EFFECTIVE AND EXPLAINABLE MECHANISMS FOR NATURAL LANGUAGE INTERFACE IN DATABASES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Akifhan Karakayalı
September 2021

Effective and Explainable Mechanisms for Natural Language Interface in Databases
By Akifhan Karakayalı
September 2021

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özgür Ulusoy(Advisor)

Uğur Güdükbay

İ. Sengör Altıngövde

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

EFFECTIVE AND EXPLAINABLE MECHANISMS FOR NATURAL LANGUAGE INTERFACE IN DATABASES

Akifhan Karakayalı
M.S. in Computer Engineering
Advisor: Özgür Ulusoy
September 2021

Structured Query Language (SQL) is a commonly used tool to extract and present structured data stored in Relational Database Management Systems (RDBMSs), yet inherited complexities of SQL create barriers for naive users who are capable of expressing queries as natural language queries (NLQs). In order to tackle this barrier we propose two different solutions; a Natural Language Interface to Database (NLIDB) pipeline with an explainable AI interface and a semantic search strategy. The first solution introduces a NLIDB pipeline that uses SQL translation algorithms along with a keyword mapper to generate SQL queries for given NLQs. Proposed pipeline is presented to the user with an explainable AI interface so that the user can reason over the constructed query. We compared our approach with two state-of-art systems; NALIR+ and Pipeline+. Our approach surpass NALIR+ in imdb, scholar and yelp datasets achieving 88.9%, 100% and 60.0% translation accuracy for single table SELECT-JOIN queries and 68.6%, 87.0% and 83.6% translation accuracy for multiple table SELECT-JOIN queries, respectively. Our approach outperforms Pipeline+ in imdb and scholar datasets but Pipeline+ is slightly better in yelp dataset. The second solution proposes a semantic search approach that uses Information Retrieval based methods to retrieve related table rows for a given NLQ. The proposed approach uses the graph representation of the database where each row and value is represented with a node and edges represent the relation between them. Query and database rows are converted to vector representations using this graph representation and Graph Convolutional Networks (GCNs). A similarity calculation is performed using these vector representations and database rows are ranked according to their relevance to the query. Cosine distance metric is employed for similarity calculation. We tested our approach with college schema from Spider dataset collection and achieved a 42.8% top-5 accuracy.

Keywords: NLIDB, Query Translation, Explainable AI, Semantic Search.

ÖZET

VERİTABANLARINDA DOĞAL DİL ARAYÜZÜ İÇİN ETKİLİ VE AÇIKLANABİLİR MEKANİZMALAR

Akifhan Karakayalı
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özgür Ulusoy
Eylül 2021

Yapılandırılmış Sorgu Dili (SQL) İlişkisel Veritabanı Yönetim Sistemlerinde yapısal şekilde depolanan veriye erişmek ve görüntülemek için sıkça kullanılan bir araçtır, ancak SQL'in temelinde var olan karmaşıklıklar, sorgularını doğal dil sorgusu olarak ifade edebilen sıradan kullanıcılar için engeller oluşturmaktadır. Biz bu engelleri aşmak için iki farklı çözüm öneriyoruz; açıklanabilir yapay zeka arayüzüne sahip bir Veritabanına Doğal Dil Arayüzü (NLIDB) boru hattı ve anlamsal arama stratejisi. İlk çözüm verilen doğal dil sorgusu SQL'e çevirmek için bir anahtar kelime eşleyici ile beraber SQL çevirme algoritmaları kullanan bir NLIDB boru hattı sunmaktadır. Önerilen boru hattı kullanıcıya açıklanabilir yapay zeka arayüzü ile sunulmakta, böylece kullanıcı, oluşturulan sorguyu anlamlandırabilmektedir. Yaklaşımımız iki son teknoloji sistemle karşılaştırılmıştır; NALIR+ ve Pipeline+. Yaklaşımımız imdb, scholar ve yelp veri kümelerinde sırasıyla tek tablo içeren SELECT-JOIN sorguları için %88.9, %100 ve %60.0 ve çoklu tablo içeren SELECT-JOIN sorguları için %68.6, %87.0 and %83.6 çeviri doğruluğuna erişerek NALIR+'yı geride bırakmaktadır. Yaklaşımımız imdb ve scholar veri kümesinde Pipeline+'dan daha iyi fakat Pipeline+ yelp veri kümesinde biraz daha iyi bir performans göstermektedir. İkinci çözüm bilgi çekme tabanlı yöntemler kullarak verilen sorgu için alakalı olan veritabanı satırlarını çeken bir anlamsal arama yaklaşımı önermektedir. Önerilen yaklaşım, her bir satır ve değerin düğümlerle, bu satır ve değerler arasındaki ilişkinin de kenarlarla temsil edildiği bir çizge kullanmaktadır. Sorgu ve veritabanı satırları Çizge Evrişimsel Ağlar kullanılarak vektör temsillerine dönüştürülmektedir. Bu vektör temsilleri üzerinde bir benzerlik hesaplaması gerçekleştirilmekte ve veritabanı satırları sorgu ile alakalarına göre sıralanmaktadır. Bu benzerlik hesaplaması için kosinüs benzerliği kullanılmıştır. Yaklaşımımız Spider veri kümesinden college şeması ile test edilmiştir ve %42.8 ilk-5 doğruluğu elde edilmiştir.

Anahtar sözcükler: İlişkisel Veri Tabanlarında Doğal Dil Arayüzü, Sorgu Çevirisi, Açıklanabilir Yapay Zeka, Anlamsal Arama.

Acknowledgement

Firstly, I would like to express my sincere gratitude to my advisor Prof. Dr. Özgür Ulusoy for his endless support, guidence and help throughout my studies. Without his support I would not be able to conduct this work.

Also, I would like to thank to my jury members Prof. Dr. Uğur Güdükbay and Assoc. Prof. Dr. İ. Sengör Altıngövde for reading this thesis and providing valuable feedback.

I would also like to acknowledge TUBİTAK for supporting me financially for this thesis under grant number 118E724.

