Towards Building A Domain Agnostic Natural Language Interface to Real-World Relational Databases

Sree Harsha Ramesh, Jayant Jain, Sarath K S, and Krishna R Sundaresan
Surukam Analytics
Chennai
{harsha,jayant,sarath,krishna}@surukam.com

Abstract

In this paper we present Surukam-NLI — a novel system of building a natural language interface to databases, which composes the earlier work on using linguistic syntax trees for parsing natural language queries with, the latest advances in natural language processing such as distributed language embedding models for semantic mapping of the natural language and the database schema. We will be evaluating the performance of our system on a sample online transaction processing (OLTP) database called as AdventureWorksDB and show that we achieve partial domain independence by handling queries about three different scenarios — Human Resources, Sales & Marketing and Product scenarios. Since there is no baseline for query performance on OLTP databases, we report f-measure statistics on an internally curated query dataset.

1 Introduction

Natural language interfaces to databases (NLIs) are front-ends to a database which help casual users query the database even without knowing the schema of the database or any specialised query languages like Structured Query Language (SQL). This is especially useful for people in decision making roles who largely depend on analysts for their daily dose of reports. For reasons that we'll see in the following sections, we are still far from building a truly domain agnostic system that converts any natural language query successfully into SQL, given just the information about the database schema.

1.1 Background

Building a natural language interface to relational databases is almost as old as the concept of relational databases itself. Codd (1970) defined relational database as a digital database whose organization is based on the relational model of data in his seminal paper of 1970, while the earliest documented Natural Language Interface(NLI) is the Lunar Sciences Natural Language Information System (LSNLIS), (Woods et al., 1972). It was a question answering system built in 1972, that enabled lunar geologists to query the data collected during the Apollo missions.

Despite there being many NLIs since the LSNLIS such as PRECISE, PARLANCE, NaLIR, SEEKER and TEAM (Popescu et al., 2003; Bates, 1989; Li and Jagadish, 2014; Smith et al., 2014; Grosz et al., 1987), there has not been an encouraging adoption of this technology in the software industry, probably because of lengthy configuration phases and domain portability issues. SQL is still the preferred mode of querying relational databases which have highly complex architecture and for sensitive operations like inserting, updating and deleting data.

For a broad class of semantically tractable natural language questions, PRECISE described in (Popescu et al., 2003) is guaranteed to map each question to the corresponding SQL query. PRECISE determines whether a question is semantically tractable using max-flow algorithm and outputs the corresponding SQL query. It collects max-flow solutions corresponding to possible semantic interpretations of the sentence and discards the solutions that do not obey syntactic constraints and generates SQL queries based on remaining solutions. PRECISE is only effective on semantically tractable questions in which sentence tokenization results in distinct tokens which contain

D S Sharma, R Sangal and A K Singh. Proc. of the 13th Intl. Conference on Natural Language Processing, pages 305–314,
Varanasi, India. December 2016.

at-least one wh-word[1]. So, queries like *list the highest selling product* and *show me all the goods purchased in the past week* would not be handled due to lack of wh-words.

Li and Jagadish (2014) present NaLIR which is an interactive natural language query interface for relational databases. The system contains a query interpretation part, an interactive communicator and a query tree translator. The query interpretation part is responsible for interpreting the natural language query and representing the interpretation as a linguistic parse tree. By communicating with the user, interactive communicator ensure that the interpretation process is correct. The query tree, possibly verified by the user, is translated into an SQL statement in the query tree translator and is then evaluated against an RDBMS. Although, this NLI correctly interprets complex natural language queries across a range of domains, the use of a conversational agent in the pipeline, in order to resolve the ambiguities in linguistic parse trees and query interpretation, precludes casual users who do not have the knowledge of the underlying schema and linguistic representation, from using the tool.

