CRESST REPORT 805

A NEW FRAMEWORK FOR TEXTUAL INFORMATION MINING OVER PARSE TREES

SEPTEMBER, 2011

Hamid Mousavi
Deirdre Kerr
Markus R. Iseli

National Center for Research
on Evaluation, Standards, & Student Testing
UCLA | Graduate School of Education & Information Studies

A New Framework for Textual Information Mining Over Parse Trees

CRESST Report 805

Hamid Mousavi
CSD/University of California, Los Angeles
Deirdre Kerr & Markus R. Iseli
CRESST/University of California, Los Angeles

September, 2011

National Center for Research on Evaluation,
Standards, and Student Testing (CRESST)
Center for the Study of Evaluation (CSE)
Graduate School of Education & Information Studies
University of California, Los Angeles
300 Charles E. Young Drive North
GSE&IS Bldg., Box 951522
Los Angeles, CA 90095-1522
(310) 206-1532

Copyright © 2011 The Regents of the University of California.

The work reported herein was supported by grant number OPP1003019:1 from the Bill and Melinda Gates Foundation with funding to National Center for Research on Evaluation, Standards, and Student Testing (CRESST).

The findings and opinions expressed in this report are those of the author(s) and do not necessarily reflect the positions or policies of the Bill and Melinda Gates Foundation.

To cite from this report, please use the following as your APA reference:
Mousavi, H., Kerr, D., & Iseli, M. R. (2011). *A New Framework for Textual Information Mining over Parse Trees.* (CRESST Report 805). Los Angeles, CA: University of California, National Center for Research on Evaluation, Standards, and Student Testing (CRESST).
.

TABLE OF CONTENTS

A NEW FRAMEWORK FOR TEXTUAL INFORMATION MINING OVER PARSE TREES

Hamid Mousavi
CSD/University of California, Los Angeles
Deirdre Kerr & Markus R. Iseli
CRESST/University of California, Los Angeles

Abstract

Textual information mining is a challenging problem that has resulted in the creation of many different rule-based linguistic query languages. However, these languages generally are not optimized for the purpose of text mining. In other words, they usually consider queries as individuals and only return raw results for each query. Moreover they cannot effectively express ambiguities, cannot adapt to different domains, require a large number of rules in order to accurately extract information, and are not very user-friendly. This paper introduces a new text mining framework using a tree-based Linguistic Query Language, called LQL. The framework generates more than one parse tree for each sentence using a probabilistic parser, and annotates each node of these parse trees with main-parts information which is set of key terms from the node's branch based on the linguistic structure of the branch. The main- parts can be specialized for different domains based on a user- generated list of concepts. Using main-parts-annotated parse trees for a given textual dataset, the system can efficiently answer individual queries as well as mine the text for a given set of queries. The framework also has the ability to support grammatical ambiguity through probabilistic rules and linguistic exceptions in order to increase the quality of the extracted information.

Introduction

Using rich grammatical structures to extract information from unstructured text has recently attracted more attention in several applications such as content-based search engines, automatic reviewing systems, biomedical text mining systems, text summarization and topic extraction systems, and spam detection systems. In such techniques, the text is converted by linguistic parsers to a tree-based structure, called *parse tree*. Then, a query language is used to define pattern-like queries and a query engine searches the parse trees to find matches for the queries and to extract meaningful connections among the words in the text. The results can be used directly or indirectly through statistical techniques in each application. We will refer to this type of technique as pure NLP-based techniques as opposed to statistical NLP techniques. The pure NLP-based techniques promise more accurate results than the statistical NLP techniques, which are essentially based on the co-occurrence of words in the same piece of text.

Although a lot of research has been done on NLP-based techniques (see Section VII for more details), several challenges still need to be addressed before text mining applications can efficiently utilize pure NLP-based techniques. First, to mine information from texts (such as the features of a specific electronic device, or the protein-gene relationship in biomedical contexts), many queries have to be created since there are i) many ways to express the same thing in a natural language and ii) existing query languages are not usually designed specifically for the purpose of text mining. Automatic generation of queries are not much better, because it usually generates a lot of queries and needs a huge textual

dataset for its learning phase. Second, query engines in many existing works are slow due to not only the high number of queries but also the structure of the queries, which requires search in depth of parse trees to extract information. Many techniques try to address this issue by limiting the expressiveness of their query language. Third, to the best of authors' knowledge, all existing works consider each query on a stand-alone basis. In other words, they are not best optimized for combining the results of different queries for information mining inside their system. Fourth, most of proposed frameworks can not efficiently adapt to different domains, that is they are either best optimized for a fix domain or they do not consider a specific domain. Moreover, several of these languages are not user-friendly and need complex coding which may be a hindrance when asking linguists to design rules (queries) for linguistic systems. In these languages, it is also hard to handle issues such *linguistic exceptions*, *Anaphora resolution*, *co-reference resolution*, etc.. In this paper, we propose an text mining framework and a new Linguistic Query Language (LQL) to address the aforementioned issues. The framework is specifically designed for efficiently querying and mining parse trees generated from probabilistic parsers. It enriches the nodes in the parse trees with main-parts during pre-processing. Main-parts extract key terms for each node from its branch based on the linguistic structure of the branch. These main-parts can be specialized for a specific domain through a user-generated list of concepts. This way, the query language can be designed with simpler, yet sufficient features. Consequently, the query engine can run faster while finding more results (See Section VI).

In a glance, the proposed framework and language let users specify sets of tree-like patterns, called *pattern trees* and *formatting results* parts to extract tuples from the *matched trees*. The combination of *pattern trees* and *formatting results* parts is referred to as a *linguistic rule* or simply a *rule*. As we will show in this paper, generating these rules does not require a broad knowledge of programming. It can also be easily supported by a user friendly graphical user interface. In addition, LQL supports probabilistic rules in order to handle ambiguities in a more efficient way. Being developed in the process of designing the NLP-based text mining framework, LQL includes only necessary and practical utilities which improve computational efficiency and user-friendliness. More specifically, our paper makes the following contributions:

- We introduce a new NLP-based text mining framework, which utilizes more than one parse tree for each sentence. The framework provides each node in the parse trees with appropriate information, called *main-parts* in preprocessing. This enables the system to answer queries in real time and mine more information from less text compared to existing approaches (See Section V).
- We provide a new tree-based Linguistic Query Language (LQL) capable of specifying tree-based patterns called *pattern trees* and capturing results as sets of tuples (with any number of items in them). This helps in converting parse trees into more meaningful and machine understandable structures such as labeled directed hypergraphs. The language can also be fed with a list of concepts to adapt to a specific domain (See Section III).
- We support probabilistic rules and relationships using more than one parse tree, which to our knowledge has not been done at the linguistic query language level. This is essential to deal with the inherent ambiguities in natural language and provide more accurate results.