I would like to express my thanks to Dr. Arif Usta for supporting and guiding me throughout my studies. Furthermore, I would like to thank to Hasan, Mustafa, Sinan and all EA-405 office for providing joy throughout this journey.

I would like to thank to my friends Mustafa, Oğuzhan, İlayda, Alperen and Yusuf for their everlasting friendship. I also would like to thank to Aamir, Mubashira and Elmira for providing me a chance to meet new people from different cultures and get to know them.

I am forever grateful to my friends Fatih, Ömer, Enes, Haluk, Alperen, Yusuf and many others whom I cannot name here for their endless friendship and support.

I would like to thank Prof. Dr. İhsan Doğramacı, Prof. Dr. Ali Doğramacı and İhsan Doğramacı Vakfı who supported my education starting from high school. That support is the spark of everything I could achieve.

Last but not least, I would like to thank to my dearest family. I would not be successful without the support of my mom Pakize, my dad Cemalettin and my brother Sefa Karakayalı.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Relational Database Management Systems (RDBMSs) are the most popular storage infrastructures for storing structured and processed data. For extraction and presentation of this stored data Structured Query Language (SQL) is used as a common tool. However, using SQL inherently includes some complexities; such as syntax knowledge and underlying schema knowledge. These complexities create barriers for naive users who are capable of expressing the query in natural language but cannot construct the actual SQL query. To tackle this problem Natural Language Interface to Database (NLIDB) systems are proposed.

While there exist some recent works translating natural language queries to NoSQL for document based databases such as MongoDB [1], most of the existing researches focus on query translation to SQL for relational database systems. Among these researches there are pipeline based approaches [2, 3, 4, 5] and end-to-end approaches [6, 7, 8, 9, 10] using different techniques to overcome different translation challenges. Among the two, pipeline based approaches are favored over end-to-end approaches since the latter one contains numerous challenges such as inadequate performance, huge dataset need, complex deep learning models and long training time. Machine learning and deep learning systems are largely utilized in pipeline based approaches.

Deep learning systems have been implemented as a solution to various complex problems, such as image recognition [11], image detection [12], semantic role labelling (SRL) [13], and language translation [14]. Their data driven approach eases the process of tackling the complexities of the problems. As long as you gather the required data, you can build up a deep learning architecture which can separate a wolf from a dog. Or can you? Latest studies show that what and how a deep learning architecture learns is an important factor to evaluate its performance on real world situations. A neural network that is trained to differentiate dogs from wolfs might end up making predictions based on the background [15]. Their black-box nature raises the need for interpretability of its prediction.

Interpretability of the prediction can be very challenging if an architecture does not provide a rational reasoning for its prediction. Reasoning is highly significant for a human readable interpretability between the user and architecture. This reasoning must be relatable to user, connecting the prediction with contextual information and reflecting the thinking of user [16]. In other words, architecture must "explain" its prediction so that user can evaluate and decide whether it is an interpretable prediction or not.

Performing a semantic search over a database is another approach for producing a response to a given natural language query. Semantic search uses Information Retrieval (IR) based methodologies to retrieve related rows for a query. Database rows are associated with a vector representation and when a query is issued, a similarity calculation is performed between the rows and the query to score the relevance of the rows to the given query. Most related rows are presented to the user as a response.

The main contributions in this thesis can be summarized as follows:

- **A NLIDB pipeline**: We built up a complete NLIDB pipeline utilizing a keyword mapper and SQL generation algorithms to output a SQL query that can be issued on the database.
- **Explanaible AI interface**: We provided an explainable AI interface

through a web application. The application is developed using flask micro web framework and it utilizes multiple javascript libraries. The interface presents the outputs of the keyword mapper, provides explanations for those mappings, presents the graph representation of database schema and predicted join path for given query, and displays the SQL query output of NLIDB pipeline.

- **Semantic Search approach**: We proposed a semantic search approach that converts database rows and natural language query (NLQ) to vector representations by utilizing Graph Convolutional Networks (GCN), ranks database rows according to a relevance score, and retrieves the most relevant rows for the given query.

The rest of the thesis is organized as follows. Chapter 2 gives a background about the aforementioned topics. Chapter 3 presents the proposed approaches for NLIDB pipeline and explainable AI interface. Chapter 4 describes our semantic search approach. Performance of our proposed approaches is evaluated in Chapter 5. Chapter 6 concludes the thesis.

Chapter 2

Related Work

The work presented in this section extends over three different topics; natural language interface for database (NLIDB) systems, explainable AI, and semantic search on relational databases. A brief discussion of the related work performed on each of these topics is provided in each of the following subsections.

2.1 Natural Language Interface to Database (NLIDB) Systems

NLIDB systems is a long studied topic in research community. Early approaches on these systems were based on keyword search where keywords in a natural language query is identified, and the corresponding records are detected through lookups. More recent studies use parsers to identify possible keywords and hints in natural language queries to generate query sketches. Along with the advances in machine learning and deep learning, data driven translation approaches have emerged.

Yavuz et al. [17] use a process called candidate generation to create keyword mappings. First, they create all the n-grams from the question. Then, using each

n-gram they query each column of the given table to find the matches. Finally they create a mapping set for each column and n-gram match. This approach contains multiple weaknesses. First of all, it is a brute force approach and it is time consuming with long questions. The number of n-grams increases with the number of tokens in the question. Furthermore, in WikiSQL dataset every question is linked to a single table, hence there is only one table used for keyword search. However, almost all relational databases contain multiple tables and numerous columns, and querying each n-gram with all columns is not feasible.

Seq2SQL [7] uses Bi-LSTM to encode a sequence that contains columns of the related table, SQL keywords and question. It outputs three different components; aggregation classifier, select column pointer, and where clause. One of the problems with this approach is that there is only one output for the select column pointer. In most of the cases, multiple columns are required to answer a question. Furthermore, authors' work only focuses on value mapping ignoring table and attribute mapping.