The approach described in (Pérez, 2016) proposes a semantically-enriched data dictionary called as Semantic Information Dictionary (SID) which stores the information necessary for the NLI to correctly interpret a query. The performance of the interface depends on the quantity and quality of semantic information stored in the domain dictionary. A translation module, which consists of components for lexical analysis, syntactic analysis, semantic analysis generates the SQL representation of the natural language query. This approach has a lengthy customization phase where a proficient customizer would need to fine-tune the NLI using the domain editor provided with the tool, similar to the approach adopted by Access ELF (Elf, 2009), one of the few surviving commercial NLIs. Yet another NLI that requires a domain and linguistic expert for configuring the NLI is C-Phrase (Minock, 2010). C-Phrase is a natural language interface system that models queries in an extended version of Codds tuple calculus, which are then automatically mapped to SQL. The NLI author would have to use synchronous context free grammars with lambda-expressions to represent semantic grammars. The given grammar rules may be used in the reverse direction to achieve paraphrases of logical queries by configuring the parser and generator to associate phrases with database elements.

As detailed in (Androutsopoulos et al., 1995), NLIs have an ambiguous linguistic coverage and are prone to tedious and lengthy configuration phases. Sometimes, they also have an additional requirement of having a highly specialized team of domain experts and skilled linguists who are capable of creating grammar rules. Motivated by the lack of an easily configurable tool that handles multiple domains with reasonable accuracy we propose Surukam-NLI[2], a domain-agnostic NLI that has an automatic configuration phase that uses a word embedding model trained on the Google News data-set and the entity-relationship details about the database.

1.2 Organization

In section 2, we describe the architecture of Surukam-NLI which includes a discussion on the parsing phase which identifies the entities and constraints, the schema generation phase which generates a dictionary using word-embeddings that is subsequently used for generating SQL and the mapping phase that maps entities and constraints to SQL clauses and columns. Section 3 contains a description of the AdventureWorks [3] database and an evaluation of our system's performance across multiple domains of the database. Section 4 contains some observations about our system's limits and compares our approach with the other approaches of building NLIs to databases.

2 Surukam-NLI Architecture

In this section we describe the pipeline of steps involved in translating the natural language query into a SQL query. This has also been outlined in Figure 1. Firstly, an intermediate dictionary which has tokens classified into entities and attributes using the parse tree output and named entities, is generated. Secondly, the intermediate output is mapped to the database schema using a combination of word-embeddings and database look-ups to generate the various components of a complete SQL query such as SELECT, FROM, WHERE

[1]They are function words used to ask a question, such as what, when, where, who, whom, why, and how.

[2]This system was developed at Surukam Analytics.

[3]https://technet.microsoft.com/en-us/library/ms124825(v=sql.100).aspx

and JOIN. We would be using the following query as the running example throughout this section:

total sales by year in Southeast Asia after 2001

2.1 Terminology

Let us formalize some terminology first - *entity* refers to the "target"of the user query, and hence the target of the SQL query. These typically map to SELECT statements in the final query, but may also contain other information — the "total" in the running example gives clues about aggregation and grouping. Information along with the entities usually tends to be about ordering, limits and aggregate functions viz. SUM, AVG and math operations viz., MAX, MIN. Any such supplementary information is considered a *modifier*.

The *constraints* refer to other conditions specified in the user query. Most constraints map to WHERE, GROUP and ORDER clauses. They typically correspond to NN phrases combined with PP phrases and CD values. Within a constraint, the *terms* come from noun phrases and the *condition* from prepositional phrases. Each entry in *terms* can be mapped to either a database column, relation, function or a value in a column, whereas the *condition* corresponds to either an operator or a clause.

2.2 Entity & Constraint Recognizer

In the first step of the pipeline, the natural language query is transformed into a dictionary of entities and constraints based on some rules applied on the parse tree output and the named entities.

2.2.1 Named Entity Recognition (NER)

The natural language query is run through the named entity recognizer (Finkel et al., 2005) which classifies the named entities in text into predefined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values and percentages. This information is used to deduce the constraint attribute from the NER tags of the constraint values. For e.g., in the syntax tree shown in figure 2, *Southeast Asia* is classified as location and *2001* is classified as date which would help in adding country/territory/region and year/date as the constraint attributes.

2.2.2 Syntactic Parse Tree

Parse trees are syntactic representations of a sentence that convey the part of speech (POS) for each word of a sentence and also denote patterns of useful relationships such as subject-verb-object. POS tags are assigned to a single word according to its role in the sentence. Traditional grammar classifies words based on eight parts of speech: the verb (VB), the noun (NN), the pronoun (PR+DT), the adjective (JJ), the adverb (RB), the preposition (IN), the conjunction (CC), and the interjection (UH). The tag set is based on the Penn Treebank Tagging Guidelines (Santorini, 1990). Figure 2 shows the syntax tree representation of the running example query. Next, we explain how the patterns of relationships are exploited to classify a token into entities and constraints.