LQL also provides several other features: each query in LQL can return one or more tuples each with their own probabilities, queries for the extraction of multi-word terms are supported, and a *'whether or not'* feature is included to reduce the total number of rules.

In order to evaluate our framework, we have generated a set of 38 LQL rules to extract relationships between mathematical concepts in textual datasets. Our evaluations (Section VI) shows that the system can provide results with both higher precision and recall than PTQL, which is one of the best existing systems known to the authors. The time performance of our systems is also comparable with that of PTQL.

Preliminaries and Background

A collection of texts prepared for the purpose of linguistic computations is usually called a *corpus*. A common practice in Natural Language Processing techniques is to parse/annotate the corpus and generate tree-based structures containing the grammatical connection of the text. The resulting structures are called *Parsed corpora* or *TreeBanks*. Since the term treebank often implies that the text is manually annotated with the help of linguists, in this study, we use the term *Parse Tree* to indicate that the text does not need to be manually parsed. To understand the parse tree structure and provide a motivating example for our paper, we provide the following example.

Motivating Example

Figure 1 shows two possible parse trees for the sentence *"The elderly lady and gentleman held hands and were safely escorted to their car"*, both taken from a probabilistic parser[1]. As you can see, each parse tree has an associated correctness probability. We labeled words with numbers to identify the position of words in the sentence and to differentiate among words used more than once (*e.g.* *'and*$_4$*'* and *'and*$_8$*'*). Figure 2 graphically depicts Parse Tree 1 from Figure 1. In the parse trees, words are placed at the leaf nodes and are annotated with part-of-speech tags (POS-tags). A set of annotated nodes can additionally be annotated with phrase/clause/sentence tags to indicate that they belong to a phrase, clause, or sentence.

[1] Charniak Parser: www.cfilt.iitb.ac.in/~anupama/charniak.php

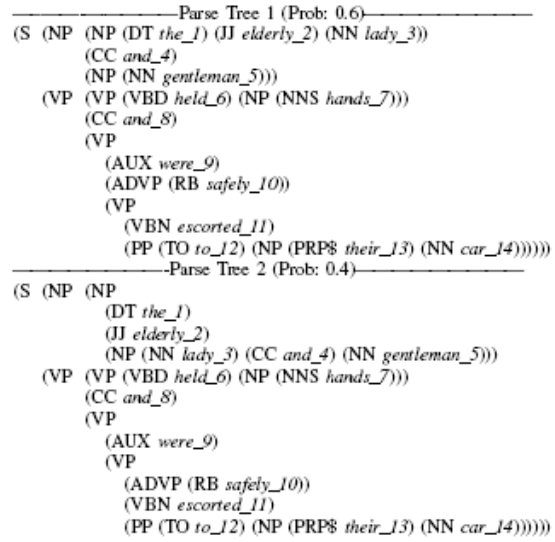

Figure 1. Two parse trees for the sentence "The elderly lady and gentleman held hands and were safely escorted to their car." in parenthesized format.

Addressing Schema

LQL uses a simple addressing schema to identify each node in a parse tree. In this schema, the nodes of the tree are numbered as follows; the root is -1, and then, for each node in the resulting tree, its children nodes are consecutively numbered such that the first child is 0, the second child is 1, and so forth. To address a node in the tree, say n_i, we can list the number of all the nodes in the path from the root to n_i. We skip the root's number since it is always the same, unless we need to address the root itself. For example, all the addresses for Parse Tree 1 are shown on the left side of Figure 2, where addresses [1, 2, 0] and [1, 2, 2, 0] specify the nodes (*AUX were* 9) and (*VBN escorted* 11), respectively. To address multiple nodes that are attached to each other, we can use the addition operation (+). For example, the value of the multi-node address [1, 2, 0] + [1, 2, 2, 0] will be *<were* 9 *escorted* 11>. This form of addressing is called *multi-node* addressing. Note that instead of addressing a node, we can simply use any string.

The Linguistic Query Language (LQL)

In this section, we explain the general structure of the Linguistic Query Language, LQL. The central element of LQL is called a *Rule*. Although all the rules in LQL have the same format, different sets of rules are used for different purposes which will be discussed in this section.

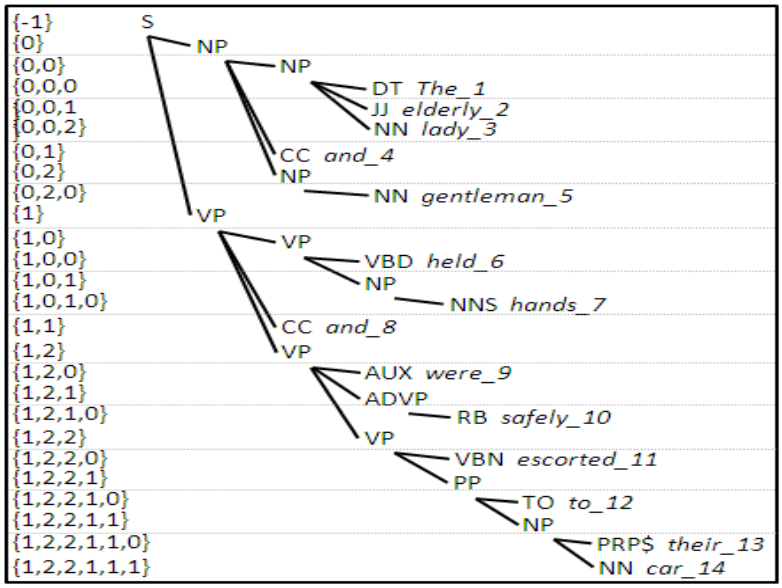

Figure 2. The graphical representation of Parse Tree 1 in Figure 1.