SQLNet [8] defines a seq-to-set approach to eliminate reinforcement learning process of Seq2SQL. Network outputs a score for each column, and a thresholding is applied to decide which columns will be in the where clause set. The output structure of SQLNet is similar to Seq2SQL and it contains the same weaknesses. Furthermore, the works in both papers use WikiSQL dataset, therefore each question is linked to a single table. Such a setting is not a feasible solution for most of the relational databases.

Baik, Jagadish and Li [18] add a "Keyword Mapper" to the NLIDB pipeline to improve keyword mapping performance. It uses a similarity model, such as word2vec to calculate similarities between keywords and database elements, then maps them based on that similarity. It also utilizes the query logs to increase the performance. The mapper expects multiple preliminaries including parsed keywords and associated metadata with each keyword. It requires NLIDB to parse the keywords that may contain multiple words, which is one of the major challenges for keyword mapping. Therefore, the mapper cannot be plugged into NLIDB pipelines, so that does not perform detailed keyword recognition and

parsing.

SQLizer [5] uses a semantic parser to extract a query sketch from the natural language query, and iteratively repairs the sketch until a confidence threshold is met. Output of the semantic parser includes mapped hints from the natural language query words as n-grams. This mapping is not a complete mapping; database elements are not included in the initial sketch, therefore mapping includes only natural language query tokens. Mapping is updated in each iteration and completed at the end.

2.2 Explainable AI

Reasoning of decision made by deep learning models has been one of the most popular research discussions in artificial intelligence (AI) community. Due to its black-box nature, the question of what makes the model predict a particular result remained an unanswered question for a long time. To solve this remedy, many researches that use various approaches (back-propagation, perturbation, local approximation, etc.) have been carried out.

Earliest researches focus on how to fool a neural network by exploiting the activation information of network nodes. The work presented in [19] examines how a convolutional neural network extracts features from images to classify them, and adds small amount of distortions on images which causes the missclassification of image. The study reveals that although distortions are small and visually hard to distinguish, they manage to fool a neural network for classification of the image. Another study carried by Goodfellow et al. [20] reveals that adversarial examples generated using cheap algorithms are capable of fooling multiple neural network architectures. The authors state that vulnerability of neural networks to adversarial examples is a result of their linear nature, and the former explanations based on nonlinearity and overfitting are incorrect. Based on this fact, it is claimed that generating adversarial examples is a fast and simple process, and including these adversarial examples during training of neural network can

improve its performance by regularizing the network.

The work in [21] constructs an approach for describing the importance of certain tokens on output of an LSTM classifier. The authors search for phrases that are making high amount of contributions in classification of the text to its class by calculating importance scores for phrases. Using these importance scores, they extract phrase patterns which are constantly important for the LSTM architecture. With the help of there phrase patterns they try to create rule based classifiers which estimates the prediction of the LSTM classifier.

A general erasing methodology is proposed in [22] to observe the effects of erasing some pieces of representation on decision of neural network model. It is stated that, important representations can be detected through analyzing the harm caused by erasing a piece and similarly inappropriate attention can be detected through analyzing the benefits of an erasure and used as a form of error analysis. The erasure of a piece can be performed in multiple ways including input vector representation and hidden or intermediate level representation. A similar study carried by Zeiler et al. focuses on extracting strongest feature maps and pixels which plays a significant role on the classification of a given image [23]. They replace a portion of the image with a grey patch and observe the variation in prediction of model. When patch covers a significant image area, the prediction varies drastically.

LIME [15] provides explanations for single predictions to tackle the "trusting a prediction" problem. It tries to construct a locally faithful explanation for the black-box model by taking local samples close to the given instance and weighing them by the proximity to the instance. One of the most important features of the LIME is that it does not need to know the model function that is being explained. It uses the model as a black-box and only retrieves predictions for the samples it generated. Therefore the explanation can be constructed for any type of classifier. However there exists some limits to the faithfulness of the explanation. If the black-box model is highly non-linear even locally the constructed explanation would not be powerful enough.

2.3 Semantic Search on Relational Databases

There are numerous researches that focus on vector representations (embeddings) of table rows. While some of these works produce answers to natural language queries with the information gathered from a table, other works try to produce natural language expressions to table rows. Early researches represent tables with knowledge graphs or hand crafted features, whereas later works use machine learning techniques such as encoder-decoder networks to produce vector representations.

Pasupat and Liang [24] propose a logical-form driven parsing algorithm which converts a natural language query into candidate logical forms and ranks them together with the source tables knowledge graph encoding. The ranking process uses a log-linear model, and the logical form with the highest ranking is executed to retrieve the answer. A feature vector that contains the knowledge from the knowledge graph, question and logical form is used as an input to the log-linear model. This feature vector contains phrase-predicate, missing-predicate, denotation, phrase-denotation, and headword-denotation.

Zhang and Balog [25] introduce a framework that converts tables and queries into semantic vectors and computes similarities between those vectors. First, they represent the contents of the tables and queries as a set of terms. After that each term is mapped to a semantic vector representation. Finally, a semantic similarity score is computed using the vector representations. For similarity computation they propose two different strategies, namely early and late fusion, and generate four different (early, late-max, late-sum, late-avg) measures. They rank the tables according to the relevance scores between the tables and the given query.

The main problem studied in Table-to-Text [26] approach is conversion of a table row to a natural language description. The core of the approach contains an encoder-decoder framework. Encoder network tries to represent each row as a continuous vector and decoder network employs a strong copying mechanism

that retrieves rare words from table cells, columns and captions. Encoder network takes cell embeddings and attribute embeddings as input and outputs a representation for each row. Encoding is then feeded to the decoder network and a natural language description is generated. Attention mechanism is used to improve the description result.

TaPas [27] is a weakly-supervised question answering model which reasons over tables without logical form (some kind of relational algebra representation) generation. It utilizes a pre-training phase with a huge number of tables and related text segments which is a crucial step. It gets the query and flattened table as a sequence input, and outputs a subset of table cells and a possible aggregation operation which it can learn from natural language query. It implements an extended BERT architecture with additional embeddings that capture table structures.