2.2.3 Generate dictionary of entities and constraints

These are some rules we have used in extracting entities and constraints from the syntax tree.

1. In the syntactic parse tree, the prepositional phrase (PP) which comes under a prepositional phrase node is classified as a constraint, while nouns (NN*) are classified as attribute, cardinal numbers (CD) as value, and adjectives (JJ) as conditions on attribute.

2. The first noun phrase (NP), which is not a subtree of a prepositional phrase (PP) is classified as a candidate for an entity chunk. All the nouns under this noun phrase (NP) belong to entity list, with adjectives (JJ) as modifier and cardinal numbers(CD) as value. All the tokens with the POS tag as CD (cardinal numbers) but have a word representation are converted into numerical form, to generate valid SQL queries.

3. There are queries which do not have an entity-item explicitly mentioned. In such cases, the domain to which the query belongs is considered as an entity. The contextual or semantic relation between query and domain is inferred with the help of attribute words of all the constraints. For inferring the domain name from the set of tokens identified as attributes in the query, WordNet (Miller et al., 1990) is used. WordNet is a large lexical database for English, that collects a network of meaningfully related words into a *SynSet*. For example, in the query — *who has the highest salary in last twenty years*, the wh-word who is resolved into the domain

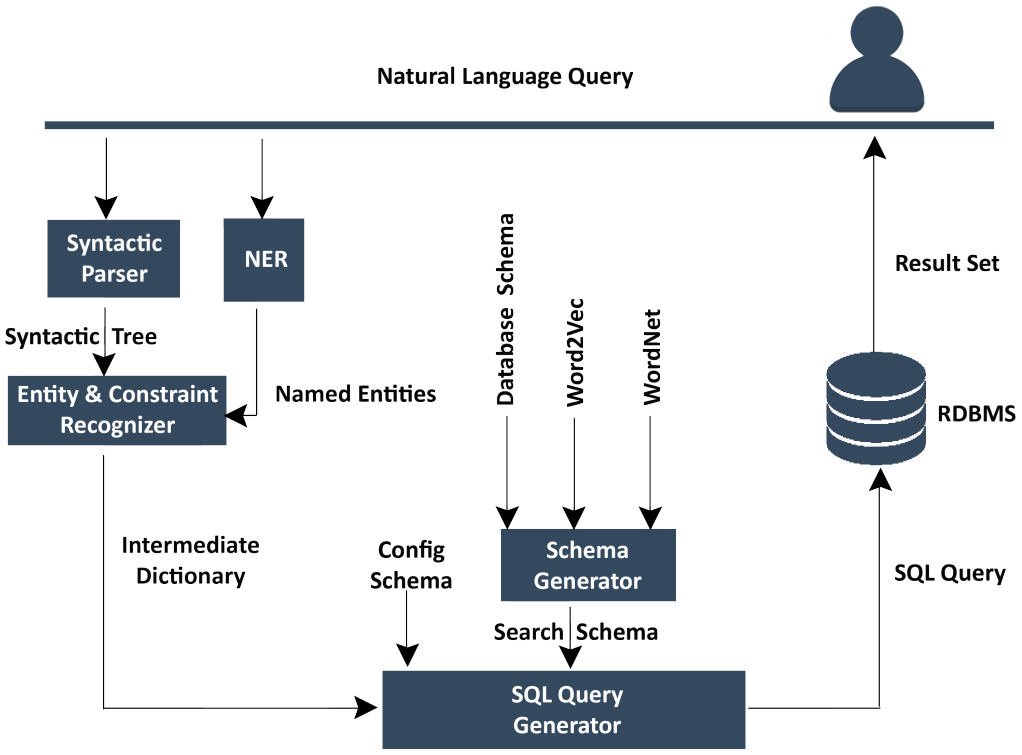

Figure 1: Surukam-NLI System Architecture

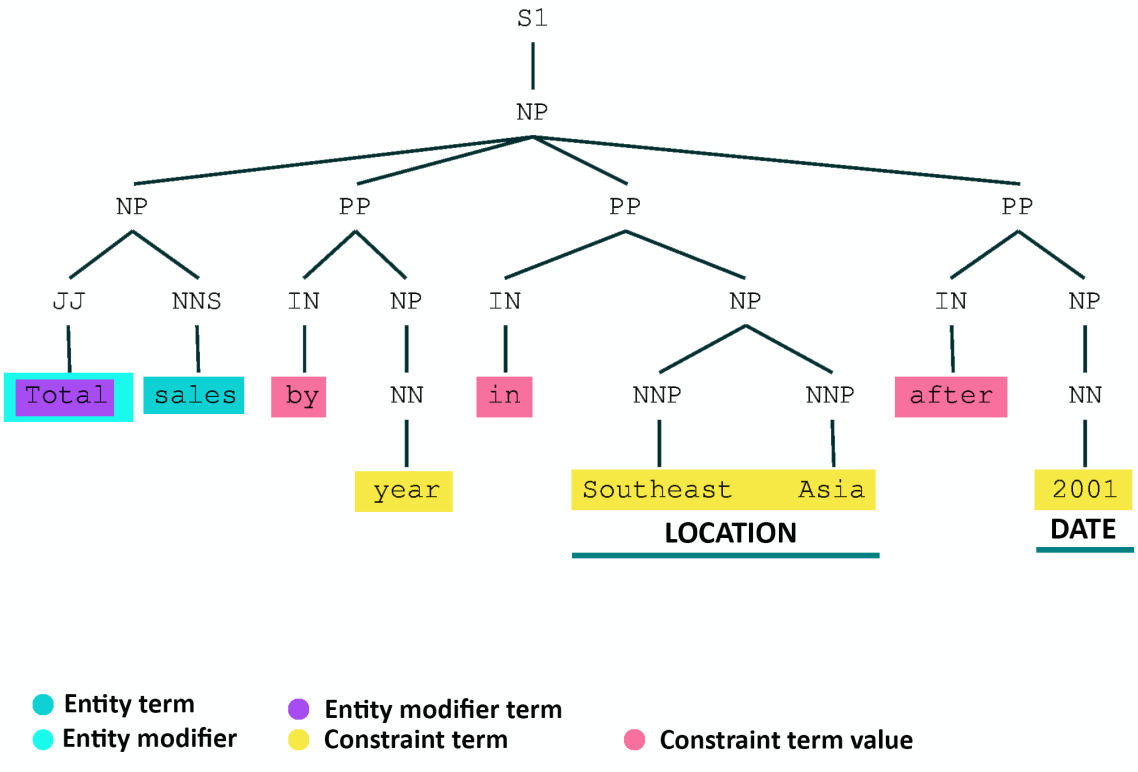

Figure 2: Syntax Tree and NER Output

employee based on the constraint term *salary* which is semantically related to the domain name *employee*, and the token *employee* is added as an entity for this query.