Extraction Rules

Each rule in LQL is composed of two main parts; *Pattern Tree* and *Resulting Format*. Let us start with a simple example: Consider Parse Tree 1 in our running example in Figure 1. Assume that the aim is to find all the adjectives in the sentence and connect them to their appropriate noun through a link labeled '*prop of*', meaning "property of". To do so, one should look for noun phrase (*NP*) nodes in which there is an adjective (*JJ*) node directly followed by a noun (*NN*) node. The following rule can represent such a query. The set of such rules for extracting user-specified pieces of information is referred to as *Extraction Rules*.

Rule 1.

```
RULE adjectiveToNoun {                    PATTERN: (NP
    *
    (JJ ) (NN/NP ))
  RESULT (prob=.90): <[0], 'prop of', [1]> }
```

Given such pattern trees, the query engine finds matches for this pattern over the parse tree(s). In the above example it looks for nodes tagged as *NP* that contain two consecutive children nodes one tagged as *JJ* and the other tagged as either *NN* or *NP*. The latter is done by means of the disjunction operation (/). The asterisk notation (*) in the pattern means that there can be any number of branches (including zero) before the *JJ* node in the graph.

After searching for the above pattern tree, the query engine will find the *matched tree* *(N P (J J elderly 2) (N N lady 3))*. Note that the query engine does not care about other branches (*e.g. (DT the 1)*), since the user has not specified them explicitly in the pattern tree.

Now that the query engine has found the matched tree(s), to extract information we can address nodes (or multi-nodes) in the matched tree in the manner described in Section II. In order to do so, we define the *Resulting Formats* as sequences of one or more multi-node addresses over the matched trees.

These *resulting formats* will be applied over all the matched trees. Getting back to our example, <[0], 'prop of ', [1]> will return the triple we were looking for from Rule 1 (<elderly 2, prop of, lady 3>). The *resulting format* also has some optional parameters. As it is shown in Rule 1, each *resulting format* can have a probability indicating the certainty of the results. We will discuss these parameters in more detail later in this paper.

Whether or not. For some cases, the existence of a branch in the pattern tree may be optional. For example, there may be an adverb before a verb or there may not be one. In such situations, to reduce the number of total rules and capture more tuples with less rules, we introduce the concept of whether or not (?) in LQL. To understand this concept let's continue with an example. Consider Parse Tree 1 in Figure 1 and assume that the goal is to capture the relationships between auxiliary verbs and their main verbs in the sentence. As you can see, there is an adverb between the only auxiliary verb *(AUX were 9)* in the sentence and its main verb *(VBN escorted 11)*. This adverb may not appear in many other similarly structured texts. Rule 2 captures this relationship considering an optional adverb. Note that if no matches are found for an optional node, all the formatting results using that optional node will be ignored (the second formatting result in our example).

Rule 2.

```
RULE auxiliaryToVerb {
   PATTERN: (VP
        (AUX ) (?/RB/ADVP ) (VP
          (VBN )))
     RESULT (FO1='VMP', FO3='AVMP', prob=.93):
     < [0], 'prop of ', [2, 1] >
     RESULT (FO1='NMP', FO3='VMP', prob=.95):
     < [1], 'prop of ', [2, 1] > }
```

Main-Parts Generation Rules

Assume that you want to find the subject-to-verb relation- ships in Parse Tree 1 in Figure 1. The subject(s) and verb(s) of the sentence are respectively inside the noun phrase at address [0] and verb phrase at address [1]. However, it is not easy to pull them out using the simple form of the rules that we explained in the previous section. First, the actual subjects (or verbs) may be deep inside the branches. Second, there are so many possible combinations that each may need a separate rule. Note the number of rules exponentially increases considering the combinations of possible cases for each branch. To alleviate this issue, we introduce the concepts of *Noun*, *Active-Verb*, *Passive Verb*, and *Prepositions* Main-Parts for each node in the parse tree.

Noun Main-Part is defined for nodes related to nouns *(S, NP, NN, NNS, CD, JJ, ADJP, ...)*, and it indicates the actual noun(s) of that node. For example, the noun main-part of the node at address [0, 0] is <'lady 3'> and of the node at address [0, 2] is <'gentleman 5'>. This leads us to noun main-parts <'lady 3', 'gentleman 5'> for our main noun phrase ([0]).

A similar concept is used for the verbs; however, since verbs have two forms, passive and active, we have to have two types of main-parts for verb-related nodes (*S*, *VP*, *VB*, *VBZ*, *VBD*, *VBN*, ...). As an example, the active verb main-part and passive verb main-part of the verb phrase at address [0] are respectively <'held 6'> and <'escorted 11'>. The fourth main- part set is for prepositions. Unlike the other types

of main- parts, preposition main-parts can be for both noun-related nodes and verb-related nodes. This is because prepositions can fall inside either noun phrases or verb phrases.

We will explain the process of extracting the main-parts information later in this subsection, but now we go back to our running example to show how Rule 3 can extract some of the subject-to-verb relationships. As you can see, this rule is very similar to the basic rule introduced earlier, with two slight changes; the use of *Formatting Options* (FOs) for some of the addressed multi-nodes, and the ability to have more than one *resulting format*. Note that the numbers next to FO indicate the item in the *resulting format* to which the option should be applied. Formatting options for each multi-node address indicate what information related to the addressed node(s) should be inserted into the resulting tuples. Currently, we have considered seven formatting options: 'NMP' for Noun Main-Parts, 'AVMP' for Active Verb Main-Parts, 'PVMP' for Passive Verb Main-Parts, 'VMP' for Verb Main-Parts (including all active and passive verbs), 'PMP' for Preposition Main- Parts, 'NT' for Node's Tag in the parse tree, and 'WP' for the Whole Parts (The entire section of the sentence underneath the selected node).