TaBert [28] is an approach that jointly learns and represents NL text and semi-structured tabular data. It creates a vector representation for NL tokens and table columns. The most important feature of the approach is content snapshot proposal. TaBert tries to encode the most relevant subset of the table to cope with large tables with high number of rows. They also use a vertical attention mechanism to improve its performance. The authors claim that TaBert can be plugged into neural semantic parsers as a general purpose encoder to compute vector representations.

TURL [29] is a framework for learning contextualized representations on relational tables that utilizes an unsupervised pre-training and task-specific fine tuning. The authors introduce two main challenges which are relational table encoding and factual knowledge modelling. To tackle the first challenge they encode information from different table components (table caption, table headers, topic entity, table cells) into separate embeddings and then fuse them together. After fusing, they employ a structure-aware Transformer [30]. To overcome the second challenge, they learn embeddings for each entity in the pre-training phase, and model the relation between entities in the same row or column. As the last step, they propose a Masked Entity Recovery technique, which randomly masks out

some of the entities in the table and tries to recover them by utilizing information that comes from other entities.

GraPPa [31] induces a synchronous context-free grammar (SCFG) specific to mapping natural language to SQL queries. Then, using this grammar it generates synthetic natural language queries (data augmentation) and uses these queries in training. They prevent the possible overfitting caused by generated queries by including masked-language modeling loss. GraPPa utilizes a language model pre-training before applying end task training. The output of the end task is called SSP (SQL semantic prediction). The task is described as predicting whether a column appears in the SQL query and what operation is triggered on that column (SELECT, GROUP BY, WHERE, etc.).

Chapter 3

NLIDB Pipeline and Explainable AI in NLIDB Systems

An overview of the architecture that we use for translating natural language queries to SQL is provided in Figure 3.1. The figure depicts the translation pipeline along with explanation components to support decisions made. The workflow starts with an input query from the user. The query first goes through a pre-processing stage which removes special characters and punctuations such as commas, single quotes, double quotes, and tokenizes the query into words. These tokens are then converted to 300-dimensional vector representations using a pre-trained word embedding model. In our architecture, we used a pre-trained fast-text [32] model. The output of the embedding model $[X_1, X_2, ..., X_n]$ is an array of 300-dimensional vectors where n is the length of the query (the number of input tokens). Each 300-dimensional vector is a representation of each word in the original query. After that, these vector representations are fed as the input to the DBTagger model [33], which outputs corresponding schema tags for each token in the query for further translation steps. DBTagger is inspired from deep neural architectures utilized for similar sequence tagging problems in NLP field such as POS tagging and NER.