In our running example query — *total sales by year in Southeast Asia after 2001*, the tokens - *total* and *sales* would belong in entities with *sales* being the entity term and *total* being the entity modifier. In the same query, the tokens such as *after*, *2001*, *Southeast*, *Asia*, *by* and *year* would be classified as constraint phrases. The constraints would be further classified as: *after* into condition and *2001* as its corresponding value along with the implicit constraint term *year*, because of the NER tag. Similarly, *Southeast Asia* is classified into a constraint term which would be resolved into attribute name and value respectively, in Section 2.3.2.

2.3 SQL Query Generator

This section describes the process of mapping the output of the parsing stage to the correct columns, relations and clauses part of the final generated SQL statement. The basic structure of the output from the parse tree module is shown in Figure 3. The generation of the SQL query from the output of the syntactic parsing stage broadly involves two distinct parts - Schema Generation and Mapping.

```
{
  "entities":[
          {
        "terms"    :"sales",
        "modifiers":{
                  "terms":"Total",
                  "value":null
                }
          }
        ],
  "constraints":[
          {
            "terms":["year"],
            "condition":"by"
          }
          {
            "terms"    :["Southeast Asia"],
            "condition":"in"
          },
          {
            "terms"    :["2001"],
            "condition":"after"
          }
        ]
}
```

Figure 3: Intermediate representation of the parse tree output which has classified tokens into entities and constraints

2.3.1 Schema Generation

The goal of this module is to generate a search schema containing an expanded set of candidate database column and relation tokens. The expanded set of tokens is generated by making use of stemming, word embeddings and lexical ontologies to determine syntactically as well as semantically similar tokens.