Rule 3.

```
RULE subjectToVerb {
   PATTERN: (S
                                              (NP ) (VP ))
   RESULT (FO1='NMP', FO3='AVMP', prob=.95):
      <[0], 'subj of', [1] >
   RESULT (FO1='NMP', FO3='PVMP', prob=.95):
      <[0], 'obj of', [1] > }
```

The results of this simple query are the following triples all with 95% probability: *<lady 3, subj of , held 6>, <gentleman 5, subj of , held 6>, <lady 3, obj of , escorted 11>, <gentleman 5, obj of , escorted 11>*. This simple preprocessing technique to compute main-parts information of the nodes significantly reduces the total number of rules to capture a particular piece of information from differently structured texts. For instance, for the above query, without the help of main-parts, we would have had to write four different rules to extract those four tuples. Note that although this rule may catch a large portion of the subject-to- verb relationships for different parse trees, it does not cover them all. In other words, to capture all such relationships for any given text several rules may be needed. The concept of main-parts also plays a very important role in the performance of the matching engine since it makes the feature of searching in depth of the parse tree unneeded. This matter is discussed in further detail in Section V.

Generating the main-parts information. To extract main-parts information from the parse trees of the given texts, LQL uses the rules introduced in the previous subsection. The whole idea is to find the dependencies among nodes in the parse trees in a bottom-up technique. An example of such rules is depicted in Rule 4, which says that the noun main- parts of any noun phrase in the parse tree contains (depends on) the noun main-parts of its last child node if the child is a noun-related node, namely *NP*, *NN*, *NNS*, or *NNP*. Note that *!** indicates that after the last matched node there should be no other node in the matched trees. This is necessary to exclude *NP* nodes with more than one noun-related child from the matched trees for our particular example. Note that in the *resulting format* parts of all the dependency rules, there should be exactly two items, and the first item must address a single node. Usually the second item

also addresses a single node; however, as the last portion of this subsection explains, this is not always the case.

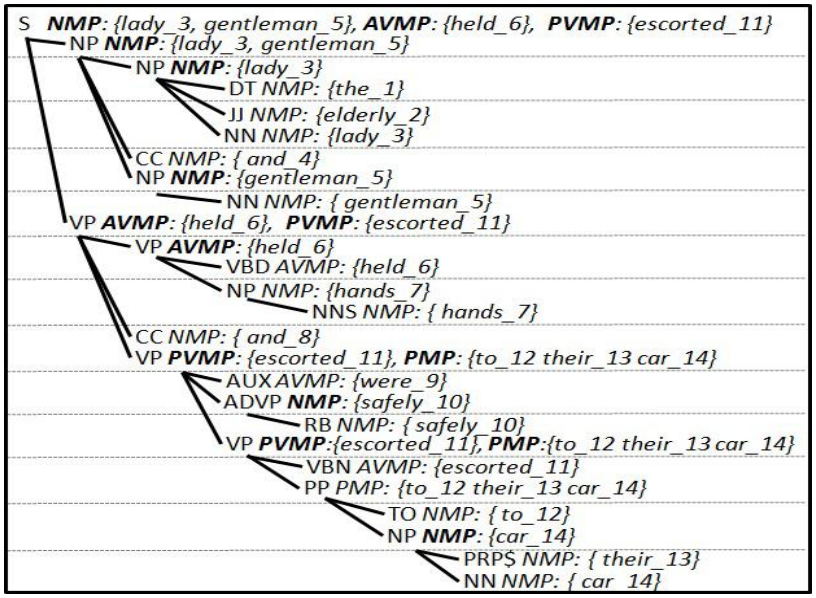

Figure 3. Main-Parts information for Parse Tree 1 in Figure 1.

Rule 4.

```
RULE nounDepRules ('NDR') {
   PATTERN: (NP
         *
         (JJ/ADJP ) (NP/NN/NNS/NNP )
         !*)
   RESULT: < [−1], [1] >
   RESULT: < [−1], [0] + [1] > }
```

We may refer to this type of rules as *dependency rules* as well; however, the dependency concept in this paper should not be confused with those in dependency graphs in Marneffe et al. (2006). The former indicates dependencies between nodes of parse trees, while the latter indicates dependencies between words of the sentences (that is, only leaves in parse trees.) For each type of main-parts, LQL has a separate set of dependency rules. That is, LQL has four sets of dependency rules which are specified by rule types in front of the rule's name: 'NDR' for noun Dependency Rule, 'AVDR' for Active Verb Dependency Rule, 'PVDR' for Passive Verb Dependency Rule, and 'PDR' for Preposition Dependency Rule. Through our studies, by examining a lot of sentences and trying to extract different tuples out of them, we realized that these four types of main-parts are enough in most types of linguistic information extracting processes.

Rules 5 and 6 respectively show example of active verb and passive verb dependency rules. Note that these two rules can pull out most of the existing verb dependencies in both Parse Trees 1 and 2.

Rule 5.

```
RULE activeDepRules ('AVDR') {
   PATTERN: (VP
      (VP/VB/VBD/VBN/VBP/VBZ ) (??/CC ) (??/VP/VB/VBD/VBN/VBP/VBZ ))
   RESULT: < [−1], [0] >
   RESULT: < [−1], [2] > }
```

We should mention that the query engine treats the *resulting formats* of these two types of dependency rules slightly different than those for noun and preposition main-parts. For active verb main-parts, the query engine adds the passive verb main-parts and active verb main-parts of the referenced node (second addressed item in the *resulting format*) respectively to the passive verb main-parts and active verb main-parts of the dependent nodes (first addressed item in the *resulting format*). On the other hand, for passive verb main-parts, the query engine only copies the active verb main-parts of the referenced node to the passive verb main-parts of the dependent node. This will make more sense if we take a closer look at Rule 6. In this rule, the type of the verb at address [2, 0] changes from active to passive when we pull it up to the the verb phrase at address [-1] because of the *to be* auxiliary verb before it.

Rule 6.

```
RULE passiveDepRules ('PVDR') {
   PATTERN: (VP
        (AUX am/is/are/was/were/be/been/being) (??/ADVP/RB )
        (VP
          (VBN )))
   RESULT: < [−1], [2, 0] > }
```

After specification of these sets of dependency rules by linguists, the query engine extracts all the dependency links other hand, in most applications users need to specify several rules in order to extract the information they need. That is, the same piece of information may be extracted more than once either from the same parse tree or from different parse trees with different probabilities.

