CIS6930 Project Proposal

ProbKB: Managing Web-Scale Evolving Knowledge

Yang Chen (1442-5921), Xing Liu (3066-8722)

Introduction

Statistical relational learning is one important sub-discipline in artificial intelligence and machine learning. It can be applied in text analysis to extract valuable information. Knowledge from text data can be represented as first order logic and can be inferred from existing knowledge using statistical relational learning. The goal of this project is to use certain statistical relational learning techniques to analyze and discover knowledge from web-scale text data.

However, managing web-scale knowledge is challenging:

- Not only do we need to achieve scalability in our application to manage huge amount of information from the web, but also apply efficient algorithm and the right tools to do inference, and to store knowledge discovered from our analysis in large scale.
- Second, we shall use proper data corpus containing existing first logic rules and text data extraction from the web to implement our knowledge inference system. Moreover, we need to find a model capable of handling uncertainty and incorrectness in raw internet text data for knowledge inference from a large amount of informations.

In this project, we will use extracted text data from wikipedia and a set of horn-clause rules to implement a knowledge base to store existing knowledge with an inference engine. Markov Logic Network, combining logic and probability, is the model applied to represent the inference problem in this project. We will first initialize the markov logic network where every vertex represents a grounded first order logic and is also a random variable. Using horn-clause rules, we can infer new grounded first order logic, or knowledge and add them to the MLN which represents many possible world. By applying certain algorithms like MCMC, we will explore the probabilistic distribution of these first order logics and utilize these results to obtain a set of possible high-confidence knowledge.

Phase I Markov Logic Network (MLN) Implementation on Datapath

1. Schema

The knowledge base schema is designed with fast grounding and inference in mind. The most popular way to represent a knowledge base in a relational model, as used by NELL and Tuffy (cite), is to store each relation in a separate table. This design makes grounding very difficult if we have many relations and rules since we have to process them one by one. Our approach is to store all relation instances in a single table so that we can use a single (at most a few) relational operators to process all of them.

The following pseudocode illustrates our design: we have a few tables to store entities, relations, and some meta-data. We store the whole MLN in 10 tables according to their characteristics (length, argument order, etc). An inference step can be performed by joining these tables as

discussed in Section 2.

```
#########################################################################
# Entities & Relations                                                  #
#########################################################################
Classes(class#, name)
Entities(ent#, name)
Predicates(pred#, name)
Instances(class#, ent#)       # index on class#
Relations(pred#, ent1#, ent2#, weight)        # no index since it changes
Relations(pred#, ent1#, class1#, ent2#, class2#, weight)

#########################################################################
# MLN Rules (Inference Tables)                                          #
#########################################################################
MLN1(head#, body#, class1#, class2#, weight)
    # p(x: c1, y: c2) := q(x: c1, y: c2)
MLN2(head#, body#, class1#, class2#, weight)
    # p(x: c1, y: c2) := q(y: c2, x: c1)
MLN3(head#, body1#, body2#, class1#, class2#, class3#, weight)
    # p(x: c1, y: c2) := q(x: c1, z: c3), r(y: c2, z: c3)
MLN4(head#, body1#, body2#, class1#, class2#, class3#, weight)
    # p(x: c1, y: c2) := q(x: c1, z: c3), r(z: c3, y: c2)
MLN5(head#, body1#, body2#, class1#, class2#, class3#, weight)
    # p(x: c1, y: c2) := q(z: c3, x: c1), r(y: c2, z: c3)
MLN6(head#, body1#, body2#, class1#, class2#, class3#, weight)
    # p(x: c1, y: c2) := q(z: c3, x: c1), r(z: c3, y: c2)
MLN7(head#, body1#, body2#, class1#, class2#, class3#, weight)
    # p(x: c1, y: c2) := q(y: c2, z: c3), r(x: c1, z: c3)
MLN8(head#, body1#, body2#, class1#, class2#, class3#, weight)
    # p(x: c1, y: c2) := q(y: c2, z: c3), r(z: c3, x: c1)
MLN9(head#, body1#, body2#, class1#, class2#, class3#, weight)
    # p(x: c1, y: c2) := q(z: c3, y: c2), r(x: c1, z: c3)
MLN10(head#, body1#, body2#, class1#, class2#, class3#, weight)
    # p(x: c1, y: c2) := q(z: c3, y: c2), r(z: c3, x: c1)
```

Question:

- Should I assign sequence ID's? If so, How should I do that? Inside the database? Right now I'm using Python.