Word stemming is an important feature present in modern day indexing and search systems. The main idea is to improve recall by reducing the words to their word roots, at both index time and query time. Recall is increased without compromising on the precision of the documents fetched, since the word roots typically represent similar concepts as the original word. Stemming is usually done by removing any attached suffixes and prefixes (affixes) from index terms before the actual assignment of the term to the index. We make use of the Snowball stemming algorithm in our implementation (Porter, 2001), which is an improved version of the Porter stemmer (Porter, 1980). Although, Lancaster stemmer (Paice, 1990) is marginally faster than Snowball, it has significantly lower precision.

Word embeddings are dense, distributed vector representations of words which try to capture semantic information about the word. Distributed representations of words in a vector space help learning algorithms to achieve better performance in natural language processing tasks by grouping similar words, and by solving the sparsity problem present in n-gram based models. Word representations computed using neural networks are especially interesting because the learned vectors explicitly encode many linguistic regularities and patterns. Semantically similar words can be found by determining cosine similarities between the word vectors. We use the skip-gram word2vec model (Mikolov et al., 2013a; Mikolov et al., 2013c) for training of word representations.

Pre-trained embeddings released as part of (Mikolov et al., 2013b) have been used. The word embeddings were trained on a Google News corpus consisting of 100 billion tokens with a vocabulary of 3 million unique words.

A lexical database is a source of lexical information that can be used effectively by modern computing methods. A lexical database often stores information about a large variety of semantic relations, such as synonymy, antonymy,

hyponymy and entailment. These semantic relations can be made use of to generate additional tokens for database column and relation tokens. The WordNet lexical database is used in our paper to determine such tokens.

The set of additional tokens is generated by making use of the above - Snowball stemming, word2vec similarity, and WordNet synsets. Some preprocessing of the original database tokens based on simple pattern matching rules commonly relevant to database column and relation names - pascal cased tokens (eg: SalespersonName), camel cased tokens (eg: territoryID) and punctuation (eg: salesperson_id) - is also performed to obtain better tokens. Finally, the generated tokens are stored in a reverse index to facilitate easy search and retrieval, and this reverse index is called the search schema.

In addition to this, a config schema is also created which contains mappings between terms and certain SQL operators, functions and clauses. (eg: greater: $>$, top: ORDER BY ASC). These mappings are seeded with initial values manually, and then the same approach as the one used for creating the search schema is applied. Note that SQL function names are dependent on the version and type of database used, and hence this config schema is made to be configurable by the database administrator.

2.3.2 Mapping

The aim of the mapping module is to map each of the elements in both entities and constraints to a clause and column. This also includes applying any possible operators to values and aggregate functions to database columns. The mapping module makes use of the parse tree output from the syntactic parsing stage, and the search and config schemas generated in the schema generation module.

The mapping is done by first doing a direct lookup on the search schema, and in case that fails, by determining a similarity score between the terms in the user query elements and keys of the search schema. A combination of Levenshtein distance (Levenshtein, 1966), and semantic similarity is used to compute this similarity.

Levenshtein distance is a commonly used distance metric between two strings given by counting the minimum number of operations required to transform one string into the other. It is a commonly used metric for spelling error correction. Semantic similarity is calculated from word embeddings by taking their cosine similarity. In case the word is out of vocabulary, ie there is no word embedding present for the word, simply the Levenshtein distance is used.

The mapping process is made configurable by a database administrator in case of special cases where Levenshtein distance and semantic similarities are not applicable, or in case the database administrator wishes to manually override any similarity based mapping. The manual configuration option is provided only for systems intended to run in production environments, and no manual configuration has been done to generate the results in this paper.

There are also cases where an explicit column term is not specified and simply a value is given in the query. The running example query *total sales by year in Southeast Asia after 2001*, does not specify year specifically, it simply mentions *after 2001*. This is known as ellipsis. To handle such cases, the mapper maintains a reverse index of all value tokens mapped to the column they belong to. Search query tokens that are not mapped to any candidate column or relation are checked against this reverse index to see if they can be mapped to a column.