Considering Rule 1 for Parse Tree 2 in our example sentence in Figure 1, two triples <*elderly* 2, *prop of* , *lady* 3> and <*elderly* 2, *prop of*, *gentleman* 5> will be extracted with the probability of 90%. Note that <*lady* 3, *gentleman* 5> are the noun main-parts of the node at address [0, 0, 2] in this parse tree. As you remember, Rule 1 already found the first triple from Parse Tree 1, but the second one is a new one which had not been extracted from Parse Tree 1. This shows the ambiguity and uncertainty for this piece of information, as it is not clear whether the adjective word '*elderly*' goes only to the '*lady*' as indicated in Parse Tree 1 or it goes to both the '*lady*' and '*gentleman*' as indicated in Parse Tree 2.

To show such ambiguities, we try to combine the probability of the identical results generated from different parse trees. We assume the results are independent random variables. Considering N parse trees each with probabilities $w1$, $w2$,..., wN such that $\sum w_i = 1$ and $w1_i$ is the largest weight, between the nodes in the parse tree and then propagates the main-parts in a bottom-up fashion. For each leaf, the PMP and PVMP parts are set to nil. If a leaf node has a verb-related POS tag (which are *AUX*, *MD*, *VB*, *VBD*, *VBG*, *VBN*, *VBP*, and the probability of a resulting tuple t can be computed in the following ways:

Weighted Union: $p = 1 - \Pi_i(1 - \frac{w_i}{w_1} p_i(t))$ *VBZ*) the AVMP of the node is set to the nodes value and its NMP is set to nil. Otherwise, if the node does not have a verb- related POS tag, its NMP is set to the nodes value and its AVMP is set to nil. Also, the PMP part of those nodes tagged with *PP* (preposition tags) that do not have any descendent node tagged with *PP* are set to that part of the sentences under their branch. After this initialization phase, the query engine repeatedly picks a node whose main-parts of

all its children are already extracted, and propagates the main-parts based on the found dependencies for the node. The main-parts-annotated parse tree for Parse Tree 1 is shown in Figure 3. To create this kind of annotated parse trees, we have generated around 100 dependency rules so far.

Feeding LQL the Concepts List: Since many NLP-based applications may be specialized for a particular domain, LQL allows the users to import the list of terms and concepts that they are interested in. Having a concepts list allows the users to restrict the results to concepts in the domain. However in some cases, the terms in the concepts list can consist of multi-word combinations (*e.g.* 'The United Nations') which should appear as an atomic entity or concept (not three separate nodes). To capture these concepts, the second item of the *resulting formats* in main-parts rules should use a multi-node address. For instance, the second *resulting format* of Rule 4 adds *'elderly lady'* to the noun main-parts of the *NP* node with address [-1], only if *'elderly lady'* is in the user specified concepts list.

Supporting Ambiguity

As already discussed, each *resulting format* is associated with a probability. These probabilities show the confidence of the users about the correctness of the generated results. On the *Weighted Mean:* $p = \sum w_i p_i(t)$

Where pi (t) is the probability of finding *t* in parse tree *i*. The idea behind the union-based technique is to boost up probabilities when more evidence is observed. In other words, the probabilities are interpreted as chances for these cases. For instance, the probability of having *<elderly 2, prop of , lady 3>* as a result in our running example would be 1-(1-.9)(1-(.4/.6).9)=.96. However, the probability of <elderly 2, prop of, gentleman 5> will be 1-(1-0)(1-(.4/.6).9)=.6. On the other hand, in the mean-based method, we try to report the average probabilities, meaning that we care about the negative part of probabilities as much as the positive part. The choice between these aggregation techniques completely depends on the way the rules are set up and the interpretation of the probabilities.

Supporting exceptions. Another important feature of LQL is the support for linguistic exceptions. As already mentioned, in most applications several rules must be defined in order to extract any particular type of information. However, due to the nature of natural languages, finding rules without exceptions may be very hard. To address such an issue in LQL, we simply define the *resulting formats* of the exception rules with negative probabilities. This way, the same probability combinations can be used without any changes. The only point to consider is that exception rules are superior to regular rules in the same parse tree.

Application: Generating TextGraphs

In text mining applications such as reviewing systems, text summarization techniques, text categorization systems, content-based search engines, and automated essay scoring systems, it is easier and more efficient to work on a graph showing various types of relationships between words and concepts of the given text. This way, more accurate results can be provided, since more relationships are explicitly presented in the graph than in the text. Moreover, concepts such as Anaphora resolution and co-

referencing resolution can be addressed more easily in graph-based structures than in texts. In the context of these applications and with the use of the rules explained so far, one can extract relationships in a given text as triples and connect them to make a graph. The graph generated in this way is referred to as a *textGraph*. At the time of writing this paper, we have already created more than 150 rules for generating textGraphs. Thus, for each given textual data item, we extract the set of triples using these rules. Each triple is interpreted as an edge in which the first and the third items are the two ends of the edge and the second item is the edge's label. By connecting these triples, we can construct the final textGraph for each textual data item. Although we believe that these rules do not extract all the possible relationships in texts, they still provide an acceptable number of relationships.

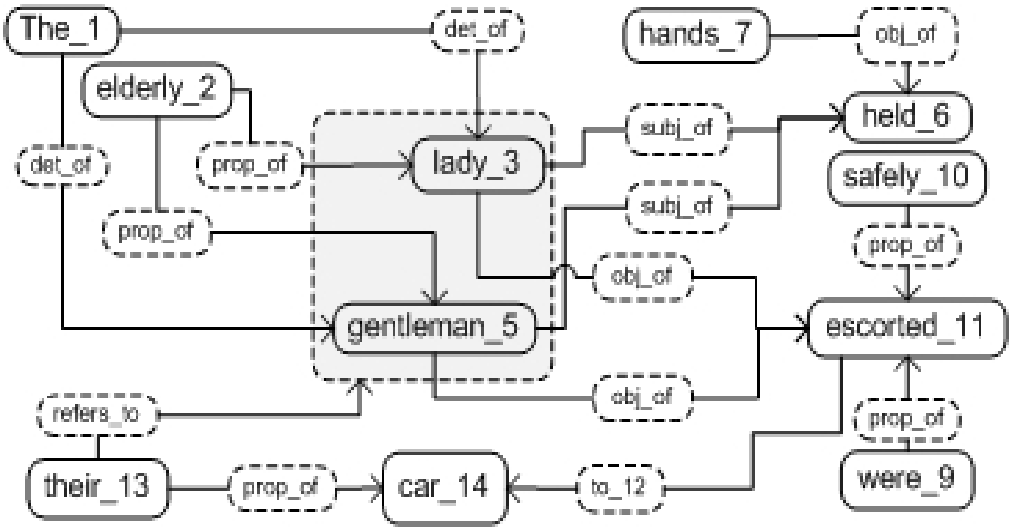

Figure 4. A possible textGraph for the example sentence in Figure 1 generated using LQL rules.