Discussion:

- Unmanageable if we have longer rules. How can we generalize this idea by using another table that encodes the argument?

2. Grounding

Grounding is hard for two reasons:

- From an algorithmic point of view, a complete grounding of a first-order logic system consists of all possible substitutions of free variables by constants (entities), which is intractable when the number of entities is large.
- From a system implementation point of view, we need to pull all the data out of the file system (either a normal file system or a database), requiring much I/O if the numbers of relations and rules are large.

We decided to restrict ourselves to Horn-clauses to minimize computation. Although Horn-clauses are not able to express complex concepts like FORALL or EXISTS, for our purpose of *discover* knowledge, we argue they are sufficient since we do not need constraints. In addition, Horn-clauses are easy to learn (site Sherlock paper) and allow efficient and space-saving grounding algorithms.

With this restriction, we will be able to encode the MLN (with up to 31,000 rules) in ten tables. This allows us to ground MLN rules one type at a time. Each round of grounding can be completed with ten joins. Example grounding queries are shown below.

Example SQL queries used for grounding:

Rule: `p(x: c1, y: c2) => q(x: c1, y: c2)`
SQL:

```
INSERT INTO Relations
  SELECT 0.0, I1.pred2#, Relations.ent1#, Relations.ent2#
  FROM Relations JOIN I1 ON
    Relations.pred# = I1.pred1#
  JOIN Instances Inst1 ON
    Inst1.ent# = Relations.ent1# AND Inst1.class# = I1.class1#
  JOIN Instances Inst2 ON
    Inst2.ent# = Relations.ent2# AND Inst2.class# = I1.class2#
```

Rule: `p(x: c1, y: c2), q(x: c1, z: c3) => r(y: c2, z: c3)`
SQL:

```
INSERT INTO Relations
  SELECT 0.0, I3.pred3#, R1.pred2#, R2.pred2#, 0.0
  FROM Relations R1 JOIN I3 JOIN Relations R2 ON
    R1.pred# = I3.pred1# AND R2.pred# = I3.pred2# AND R1.ent1# = R2.ent1#
  JOIN Instances Inst1 ON
    Inst1.ent# = R1.ent1# AND Inst1.class# = I3.class1#
  JOIN Instances Inst2 ON
    Inst2.ent# = R1.ent2# AND Inst2.class# = I3.class2#
  JOIN Instances Inst3 ON
    Inst3.ent# = R2.ent2# AND Inst3.class# = I3.class3#
```

Moreover, we'll use the following two strategies to parallelize the grounding process:

- Use a distributed database like Greenplum or Titan. With either approach, we need to carefully design a schema to represent our knowledge base.
- Use GLAs to gradually build large factor graphs by merging smaller ones. This approach allows us to build a GLA-GIST pipeline to solve MLN inference. The GLA-GIST pipeline demonstrates a common pattern to design machine learning systems: model building and computing similar to GraphLab's GraphBuilder-Inference architecture.

Challenges:

- Can we parallelize across multiple grounding iterations?
- Deduplicate?
- How do we merge two factor graphs?

3. Inference

The inference is done by running MCMC GISTs on the grounded factor graph. The GIST is a programming model provided by the Datapath system to. It receives or generates an initial set of data, and then performs rounds of transitions upon that state until it converges to the desired result. The differences between GIST and GraphLab models are:

- The GIST supports a more general execution model that allows users to schedule tasks. This is fundamental to general MH algorithms and crucial to query-aware machine learning algorithms.
- [Optional] More flexible locking scheme. Sometimes a relaxed locking scheme will increase performance while only neglectably affecting computation results (Hogwild!). The GIST update function allows you to specify your own locking scheme so as to trade off quality and performance.

Challenge:

- [Hard] How can we relax the consistency model to further improve efficiency?