In addition to tokens that map to values, certain tokens may also correspond to aggregate functions. This search is performed on the previously generated config schema with common terms for such functions being generated automatically. The config schema is also manually configurable and allows adding of custom terms for aggregate functions.

Taking the example of the intermediate representation in Figure 3, the constraint with the term *Southeast Asia* gets mapped to only the *SalesTerritory* relation, since no other table contains *Southeast Asia* as either a value token or a relation or column token. For the constraints with *by year* and *after 2001*, a number of candidate columns and relations are generated, since date objects are common in a lot of relations.

Once the list of possible columns and relations has been generated, a subset of the relations is taken such that -

1. Each token has a candidate mapped column or relation that belongs to the chosen subset of relations.

2. Schema constraints imposed by foreign keys

are satisfied. This involves creating a graph of the relations in the database by making use of foreign key information.

3. The sum of the size of the subset of relations and the number of joins required between the relations multiplied by the average token dissimilarity score is minimized. The dissimilarity score is simply calculated by subtracting the similarity score from 1.

For the given intermediate representation, the *SalesTerritory* and *SalesOrderHeader* relations are chosen, with *total sales* mapped to *SUM(salesorderheader.TotalDue)*, *by year* to *GROUP BY YEAR(salesorderheader.OrderDate)*, *in Southeast Asia* to *WHERE salesterritory.name = Southeast Asia*, and *after 2001* to *WHERE YEAR(salesorderheader.OrderDate) = 2001*.

Mapping of the condition term in constraints and the modifier term in entities is again done with the use of the config schema. Generation of the config schema makes use of semantic similarity computed using word embeddings, as well as WordNet synsets, after it has been manually seeded with initial values.

3 Experiments and Evaluation

Evaluation of the performance of our system has been done on a sample online transaction processing (OLTP) database called AdventureWorksDB, which is modeled very much along the lines of an Enterprise Resource Planning (ERP) solution.

AdventureWorksDB is an open database released as part of Microsoft's SQL Server and it has also been ported to MySQL [4]. It contains data about a fictitious bicycle manufacturing company called Adventure Works Cycles. There are five different scenarios covered in this database:

1. **Sales & Marketing Scenario** - It covers the customers and sales related information of Adventure Works Cycles. Typical queries include *show me all the individual customers who purchased goods worth greater than 100 dollars in the last year*

2. **Product Scenario** - It contains tables related to the product information like the cost of production, product description and product models, that is represented in the database. A typical query would be — *Which was the most expensive product manufactured in Southeast Asia*

3. **Human Resources Scenario** - It contains employee-related information such as the salary, joining date and manager details. Typical query — *Which employee had the highest salary in 2001?*

We create a manual dataset of 100 Natural Language queries each for three domains in the AdventureWorks DB - Sales, Product and Human Resources. The different domains have been picked to evaluate if the system is domain independent enough to handle queries for different scenarios without manual configuration.

The performance of the system is evaluated by running each of the natural language queries in the dataset through the parsing and mapping system. In case the system successfully generates a query, the resulting query is executed on the database, and the results of the query are compared to the gold standard.

The evaluation metrics are computed as described in (Minock et al., 2008). The precision is defined as the percentage of successfully generated SQL queries that result in the correct output, and the recall is defined as the percentage of natural language queries that successfully generate an SQL query.

$$precision = \frac{\#\ of\ correct\ queries}{\#\ of\ sql\ queries\ generated}$$

$$recall = \frac{\#\ of\ sql\ queries\ generated}{\#\ of\ natural\ language\ queries}$$

$$f1\text{-}score = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

4 Results

Figure 4 depicts the results of the query translation by Surukam-NLI across all the three domains we have considered in this paper and we observe similar performance throughout. The relatively small variations in precision and recall can easily result from randomness due to small sample sizes.