As mentioned earlier in Section 2.2, we used LIME [15] to explain output

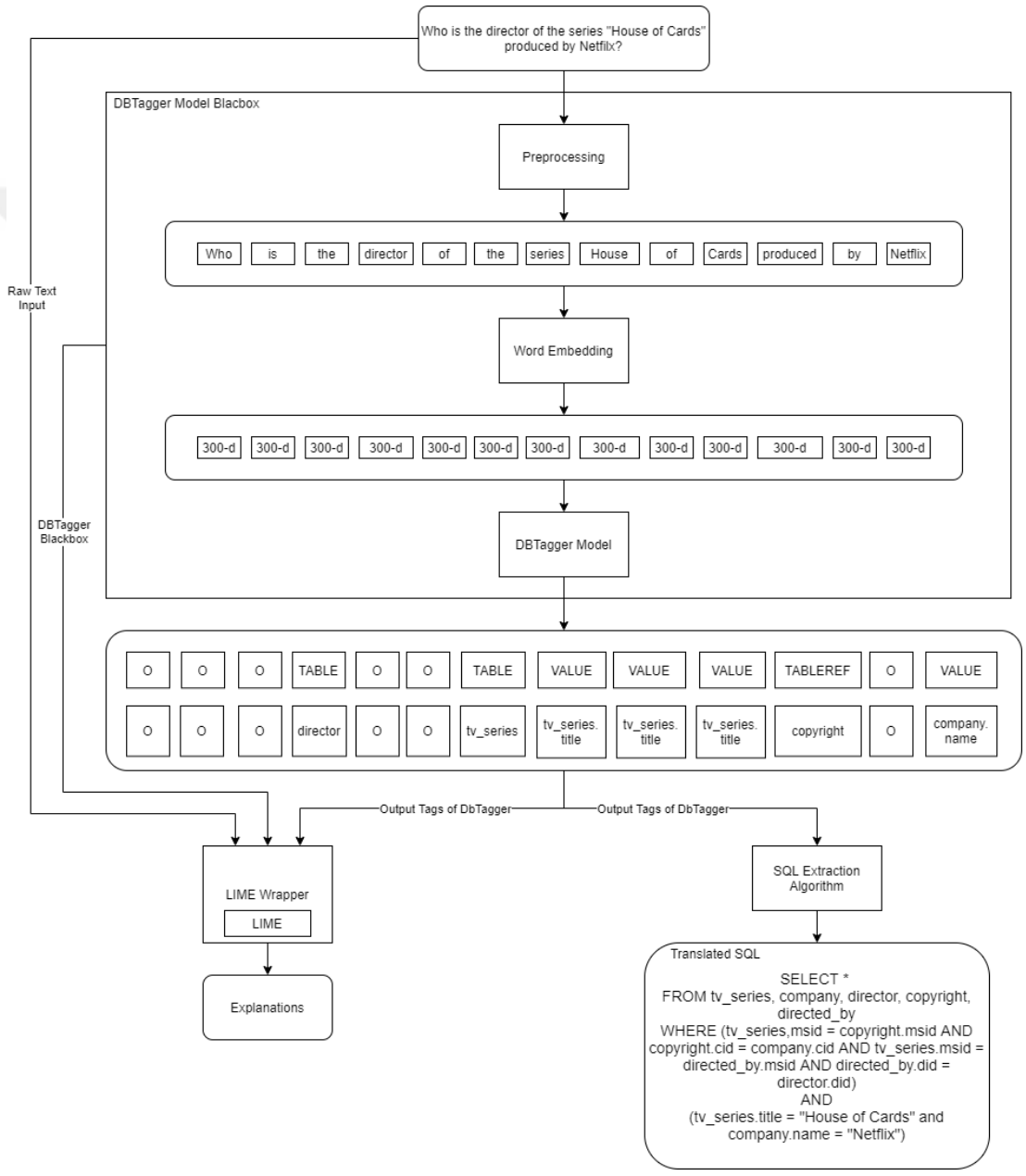


Figure 3.1: Flowchart of NLIDB Pipeline

schema tags of our DBTagger model. LIME requires a black box model which can output prediction probabilities and raw input text for explanation. To satisfy these conditions, we constructed a DBTagger Model Blackbox which takes the raw text as input and gives predictions with probabilities as output. We needed another modification for LIME to explain our tags. Original LIME assumes that given black box model classifies the whole input text and tries to explain that classification, however it is not the case in our problem since our model is a sequence tagger and predicts a class for each word. To overcome this issue, we implemented a wrapper for LIME which is explained in section 3.3. This wrapper takes DBTagger Model Blackbox, raw text input and predicted output tags to produce an explanation using LIME.

In the following subsections, we first explain the logic behind deep sequence architectures for sequential input data and state our modifications on those state-of-the-art models to better reflect keyword mapping problem for NLIDB systems. We then provide our algorithms to generate SQL translation given the schema tags output by the DBTagger model. For schema outputs, we implemented a LIME wrapper to explain the reasoning behind the decisions made by DBTagger, which is given in the last subsection.

3.1 Mapping Scheme

3.1.1 POS Tags

In our problem formulation every natural language query token (words) associates three different tags; part-of-speech (POS) tag, type tag and schema tag. To optain the POS tags of our natural language queries we used toolkit of Stanford Natural Language Processing Group named Stanford CoreNLP [34]. Although POS tags of all words are not correct, the output of the tagger is sufficient for our purposes.

NLQ	Type Tag	Schema Tag
Who	O	O
is	O	O
the	O	O
director	TABLE	director
of	O	O
the	O	O
series	TABLE	tv_series
House	VALUE	tv_series.title
of	VALUE	tv_series.title
Cards	VALUE	tv_series.title
produced	TABLEREF	copyright
by	O	O
Netflix	VALUE	company.name

Table 3.1: Predictions of DBTagger architecture for the given query

3.1.2 Type Tags

Every natural language query contains keywords which can be mapped to database entities. We separated this keyword mapping to two levels; type tagging and schema tagging. Type tags represent the type of the mapped entity in the SQL query. In total we have seven different type tags:

- **TABLE**: Natural language queries contain some nouns which are direct references to the tables in the schema, and we tag such nouns with TABLE tag. In the example at Table 3.1 nouns 'director' and 'series' are tagged with TABLE tag since they refer to the tables 'director' and 'tv_series' respectively in the SQL query.

- **TABLEREF**: Although the primary source for table references are nouns, some verbs contain references to the tables most of which are transition tables. TABLEREF tag is used to identify such verbs. In the example the verb 'produced' is tagged with this tag since it is a verb and refers to the table 'copyright' in the SQL query.

- **ATTR**: In SQL queries attributes are mostly used in SELECT, WHERE and GROUP BY clauses. Natural language queries may contain nouns that can be mapped to those attributes. We use ATTR tag for tagging such nouns in the natural language queries.

- **ATTRREF**: Like TABLEREF tag, ATTRREF tag is used to tag the verbs in the natural language query that can be mapped to the attributes in the SQL query.

- **VALUE**: In natural language queries there numerous n-grams used as filter criteria in the WHERE clause of SQL. Mapping of those n-grams is important for generation of SQL statement. We used VALUE tag for tagging those n-grams. In our design we assign a tag for each word, therefore we tagged each word of such n-grams with VALUE tag.

- **COND**: This tag is used to distinguish the keywords which contain clues about the logical operators ('<', '>', '<>', '=' etc.) used in the WHERE clause of SQL query.

- **O (OTHER)**: We tagged the remaining tokens in query with 'O' tag.

3.1.3 Schema Tag

Schema tags of keywords represent the database entity that keyword is referring; name of a table, or attribute. Tagging a keyword with a type tag is important yet incomplete. To find the exact entity keyword refers to, we defined a second level tagging where the output is the name of the tables or attributes. Originally two level tagging is not necessary for tagging tables and conditions since it is possible to create a uniqueness on single level. However, it is not possible to create the same uniqueness for values and attribute entities.

3.2 Translation Algorithms

ExtractGraph (D)
Inputs : Relational database D
Outputs: Graph model of database *dbGraph*
$dbGraph \leftarrow Graph()$;
foreach *table* $t_i \in D$ **do**
  $dbGraph.addNode(t_i)$
  foreach *column* $c_i \in t_i$ **do**
    if $s_i \notin dbGraph$ **then**
      $dbGraph.addNode(c_i)$
    $dbGraph.addEdge(t_i, c_i)$
  end
end
return *dbGraph*;

Algorithm 1: Conversion of database schema to graph

SQL extraction uses simple yet effective algorithms to construct a translation query given a set of schema tags output by DBTagger. We use the tag outputs and the input query in the algorithms. There are three different algorithms responsible for SQL query translations, which are Schema Graph Extraction, *Join* Path Generation, *Where* Clause Completion.

```
ExtractJoinRelation (G, T)
    Input  : Database Graph G and list of tables T
    Output: Graph paths that contains SQL join information
    joinPath ← ∅;
    candidate ← null;
    foreach table t_i ∈ T do
        foreach table t_j ∈ T do
            if t_i ≠ t_j then
                paths ← findShortestPaths(G, t_i, t_j)
                foreach path p ∈ paths do
                    if T ⊆ p then
                        returnPaths.append(p);
                        return returnPaths;
                    else
                        missingTables ← T \ p;
                        if candidate == null then
                            candidate ← (p, missingTables);
                        else
                            if length(missingTables) < length(candidate_1)
                             then
                                candidate ← (p, missingTables);
                end
        end
    end
    returnPaths.append(candidate_0);
    foreach table t ∈ candidate_1 do
        paths ← findShortestPaths(G, candidate_00, t);
        foreach path ∈ paths do
            listReduced ← False;
            foreach table t_2 ∈ candidate_1 do
                if t_2 ≠ t ∧ t_2 ∈ path then
                    candidate_1.remove(t_2);
                    listReduced ← True;
            end
            if listReduced then
                returnPaths.append(path);
        end
    end
    return returnPaths;
```