Phase II Datapath for Incremental Inference

One advantage of the MH inference algorithm is that users are allowed to define arbitrary proposal functions to explore the space. Researchers already began to investigate this property and developed a query-aware inference algorithm (\cite query-aware MCMC). Focused computations in a broader context of graphical models also show the promise of this approach (cite). However, existing large-scale machine learning infrastructures (cite) are not particularly good at this query-aware style since it requires very general and flexible execution scheduling. For example, the execution model of GraphLab requires vertices to schedule their neighbors and disallows global scheduling.

We'll adapt query-aware inference to our incremental setting: given a graphical model that (almost) converges and a set of new variables, we focus inference on the new variable set rather than the entire model. We believe that Datapath is suitable for this task since it supports user scheduling. We'll evaluate our approach in terms of both performance and convergence

rate.

Challenges:

- How do we parallelize Breadth-First-Search? Or do we limit the depth?
- How do we partition to preserve influence function?
- [Hard] Can we apply this technique to build a grounding-inference pipeline?

Phase III Experiments (Design)

Our experiments will focus on showing the benefits of our approach in terms of performance and quality. We'll evaluate how GLAs help improve the efficiency of grounding, how query-aware strategy helps speed-up convergence, and how GISTs help improve inference. We'll also show that the new facts discovered by MLN inference is of practical value.

Performance (time) - Grounding

Graph size	Naive	GLA	Tuffy
25%			
50%			
75%			
100%			

Table 1: Time to finish grounding for different grounding strategies.

Performance - Inference

Graph size	Naive	GIST	GIST-Incremental	Tuffy	Chromatic Sampler	Splash Sampler
25%						
50%						
75%						
100%						

Table 2: Time to generate N joint samples for different inference algorithms. N should be chosen to ensure convergence. Convergence is defined to be a state where the total marginal errors are within a pre-specified value \epsilon.

Performance - Incremental Inference (with GIST)

Graph size (or Ratio?)	Naive	Incremental
25%		
50%		
75%		
100%		

Table 3: Time for convergence. Both naive and incremental inference algorithms are run parallely with GIST enabled so that this table should show improvement over the previous one.

Quality

In order to show that MLN inference over our dataset actually discovers valuable knowledge, we wish to justify the following:

- High-confidence facts are really correct.
- There are inference patterns that lead to *low-confidence facts*. These facts are derived from multiple inference steps, but do not have enough supporting evidence. They're likely to be of interest so we should keep them in our knowledge base and wait for more evidence. We wish to show:
    - There are such facts.
    - These facts are potentially correct.

We first summarize the number of new facts we found, then for the above cases, we sample a subset of our derived knowledge, check them manually or through Amazon Mechanical Turk, and record the accuracy.

Challenge:

- Duplication

(Optional) Phase IV Demo

Build a web interface to allow users to browse/search our knowledge-base.

Appendix

Grounding SQL

```
INSERT INTO Relations(pred, ent1, ent2)
SELECT mln1.head AS pred, r.ent1 AS ent1, r.ent2 AS ent2
FROM relations r, mln1
WHERE r.pred = mln1.body
  UNION
SELECT mln2.head AS pred, r.ent2 AS ent1, r.ent1 AS ent2
FROM relations r, mln2
WHERE r.pred = mln2.body
  UNION
SELECT mln3.head AS pred, r1.ent1 AS ent1, r2.ent1 AS ent2
FROM relations r1, mln3, relations r2
WHERE r1.pred = mln3.body1
AND r2.pred = mln3.body2
AND r1.ent2 = r2.ent2
  UNION
SELECT mln4.head AS pred, r1.ent1 AS ent1, r2.ent2 AS ent2
FROM relations r1, mln4, relations r2
WHERE r1.pred = mln4.body1
AND r2.pred = mln4.body2
AND r1.ent2 = r2.ent1
```

```
  UNION
SELECT mln5.head AS pred, r1.ent2 AS ent1, r2.ent1 AS ent2
FROM relations r1, mln5, relations r2
WHERE r1.pred = mln5.body1
AND r2.pred = mln5.body2
AND r1.ent1 = r2.ent2
  UNION
SELECT mln6.head AS pred, r1.ent2 AS ent1, r2.ent2 AS ent2
FROM relations r1, mln6, relations r2
WHERE r1.pred = mln6.body1
AND r2.pred = mln6.body2
AND r1.ent1 = r2.ent1
  EXCEPT
SELECT pred, ent1, ent2 FROM Relations;
```