[4] https://sourceforge.net/projects/awmysql/

Domain	Query	SQL Generated	SQL Correct
Sales	Who had the highest sales in 2003?	Yes	Yes
	Which country had the highest sales in 2000?	Yes	Yes
	What are the total sales by region in the last five years?	Yes	Yes
	Who were the top 5 salesmen of 2001 by total sales?	Yes	Yes
	List top 10 orders by item price in Southwest.	Yes	No
Human Resources	Average employee salary by year over the past 10 years?	Yes	Yes
	What is the average salary by department?	Yes	Yes
	How many employees are over the age of 30?	No	-
	How many employees does the sales department have?	Yes	Yes
	What were the average vacation hours in 2002?	Yes	Yes
Product	What were the top rated 5 products in 2001?	Yes	Yes
	How many transactions took place in 2001?	Yes	Yes
	What is the average cost of products in x category?	Yes	No
	List the number of transactions by country in last 10 years?	Yes	Yes
	Number of purchases of amount 200 in the last month	No	-

Table 1: Results on sample queries

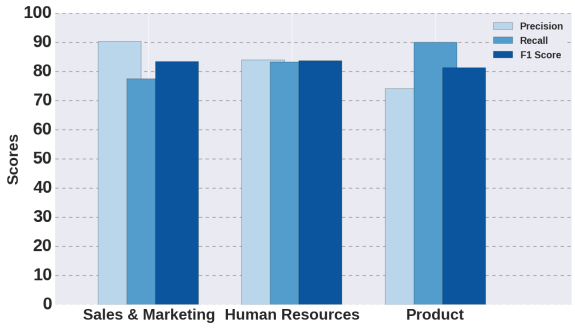

Figure 4: Histogram of Precision / Recall / F - measure for each domain.

Listed in Table 1 are a sample of the queries from each domain, whether they were successfully mapped to an SQL statement, and if the results of executing the query match the gold standard.

Delving deeper into a sample of queries for each domain, we can find some patterns in the queries that fail either in the mapping phase or in the execution phase.

1. Some complex operations require a deeper understanding of the query and the db schema. Example -
   (a) *Employees over the age of 30* - This requires the system to understand that the age of the employee can be calculated from subtracting the *BirthDate* from the current date.
2. Ambiguity in a query token leading to incorrect mapping. Example -
   (a) *What is the average cost of products of 'x' category?* - The *Products* table contains *ListPrice* and *StandardCost*, and the user intends to query on the basis of *ListPrice*, whereas *StandardCost* is the column chosen by our similarity matching algorithm
3. Ambiguity in the language of the query. Example -
   (a) *Top salesmen in 2001* - *Top* usually maps to an ORDER BY clause on a column directly mapped to the term right after top, however here in the context, *top* can refer to either amount of sales or number of sales. Without additional information, the system has no way of resolving such ambiguity.
   (b) *List top 10 orders by item price - by* in this context should map to an ORDER BY clause rather than a GROUP BY clause. This requires a deeper understanding of the query beyond the syntactic parsing.
4. Very similar database columns corresponding to query token. Example -
   (a) *How many sales have occurred in the last month?* - Here *last month* is meant

to be compared against *OrderDate*. The similarity matcher is confused by the presence of *DueDate* and *ShipDate* and is unable to resolve the ambiguity correctly.

5. Sensitivity of the parser to grammar and spelling. Example -

   (a) *How many purchases of amount 200 took place in the last month?* - The incorrect grammar of the sentence causes the dependency parser to generate an incorrect parse tree, as a result of which the mapper is unable to generate a query

6. Sensitivity of Named Entity Recognition to case information. Example -

   (a) *Sales last year in the region southeast asia* - both the dependency parser and the NER system fail to recognize *southeast asia* as a proper noun or Named Entity, and hence the term is not included in the constraints.

5 Conclusions and Future Work

In this paper, we have described a novel natural language interface to real world databases built using syntactic parse trees for query parsing and a similarity model composed of word-embeddings, WordNet and database schema rules for mapping the tokens to SQL.

By choosing a real-world database like AdventureWorks which has 67 tables spanning across 5 scenarios, our evaluation is much closer to industry requirements than a simple geological fact database like GeoBase that has 8 tables in all.

Since many NLIs like C-Phrase, Elf, and the Spanish NLI described in (Pérez, 2016) have bench-marked their performance against Atis2 (Garofolo et al., 1993), Geobase and GeoQuery250 [5] , we would like to evaluate our results against these datasets in future.

We have also shown that we were able to handle queries about three different domains without manual configuration changes, because we leveraged a very generic word embedding model that was trained on the Google News corpus, and a WordNet thesaurus to resolve the tables of a given domain. In future, we would also be enriching the auto-configuration phase by using word embedding enriched SynSets (Rothe and Schütze, 2015).

