Parameters

We evaluate the influence of dictionary sizes (i.e., the number of concept pairs) on accuracy. As can bee seen from Fig. 11, for all methods, the accuracy increases and the *Not Identifiable* rate decreases when the dictionary becomes larger, but different methods reach saturation at different dictionary sizes. EOPT is not saturated even at 0.7M while COMT saturates at about 0.1M. It is not surprising, because the coverage of entity-based method is much less than the concept-based method. Besides, 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 experiment on 6 month test data set using $P(c|e)$ ("Probability") and $P(e|c)$ ("Typicality") as conceptualization score functions separately. Table XV shows the performances of different parameters, where $*$ denotes the improvement over the baseline is statistically significant (sign-test, p value < 0.05). As Table XV(a) shows, the accuracy using the former is 86% and the latter one is 87%. Both are less than our method ("Top10"). On the other hand, the *Not Identifiable* rate of "Probability" is less than that of "Typicality" while our method ("Top10") is in the middle (in Table XV(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. We also test the performance of mapping to top 5 concepts for each entity



(a) Accuracy performance

(b) Not identifiable rate

Fig. 11: Performance on different dictionary size

instead of top 10. The accuracy is 87.5% and the *Not Identifiable* rate is larger than that of "Top 10." Thus, mapping to 10 concepts achieves better coverage and accuracy.

Table XV: Comparison of different parameters on 6 month query log

(a) Accuracy comparison

Different parameters	Accuracy (%)	Improvement (%)
Typicality	87.04	-
Probability	86.33	-0.71
Top5	87.51.44	+0.47
Top10	90.44*	+3.40

(b) Non-identifiable comparison

Different parameters	Not-identifiable
Typicality	0.000382
Probability	0.000007
Top5	0.000049
Top10	0.000348

6.7. Application on Sponsored Search

It is a challenging task to find matching ads for search queries, especially for tail queries, since tail queries 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, we label '2' for the pair; if the query and bid keywords share the same main component but with different non-negligible modifiers, we label '1' for the pair; we label '0' for pairs with no matching components. 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 XVI.

The computational cost of our head-modifier detection approach is acceptable. As the framework in Figure 3 shows, we did much offline work to mine two important knowledge: non-constraint modifier list and concept pattern dictionary. During the online head-modifier detection, the computational cost of Modifier filtering and Pattern searching can be largely reduced by using dictionary matching method. The process of Conceptualization can be also seen as a dictionary matching method, where the isA relationships with probabilistic scores in Probase are regarded the dictionary. For head-modifier detection, the most cost part is computing the product of different matched scores costs (i.e., Eq. 8 and Eq. 11), which is also not complex. In practice, the average time cost

Table XVI: 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
are benefits exercise concerning heart health	benefits exercise	exercise tips;10 benefits exercise;

of the proposed head-modifier detection approach is less than 0.01 s/query, which is acceptable for online applications. This mechanism we developed for ads matching has been used in production.

7. RELATED WORK

Most query intent detection methods are based on query topic classification [Shen et al. 2006; Li et al. 2008; Hu et al. 2009]. The task of KDD Cup 2005 was to classify queries into 67 categories [Li et al. 2005]. 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 intents but both are classified into the category of *job*. As an alternative, grouping queries into intents by means of query clustering has also been popularly studied [Cheung and Li 2012a; Hu et al. 2012; Ren et al. 2014]. However, the implicit intents (clusters) are too coarse-grained to interpret the specific intent in search queries, for example, interpreting that the user wants "smart cover," rather than "iphone 5" given "popular smart cover iphone 5."

Bendersky et al [Bendersky et al. 2010] worked on the problem of assigning different weights to different terms in a query. Kumaran et al [Kumaran and Carvalho 2009] turned long queries into short ones by dropping less significant terms in the query. 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 [Li 2010; Cheung and Li 2012b; Agarwal et al. 2010; Li et al. 2013]. Li et al. [Li 2010] 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. Li et al. [Li et al. 2013] proposed a clustering framework for finding synonymous query intent templates for canonical query intent templates. Cheung et al. [Cheung and Li 2012b] clustered queries and constructed patterns for each domain, aiming at domain-dependent structured search. Chang et al. [Agarwal et al. 2010] 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 noun phrase queries without assuming underlying common structures, and are not confined to a specific domain.

There are some existing works [Pasca and Van Durme 2007; Paşca and Van Durme 2008] 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 [Agichtein and Gravano 2000] 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.

8. CONCLUSION

In this paper, we introduce a semantic approach for query head-modifier 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 search queries. The technique described in this paper can directly benefit semantic similarity calculation between two queries as it is able to assign different weights to different components of these queries based on their head and modifier structures.

This work suggests some interesting directions for future work. For example, when handling the multi-component queries, statistical information is used to estimate the extra semantic relation strength in each component pair. A very interesting future work is combining more semantic knowledge (e.g., *Attribute-Of* relation) with existing detection mechanism. Besides, 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. It was also supported by Beijing Advanced Innovation Center for Imaging Technology (No.BAICIT-2016001), the National Natural Science Foundation of China (Grand Nos. 61370126, 61672081), National High Technology Research and Development Program of China (No.2015AA016004),the Fund of the State Key Laboratory of Software Development Environment (No.SKLSDE-2015ZX-16).

REFERENCES

Ganesh Agarwal, Govind Kabra, and Kevin Chen-Chuan Chang. 2010. Towards rich query interpretation: walking back and forth for mining query templates. In *WWW*. ACM, Raleigh, North Carolina, USA, 1–10.

Eugene Agichtein and Luis Gravano. 2000. Snowball: extracting relations from large plain-text collections. In *DL*. ACM, San Antonio, Texas, United States, 85–94.