Figure 4 shows the textGraph generated for our running example's sentence. Although this graph is similar to the dependency graph in Marneffe et al. and linkage graph in Sleator and Temperley (1995), there are a few essential differences: i) While those two are plane graphs, textGraph is a hyper-graph meaning that the links can connect sets of nodes as well as single nodes, ii) TextGraphs contain the probability of the links to be correct, which is essential to support ambiguity and more accurate results, and iii) Despite dependency and linkage graphs, nodes in textGraphs can be multi-word concepts. This way, textGraphs can be specialized for different domains.

LQL Framework and Its Query Engine

Figure 5 depicts the high-level architecture of our frame- work. As the figure indicates, the framework separates tasks into two main phases: preprocessing, and information ex- traction. This way, time-consuming and one-time tasks such as generating parse trees and main-parts information can be performed in the preprocessing phase, so that users' queries and mining tasks can run more quickly. Next, we explain the main components of the framework.

Text2Sntc: As the name indicates, this component breaks the raw input text into paragraphs and sentences. This is done by ignoring periods ('.') for the abbreviated nouns and some special cases, and mainly looking at the remaining periods (and other end of sentence indicators).

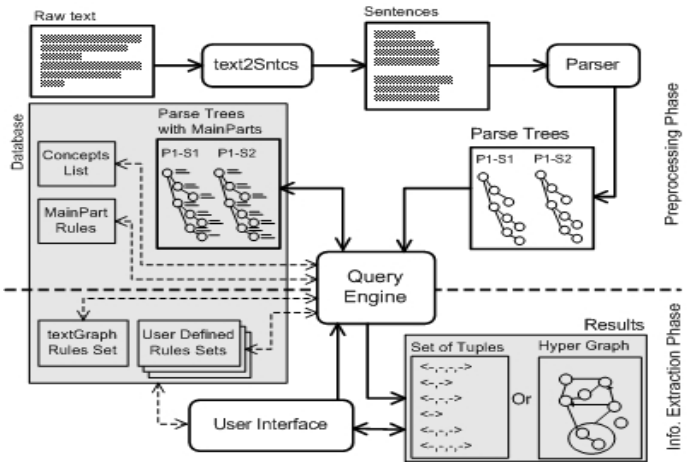

Figure 5. The high level architecture of LQL system.

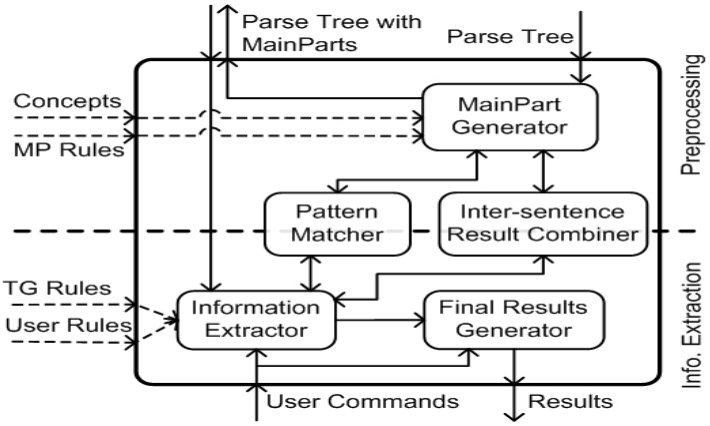

Figure 6. The main structure of LQL's query engine.

Parser

For each found sentence, we use a probabilistic parser (Charniak) to generate its parse tree(s). Users can specify how many parse trees should be used.

Query Engine

Query engine has two main jobs; generating the main-parts and information extraction from a given set of rules. These two tasks are explained in next few paragraphs. Due to the importance of this component, we have provided more details about the modules in this component in Figure 6. The most important module in this component is the *Pattern Matcher*. Given an LQL pattern and a parse tree (with main- parts), *Pattern Matcher* finds the matches for the pattern in the parse tree. Current implementation of the *Pattern Matcher* naively searches for all possible matches with some heuristics.

- *Generating main-parts:* To annotate parse trees with main-parts information, each parse tree is sent to the *MainPart Generator*. Using the *Concepts List* and the set of *MainPart Rules*, this

component generates all the possible dependencies by using the *Pattern Matcher*. The resulted dependencies are sent to the *Inter-sentence Result Combiner* which combines duplicates and removes exceptions. Finally, *MainPart Generator* populates the dependencies to generate the main-parts for each node in the parse tree. The main-part-annotated parse tree is then stored in the database.