We are improving our system by adding support for complex SQL queries like nested queries and we also plan to make it a dialog system that is able to handle state and context. A typical conversation that we would like to handle in the future is :

> Q: *Who were the top 10 salesmen of 2002?*
>
> A: This query lists the top 10 salesmen with the highest sales.
>
> Q: *Sort them by their department*
>
> A: This query resolves the coreference *their* to the *top 10 salesmen* and them by their respective departments. This information would be fetched by creating an SQL JOIN operation on the *employee* table.

[5] http://www.cs.utexas.edu/users/ml/nldata/geoquery.html

References

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. *Towards a theory of natural language interfaces to databases.* Proceedings of the 8th international conference on Intelligent user interfaces. ACM, 2003.

Barbara J. Grosz, Douglas E. Appelt, Paul A. Martin, and Fernando CN Pereira. *TEAM: an experiment in the design of transportable natural-language interfaces.* Artificial Intelligence 32, no. 2 (1987): 173-243.

Beatrice Santorini *Part-of-speech tagging guidelines for the Penn Treebank Project (3rd revision).* 1990

Chris D Paice *Another stemmer.* SIGIR Forum, 24(3), 56-61, 1990.

Edgar F Codd *A relational model of data for large shared data banks.* Communications of the ACM 13.6 (1970): 377-387.

Elf *Natural-language database interfaces from elf software co.* http://www.elfsoft.com. Accessed 25 Aug 2016

E. V. Smith, K. Crockett, A. Latham, and F. Buckingham. *SEEKER: A Conversational Agent as a Natural Language Interface to a relational Database.* Proceedings of the World Congress on Engineering 2014 Vol I, WCE 2014, July 2 - 4, 2014, London, U.K.

Fei Li, and H. V. Jagadish *Constructing an interactive natural language interface for relational databases.* Proceedings of the VLDB Endowment 8.1 (2014): 73-84.

George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J. Miller. *Introduction to WordNet: An on-line lexical database.* International journal of lexicography 3, no. 4 (1990): 235-244.

Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. *Natural language interfaces to databases an introduction.* Natural language engineering 1.01 (1995): 29-81.

Jenny Rose Finkel, Trond Grenager, and Christopher Manning. *Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling.* Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005), pp. 363-370.

Joaquín Pérez *Comparative study on the customization of natural language interfaces to databases.* SpringerPlus 5.1 (2016): 1

John Garofolo et al. *ATIS2 LDC93S5.* Web Download. Philadelphia: Linguistic Data Consortium, 1993.

Madeleine Bates *Rapid porting of the parlance natural language interface.* Proceedings of the workshop on Speech and Natural Language. Association for Computational Linguistics, 1989.

Martin F. Porter *An algorithm for suffix stripping.* Program 14.3 (1980): 130-137.

Martin F. Porter *Snowball: A language for stemming algorithms.* 2001

Michael Minock *C-Phrase: A system for building robust natural language interfaces to databases.* Data & Knowledge Engineering 69, no. 3 (2010): 290-302.

Michael Minock, Peter Olofsson, and Alexander Nslund. *Towards building robust natural language interfaces to databases.* In International Conference on Application of Natural Language to Information Systems, pp. 187-198. Springer Berlin Heidelberg, 2008

Sascha Rothe, and Hinrich Schütze. *Autoextend: Extending word embeddings to embeddings for synsets and lexemes* Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing, pages 17931803, Beijing, China, July 26-31, 2015.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient estimation of word representations in vector space.* arXiv preprint arXiv:1301.3781 (2013).

Tomas Mikolov and Jeffrey Dean. *Distributed representations of words and phrases and their compositionality.* Advances in neural information processing systems (2013).

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. *Linguistic Regularities in Continuous Space Word Representations.* In Proceedings of NAACL HLT, 2013a.

Vladimir I. Levenshtein *Binary codes capable of correcting deletions, insertions and reversals.* In Soviet physics doklady, vol. 10, p. 707. 1966.

W. Woods, R. Webber Kaplan. B. *The Lunar Sciences Natural Language Information System.* Final Report, Bolt Beranek and Newman Inc., Cambridge, Massachusetts. No. 2378. BBN Report, 1972.