Michael Bendersky, Donald Metzler, and W. Bruce Croft. 2010. Learning concept importance using a weighted dependence model. In *WSDM*. ACM, New York, USA, 31–40.

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*. ACM, Vancouver, Bc, Canada, 1247–1250.

Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.* 2, 3 (2011), 27:1–27:27.

Jackie Chi Kit Cheung and Xiao Li. 2012a. Sequence clustering and labeling for unsupervised query intent discovery. In *WSDM*. ACM, Seattle, Wa, USA, 383–392.

Jackie Chi Kit Cheung and Xiao Li. 2012b. Sequence clustering and labeling for unsupervised query intent discovery. In *WSDM*. ACM, Seattle, Washington, USA, 383–392.

David A Evans and Chengxiang Zhai. 1996. Noun-phrase analysis in unrestricted text for information retrieval. In *ACL*. ACL, Santa Cruz, California, USA, 17–24.

Andrew Hippisley, David Cheng, and Khurshid Ahmad. 2005. The head-modifier principle and multilingual term extraction. *Nat. Lang. Eng.* 11, 2 (2005), 129–157.

Jian Hu, Gang Wang, Fred Lochovsky, Jian-tao Sun, and Zheng Chen. 2009. Understanding user's query intent with wikipedia. In *WWW*. ACM, Madrid, Spain, 471–480.

Yunhua Hu, Yanan Qian, Hang Li, Daxin Jiang, Jian Pei, and Qinghua Zheng. 2012. Mining query subtopics from search log data. In *SIGIR*. ACM, Portland, Oregon, USA, 305–314.

Jinyoung Kim, Xiaobing Xue, and W Bruce Croft. 2009. A probabilistic retrieval model for semistructured data. In *ECIR*. Springer, Toulouse, France, 228–239.

Giridhar Kumaran and Vitor R. Carvalho. 2009. Reducing long queries using query quality predictors. In *SIGIR*. ACM, Boston, MA, USA, 564–571.

Taesung Lee, Zhongyuan Wang, Haixun Wang, and Seung Won Hwang. 2011. Web Scale Taxonomy Cleansing. *Proceedings of the VLDB Endowment* 4 (2011), 1295–1306.

Hang Li, Gu Xu, Bruce Croft, and others. 2011. Query representation and understanding. In *SIGIR Workshop on Query Representation and Understanding*, Vol. 44. ACM, Beijing, China, 48–53.

Peipei Li, Haixun Wang, Kenny Zhu, Zhongyuan Wang, and Xindong Wu. 2013. Computing Term Similarity by Large Probabilistic isA Knowledge. In *CIKM*. ACM, San Francisco, CA, USA, 1401–1410.

Xiao Li. 2010. Understanding the semantic structure of noun phrase queries. In *ACL*. ACL, Uppsala, Sweden, 1337–1345.

Xiao Li, Ye-Yi Wang, and Alex Acero. 2008. Learning query intent from regularized click graphs. In *SIGIR*. ACM, Singapore, Singapore, 339–346.

Yanen Li, Bo-June Paul Hsu, and ChengXiang Zhai. 2013. Unsupervised identification of synonymous query intent templates for attribute intents. In *CIKM*. ACM, San Francisco, CA, USA, 2029–2038.

Ying Li, Zijian Zheng, and Honghua (Kathy) Dai. 2005. KDD CUP-2005 report: facing a great challenge. *SIGKDD Explor. Newsl.* 7, 2 (2005), 91–99.

Marius Paşca and Benjamin Van Durme. 2008. Weakly-Supervised Acquisition of Open-Domain Classes and Class Attributes from Web Documents and Query Logs. In *Proceedings of ACL-08: HLT*. ACL, Columbus, Ohio, 19–27.

Stelios Paparizos, Alexandros Ntoulas, John Shafer, and Rakesh Agrawal. 2009. Answering web queries using structured data sources. In *SIGMOD*. ACM, Providence, Rhode Island, USA, 1127–1130.

Marius Pasca and Benjamin Van Durme. 2007. What you seek is what you get: extraction of class attributes from query logs. In *IJCAI*. Morgan Kaufmann Publishers Inc., Hyderabad, India, 2832–2837.

Xiang Ren, Yujing Wang, Xiao Yu, Jun Yan, Zheng Chen, and Jiawei Han. 2014. Heterogeneous graph-based intent learning with queries, web pages and Wikipedia concepts. In *WSDM*. ACM, New York City, USA, 23–32.

Stephen Robertson, Hugo Zaragoza, and Michael Taylor. 2004. Simple BM25 extension to multiple weighted fields. In *CIKM*. ACM, Washington, DC, USA, 42–49.

Dou Shen, Jian-Tao Sun, Qiang Yang, and Zheng Chen. 2006. Building bridges for web query classification. In *SIGIR*. ACM, Seattle, Washington, USA, 131–138.

Stephen Soderland, David Fisher, Jonathan Aseltine, and Wendy Lehnert. 1995. CRYSTAL: Inducing a conceptual dictionary. In *IJCAI*, Vol. 2. AAAI, Montreal, Quebec, Canada, N12.

Yangqiu Song, Haixun Wang, Zhongyuan Wang, Hongsong Li, and Weizhu Chen. 2011. Short Text Conceptualization using a Probabilistic Knowledgebase. In *IJCAI*. AAAI Press, Barcelona, Catalonia, Spain, 2330–2336.

Wentao Wu, Hongsong Li, Haixun Wang, and Kenny Q. Zhu. 2012. Probase: a Probabilistic Taxonomy for Text Understanding. In *SIGMOD*. ACM, Scottsdale, Arizona, USA, 481–492.