- *Information Extraction from Set of Rules:* Using the results of the previous process, (which is usually done in the pre-processing phase), LQL is capable of extracting information using a set of rules. That is, given a set of LQL rules, it finds matches for each rule in all (or part of) the main-part-annotated parse trees using the *Pattern Matcher*. The generated results for each parse tree is then sent to the *Inter-sentence Result Combiner* which combines duplicates and removes exceptions. Once the results are generated for all the parse trees, they are sent to the *Final Result Generator*. This module then combines the duplicate results based on their weights, filters them using the concepts list[2], and generates the final results in the form of a set of individual tuples or a hyper-graph.

rule:
RULE ruleName [(**ruleType**)]
{ *PATTERN:* **pattern-tree**
formatting-result^{+} }
ptrn-tree:
(**tags** [*] **ptrn-tree***)
or (**tags** [*] **ptrn-tree**$^{+}$!*)
or (**tags words**)
or (**tags**)
tags:
? { | tag } *
or tag { | tag } *
or !tag { | !tag } *
or *
words:
? { | word } *
or word { | word } *
or !word { | !word } *
formatting-result:
RESULT [(**parameters**)]:
< **addresses** >
addresses:
multi-addr {, **multi-addr** } *
multi-addr:
node-addr {+ **node-addr** } *
node-addr:
[integer {, integer } *]
parameters:
parameter {, **parameter** } *
parameter:
*FO*integer = **frmt-optn**
or *PROB* = float
ruleType:
'NDR' or 'AVDR'
or 'PVDR'or 'PDR' or 'ER'
frmt-optn:
'NMP' or 'AVMP'
or 'PVMP' or 'VMP'
or 'PMP' or 'NT' or 'WP'

Fig. 7. The grammar of LQL rules.

User Interface

The access point of users to the system. It basically provides utilities for the users to configure the system and its components, to generate new rules (of any type), to specify the domain(s) and its related concepts list, to add more text to the system, and to control the query engine.

It is important to mention that a predefined set of more than 150 rules are already in the system for generating the textGraph of the input text as described in Section IV.

Evaluation

In this section, we evaluate the performance of our system by implementing a relation extractor utility. To this end, we manually generated a set of 38 LQL rules to extract two types of relations ('type-of'

[2] Users can specify if they want the results containing only concepts, or results with at least one concept is in them, or all the results.

and 'part-of') between math concepts. We have compared our results with those from PTQL (Tari et al., 2010) which is one of the best existing parse tree mining systems. The framework is partially implemented using the Python language with a web-based user interface. We are also storing the sets of rules, concepts list, and the parse trees with main- parts in MySQL. All the experiments are run on a 32bit, Intel Core 2 Due 2.53GHz CPU machine running Linux with 2GB of main memory (RAM) and 4MB of cache.

To create these rules, we used a Math ontology manually created by CRESST over last few years. We should mention that this ontology is not a complete math ontology. It is mainly intended to contain all the concepts used in the *Common Core State Standards* and hierarchical links between those concepts. Thus, several concepts and many aliases and links are missing in this ontology; However, it was a valuable resource for generating the LQL rules. This ontology contains 877 entities (concepts) with 251 aliases, 586 original links/relations, and 869 links after population (we added $<X, L, Z>$ to the set of links if $<X, L, Y>$ and $<Y, L, Z>$ are already in the set of links.). The average number of words per concept is 2.18. Each individual concept in this ontology is also associated with a short definition in one or two sentences. These concepts are fed to the system as the concepts list.

Table I

The Results of Searching for Math Relations

Query Language	Links No.	After population	Precision	Recall
LQL-CRESST	1903	5473	87.8%	72.8%
LQL-Wiki	1187	4101	76.6%	32.3%
PTQL	-	-	83.6%	58.6%

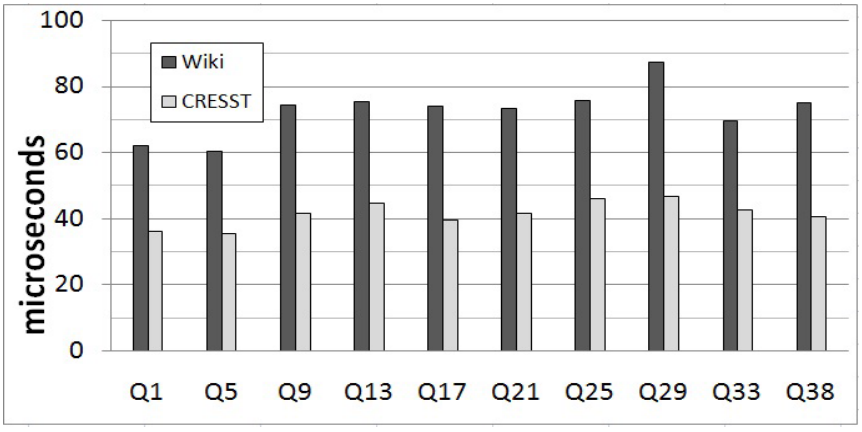

Figure. 8. The average running time per sentence for 10 selected queries. (Q1 and Q38 are the queries which generate respectively the maximum and the minimum number of results).

Considering two parse trees for each sentence, we ran our rules over two different textual datasets; the definitions in CRESST ontology, and the the definitions for the same set of concepts from Wikipedia's

latest repository (Available at: http://dumps.wikimedia.org/). The average number of words per definition in CRESST and Wiki are respectively around 34 and 91. Note that we have only used the definitions and links in the CRESST ontology to generate the LQL rules, and did not make any rule specifically for the texts taken from Wiki.

Table I summarizes the results of this experiment. As you can see, the number of generated links for each dataset before and after population is shown in the first two columns. To compute the precision (shown in third column of Table I), we asked a mathematics expert to go over all the originally generated links and count the number correct ones. However, to estimate the recall (fourth column), we computed the ratio of links that each approach covers from the set of manually generated links by CRESST. Both precision and and recall for CRESST definitions is better than those for Wiki. This is mainly because the LQL rules are specifically generated for the former one and more importantly for almost half of the concepts in CRESST, Wiki uses different terms which greatly affects the recall value for Wiki. We have also included the precision and recall values for PTQL in Table I, which shows that LQL outperforms PTQL with respect to both factors for the CRESST's dataset.

Time performance

To evaluate the time performance of our query engine, we have selected 10 queries and included the average time needed to answer each query for a given parse tree in Figure 8. To select these 10 queries, we ordered them based on the frequency of their matched results and uniformly picked 10. As the figure indicates, queries have similar running times for each dataset; However, pattern matching takes more for Wiki's sentences, since they are almost twice longer than those for CRESST. In total, the aggregate pattern matching time for 38 queries over 2 parse trees for every sentence in CRESST and Wiki datasets respectively takes about 12.2 and 33.0 seconds. These are comparable with query answering times reported in PTQL which may take more than 50 seconds to report only 10 results for a single query. Note that similar to PTQL, we have not incorporated parsing, main-parts anno- tation, and other mining tasks' time to our timings.