Algorithm 2: Extraction of SQL JOIN relations

For graph extraction we preferred a simple representation where tables and columns are considered as nodes of the graph, and edges represent the relation between them. If a column exists in a table, there must be an edge between that column node and the table node in the graph. Furthermore, we assumed that all foreign keys have the same name as their primary key, and in the graph there is only one node representing them. This simplification does not create any problem for database schemas that do not have self references or multiple foreign keys referencing the same primary key. Algorithm 1 shows how to extract a simple database graph from a schema assuming that foreign keys and primary keys have the same name. This assumption can be ignored by adding edges between foreign keys and primary keys if the given database schema contains tables with multiple foreign keys referencing the same primary key or self references.

Join path construction uses shortest path algorithm to create join relations between the required tables. After retrieving the output of DBTagger, a set of tables T is created containing all the related tables for the given query. Related tables are detected through examining the output of the DBTagger. Table entities, and tables of attribute and value entities which are detected by DBTagger are retrieved and put into set T. After that T is given as input to Algorithm 2 with database schema graph, and shortest paths containing the related tables are calculated. The output of Algorithm 2 contains a list of shortest paths which is used to construct join relations. Figure 3.2 shows the constructed join path using output of Algorithm 2 for the example query in Figure 3.1.

The next step is to construct WHERE conditions of SQL. The process starts with gathering of mappings from DBTagger. A two dimensional array M is created where M_{1i} contains i^{th} query token, M_{2i} contains type tag of i^{th} token, and M_{3i} contains the db tag of i^{th} token. After that, a simple pre-processing is done on the array M where consecutive tokens that have the same mapping information are merged together. For WHERE conditions the important mappings are $VALUE$ mappings. For each $VALUE$ mapping a tuple containing the query token and respective table column is created and added to where conditions of SQL. Algorithm 3 shows how WHERE conditions are extracted from the given list M.

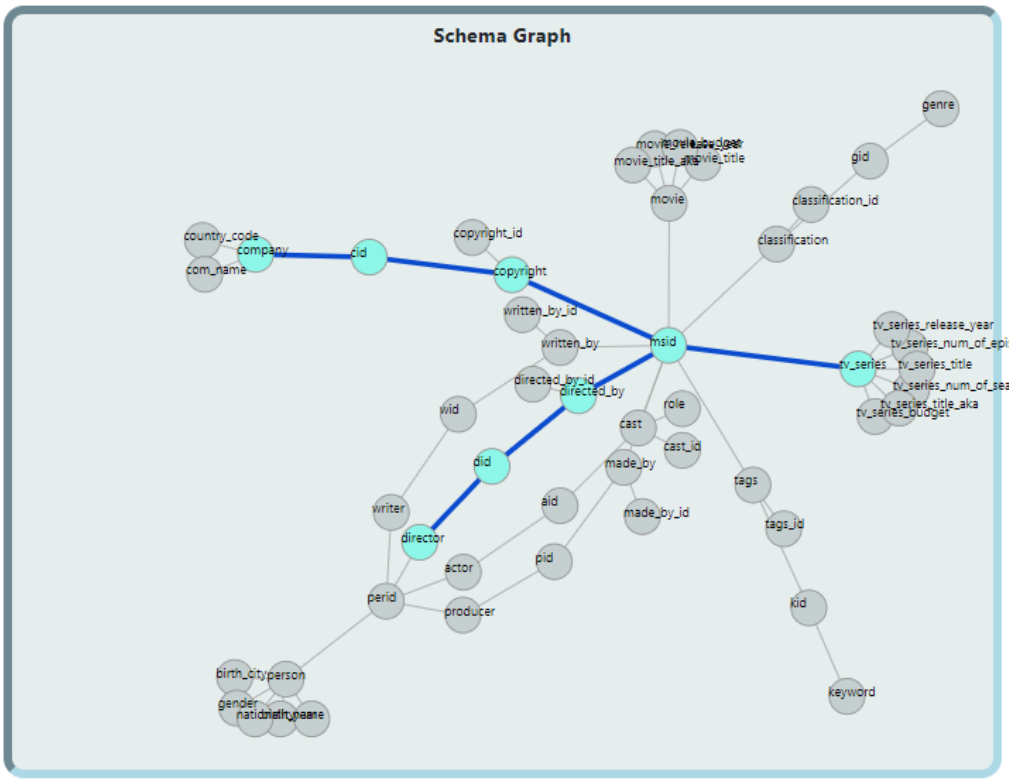


Figure 3.2: Predicted join path for given query

ExtractWhereConditions (M)

Input : Two dimensional list M which contains natural language query and keyword mapping information

Output: SQL WHERE conditions *whereConditions*

$whereConditions \leftarrow \emptyset$;

$M \leftarrow mergeConsecutiveMappings(M)$; `// Merges multi-word entities to a single mapping e.g: Brad Pitt`

foreach *token* $k_{1i}, k_{2i}, k_{3i} \in M_1, M_2, M_3$ **do**

  if k_{2i} *is VALUE* **then**

    $whereConditions.add(k_{1i}, k_{3i})$; `//` k_{1i} `and` k_{3i} `contains the keyword and column information of that keyword respectively`

end

return *whereConditions*;

Algorithm 3: Extraction of SQL WHERE Conditions

3.3 Tag Explanation with LIME

As explained in Section 2.2, LIME is a tool that explains the output of a black-box classifier algorithm locally. The tool contains text specific methods and approaches for text classification. We used these methods to produce explanations for predicted tags of our query tokens.