Related Work

In the last few years, many querying languages and information extraction techniques for parse tree structures have been proposed, a few of which include: Emu, the Annotation Graph query language, TGrep2, TIGERSearch, DIAL, NXT Search, Emdros, KnowItNow, LPath, MEDIE, XQuery based querying language, AQL, and PTQL.

Several of these systems attempt to extract information from textual data, without considering the full grammatical structure provided by parse trees. Robinson provided a context-free grammar, called DIAGRAM, that works only for part of the English grammar. Unlike LQL, these approaches can not efficiently handle issues such as ambiguity, exceptions, and grammatically incorrect texts.

Several of the other works have tried to use or extend existing query languages for linguistic queries. Bird et al. proposed a new query language for XML documents, called LPath, by extending XPath. Although they showed that LPath is more expressive than XPath in terms of linguistic queries, XPath-based techniques are not designed for extraction. Following a similar approach, Bouma et al. used XQuery to mine information from parse trees. As they also mentioned, the linguistic queries expressed by XQuery may be very complex and generating them may go beyond general knowledge of XML. Additionally in their techniques, dealing with the order of the items in patterns is not very common sense. Miyao et al. proposed an XML-like query language for parse trees stored in a database. However, their language is limited to subject-verb-object links, can not express link types, and is best optimized for biomedical texts. Only a few of these works designed their own query languages to operate over the whole parse tree structure. Using rules similar to the head-driven rules defined in Collins (2003), Marneffe et al. proposed an approach to extract information about a set of predefined dependent relations between words in texts. However, in their approach, the mined dependencies can not be used in other extraction rules. Recently, Tari et al. presented a new linguistic query language called PTQL. PTQL is designed for information extraction from parse trees stored in a relational database called PTDB. PTQL is similar to LQL in many aspects; However, it needs far more text or a many more rules to extract the same amount of information. Moreover, neither Marneffe et al. or Tari et al. support probabilistic results, exceptions, and ambiguities.

Conclusion

We introduced a new text mining framework which is based on a tree-based Linguistic Query Language, called LQL. The framework enriches the parse tree(s) generated for each sentence in a given text with *main-parts* information which is set of key terms from the node's branch based on the linguistic structure of the branch. The presence of main-parts has made the LQL's design simpler, and as a result query answering and text mining can be performed more quickly. The framework also supports grammatical ambiguity and linguistic exceptions utilizing probabilistic rules. It can also adapt to different domains using a pre-defined list of concepts. Our evaluation shows that the matching time for the query engine is comparable with one of the best existing systems (PTQL; Tari et al.), while being able to extract more information form smaller set of dataset. In the future, we mainly plan to complete the main-parts and textGraph rules set and optimize the current implementation of the framework.

References

Bird, S., Buneman, P., & Tan, W.C. (2000). Towards a query language for annotation graphs. *CoRR*, cs.CL/0007023.

Bird, S., Chen, Y., Davidson, S.B., Lee, H., & Zheng, Y. (2006). Designing and evaluating an xpath dialect for linguistic queries. In *ICDE*, page 52.

Bouma, G., & Kloosterman, G. (June 2007). Mining syntactically annotated corpora with xquery. In *Proceedings of the Linguistic Annotation Workshop*, pages 17–24, Prague, Czech Republic, Association for Computational Linguistics.

Bunescu, R.C., & Mooney, R.J. (2005). A shortest path dependency kernel for relation extraction. In *HLT/EMNLP*.

Cafarella, M.J., Downey, D., Soderland, S., & Etzioni, O. (2005). Knowitnow: Fast, scalable information extraction from the web. In *HLT/EMNLP*.

Cassidy, S., & Bird, S. (2000). Querying databases of annotated speech. In *Australasian Database Conference*, pages 12–20.

De Marneffe, M.-C., Maccartney, B., & Manning. C.D. (2006). Generating typed dependency parses from phrase structure parses. In *LREC 2006*.

Chamberlin, D., Clark, J., Florescu, D., Robie, J., Siméon, J., & Stefanescu, M. (December, 2001). *XQuery 1.0: An XML Query Language*. Technical Report W3C Working Draft, World Wide Web Consortium.

Clark, J., & DeRose, S. (November 1999). Xml path language (xpath).

Collins, M. (2003). Head-driven statistical models for natural language parsing. *Computational Linguistics*, 29(4):589–637.

Feldman, R., Regev, Y., Hurvitz, E., & Finkelstein-Landau, M. (May, 2003). Mining the biomedical literature using semantic analysis and natural language processing techniques. 1:69–80.

Heid, U., Voormann, H., Milde, J.-T., Gut, U., Erk, K., & Pad, S. (2004). Querying both time-aligned and hierarchical corpora with nxt search. In *Fourth Language Resources and Evaluation Conference*, Lisbon, Portugal.

Lezius, W., Biesinger, H., & Gerstenberger, C. (2002). Tigersearch manual.

Miyao, Y., Ohta, T., Masuda, K., Tsuruoka, Y., Yoshida, K., Ninomiya, T., & Tsujii, J. (2006). Semantic retrieval for the accurate identification of relational concepts in massive textbases. In *Proceedings of the 21st ICCL and the 44th annual meeting of the ACL*, ACL-44, pages 1017–1024, Stroudsburg, PA, USA.

Petersen, U. (2004). Emdros – a text database engine for analyzed or annotated text. In *COLING*, pages 23–27.

Reiss, F., Raghavan, S., Krishnamurthy, R., Zhu, H., & Vaithyanathan, S. (2008). An algebraic approach to rule-based information extraction. In *ICDE*, pages 933–942.

Robinson, J.J. (2005). Diagram: A grammar for dialogues. *Commun. ACM*, 25(1):27–47, 1982.

Rohde, D.L.T. (2004). *Tgrep2 user manual*. Retrieved from: http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf

Sleator, D.D., & Temperley, D. (1995). Parsing english with a link grammar. *CoRR*, abs/cmp-lg/9508004.

Tari, L., Tu, P., Hakenberg, J., Chen, Y., Son, T., Gonzalez, G., & Baral, C. (2010). Parse tree database for information extraction. In *IEEE Transactions on Knowledge & Data Engineering*.