Normally LIME produces explanations for models that classify the whole text to one class, whereas in our case there is a classification for every token in a sentence. Therefore, we had to make some modifications and add wrappers to LIME for producing explanations for each token. The first modification was adding an output mask to the DBTagger Model that masks all outputs except the output for the token that is being explained by LIME. This mask transforms the output of the DBTagger model so that LIME can evaluate the output and produce an explanation.

The second modification was adding a wrapper to LIME that contains raw text input, DBTagger Model Blackbox, predicted tag outputs of DBTagger Model and output mask. The main purpose of this wrapper is to coordinate the communication between LIME and output mask. With the help of output mask it becomes possible for LIME to produce an explanation for a specific token, however we still need to select the token that will be explained. To achive this, we used a wrapper which detects the tokens that will be explained. We preferred explaining the tokens which have a predicted tag other than "O" for time efficiency since LIME uses a lot of resources, and producing explanations is a time consuming process even for a single token. Once the token selection is done, the wrapper adjusts the output mask so that the model would give the output for the selected token, and therefore LIME can analyze the output and produce an explanation for that token. This process is repeated for every token that is selected for explanation.

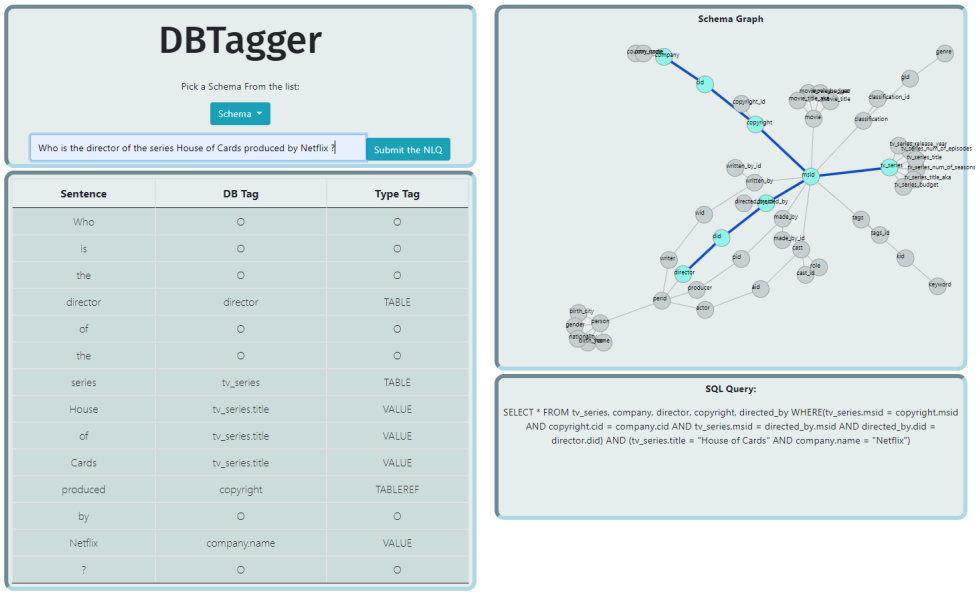


Figure 3.3: NLIDB Interface displayed with predicted tag outputs, join path and constructed query

3.4 User Interface

We constructed a simple single page web application where a user can input a natural language query to our NLIDB pipeline and retrieve the translated SQL. This web application is developed using flask micro framework and multiple javascript libraries. The interface contains four main components; natural language query input, database schema graph, keyword mapping outputs, and generated SQL of NLIDB pipeline. Figure 3.3 displays the layout of these components.

Our natural language query input component allows the user to select a database schema from a dropdown list and input a natural language query to the NLIDB pipeline. When the user submits the query it is processed in the pipeline model and generated SQL query is presented to the user along with the database schema graph and keyword mapper outputs.

Database schema graph component displays the graph representation of the selected schema. As stated in Section 3.2 database schema is converted into a

graph representation in the NLIDB pipeline. We presented this graph representation in our web application utilizing D3.js javascript library. The schema graph has a plain representation style when no query input is received from the user. When a query is processed in the pipeline and a SQL is generated, we highlight the nodes and edges of the graph that are used in the join path of generated SQL. Highlighted nodes and edges are shown in Figure 3.2 for the query shown in Figure 3.1.

Our third component contains the outputs of our keyword mapper inside a table. We present the outputs of the keyword mapper so that the user can visually see how his/her query is predicted by the keyword mapper. Furthermore, we used pop-ups to display the produced LIME explanations of each word of the query that is not tagged as 'O'. When the user clicks to a row of the table a pop-up is displayed containing the visual explanation for the word that is stored in that row. The detailed content of that visual explanation is shown in Section 5.1.3 in Figure 5.1.

The last component of our interface presents the generated SQL query for the given NLQ. This component receives the output of the NLIDB pipeline and displays it to the user as a text. Generated SQL query contains the constructed join-path and filter conditions of the WHERE clause.

Chapter 4

Semantic Search in Relational Databases

Semantic search in relational databases aims to use Information Retrieval (IR) based approaches to retrieve related rows for a given query. The main task in IR field is to rank stored documents according to their relevance to a given query so that the most relevant document is ranked highest and retrieved in first place. Semantic search uses similar techniques to rank database rows instead of documents so that the most related row to the natural language query is ranked highest. Main steps of the approach contains representation of natural language query and database rows, similarity calculation and ranking.

GetRelatedRowsForNLQ (q, k)

Inputs : Natural language query q and top k similar results for each value detected in query

Outputs: Related database rows

$dbTags \leftarrow dbTagger.predictDbTags(q)$; `// Schema tags retrieved from dbTagger architecture`

$typeTags \leftarrow dbTagger.predictTypeTags(q)$; `// Type tags retrieved from dbTagger architecture`

$V \leftarrow extractValues(q, dbTags, typeTags)$; `// extraction of values from query using predicted tags`

$T \leftarrow extractTargetTables(dbTags, typeTags)$; `// extraction of target tables from query using predicted tags`

$relatedRows \leftarrow list()$;

foreach $v \in V$ **do**

  $rows \leftarrow RetrieveRelatedRows(v, k, T)$;

  $relatedRows.append(rows)$;

end

return $relatedRows$;

Algorithm 4: Ranking the database rows related to the natural language query

The first step of semantic search is generating a vector representation of database rows and given natural language query. We used graph convolutional networks (GCN) to get vector representation of database rows. We converted the database tables into graphs where each row and each word stored in the cells of that row is represented as a graph node with an edge between them. After that we used this graph as an input to a GCN and gathered representations for each node (word node and row node) of the graph. For query representation we followed a different path to utilize the same GCN outputs. Firstly, we gathered the type tags and schema tags of the query using our DBTagger architecture. With the help of these tags, we extracted the values that are present in the query. After that for each detected value we searched the corresponding word node in the graph and took its vector representation. We considered this representation as a representation of the query. Furthermore, we used DBTagger outputs to narrow

down our search space, by defining a target table set. To decide this target table set, we extracted the attributes from the query using the predicted tags from DBTagger and inserted the table containing that attribute to the table set. The flow of value and target table set extraction is shown in Algorithm 4.

RetrieveRelatedRows (v, k, T)

Inputs : Value v detected from natural language query, k parameter, target table list T that will be used in similarity ranking

Outputs: Top k related database rows

$relatedRows \leftarrow list()$;

$R_v \leftarrow retrieveRepresentation(v)$;

$R_r \leftarrow retrieveRowRepresentations(T)$;

$sortedRows \leftarrow CalculateSimilarityAndSort(R_v, R_r)$;

foreach $i \in [0...k]$ **do**

  $relatedRows.append(sortedRows_i)$;

end

return $relatedRows$;

Algorithm 5: Retrieval of database rows that are related to the detected value

After generating the vector representations for the query and database rows, we used these representations to sort the rows according to their relevance to the query. As shown in Algorithm 5, we first retrieved the value representation as the representation for the query and row representations for the given target table set. Then, we gave these representations as input to Algorithm 6 and sorted them according to their similarity scores such that the most similar row is ranked highest. For similarity score calculation we used cosine distance.

$$cosine\text{-}similarity(A, B) = \frac{\langle A, B \rangle}{\|A\| \cdot \|B\|} \tag{4.1}$$

$$cosine\text{-}dist(A, B) = 1 - cosine\text{-}similarity(A, B) \tag{4.2}$$

CalculateSimilarityAndSort (R_v, R_r)

Inputs : Value representation R_v and database rows representation R_r

Outputs: Database rows sorted according to their similarity scores

$rowsAndSimilarityValues \leftarrow dict()$;

foreach *row* $r_i \in R_r$ **do**

$similarityValue \leftarrow calculateSimilarity(R_v, r_i)$;

$rowsAndSimilarityValues[r_i] \leftarrow similarityValue$;

end

$sortDictAccordingToValues(rowsAndSimilarityValues)$;

return $rowsAndSimilarityValues$;

Algorithm 6: Similarity score calculation between value representation and rows representations and sorting

Cosine similarity metric measures the similarity between two vectors through measuring the cosine of the angle between them. As the angle between the vectors gets smaller the similarity between them increases, therefore when cosine of the angle gets closer to 1 the vectors are considered to be more similar. Cosine similarity equation is shown in Equation 4.1. To have a convenient distance metric, where the distance between two points is less if they are closer to each other, we need to convert this similarity value to a distance value. This conversion is done by subtracting the similarity value from 1 which is shown in Equation 4.2. Cosine distance is advantageous for high dimensional representation space. When the representation space is high dimensional, small changes in each dimension adds up to a big distance in euclidean distance measure and this causes two vector representations to appear as dissimilar. Cosine distance tackles this problem and works well in high dimensional space. On the other hand, cosine distance struggles when one or both of the vector representations are filled completely with zeros (zero vector), since it cannot calculate the angle when the vector has no direction in the vector space.

Chapter 5

Experimental Results

5.1 NLIDB Pipeline and Explainable AI in NLIDB Systems

5.1.1 Dataset Analysis

In our experiments, we used imdb, yelp [5] and scholar [6] datasets which are often used to evaluate the performance of NLIDB systems. Furthermore, we used college schema from Spider [35] dataset collection for the evaluation of our semantic search models. We preferred these datasets for evaluation since training a DBTagger model requires adequate number of natural language query (NLQ) samples and these datasets have more than 100 queries which is enough for our evaluation. Table 5.1 show statistics of our datasets. Entity tables refer to the main tables such as 'person', 'author' and 'business', whereas relation tables refer to the transition tables such as 'cast', 'writes' and 'tip' that store the relation between entity tables. 'Total attributes' represents the total number of columns in dataset and 'nonPK-FK' attributes correspond to the columns that are neither primary key nor foreign key. 'Total tags' is the number of total schema tags (total tables and nonPK-FK attributes).

Properties (#)	imdb	scholar	yelp
entity tables	6	7	2
relation tables	11	5	5
total tables	17	12	7
total attributes	55	28	38
nonPK-FK attributes	14	7	16
total tags	31	19	20
queries	131	599	128
tokens in queries	1250	4483	1234

Table 5.1: Dataset Statistics

dataset	max length	min length	average length
imdb	14	4	9.07
scholar	16	2	7.53
yelp	15	6	10.00

Table 5.2: Dataset Natural Language Query Statistics

Maximum, minimum and average length of natural language queries are shown in Table 5.2. While the shortest and longest sentences are in the scholar dataset, yelp have the longest sentences on the average. Although the shortest sentence is only 2 words long, on the average the sentences contain 7-10 words which is ideal for both complex and simple queries.

5.1.2 Query Generation Results

In this section, we analyze performance of SQL translation algorithms discussed in Section 3.2. While developing the algorithms our scope was translating basic SQL queries that include simple SELECT-JOIN statements, leaving nested queries and aggregate queries out. Therefore, for the evaluation we split our metrics into two different categories; translation accuracy for simple SELECT-JOIN queries and structure correctness for the queries that contain aggregates. In both categories we used accuracy as our evaluation metric. For the simple select join queries, we accepted a query as correct if the query is totally correct. For the queries that

	Full Query Accuracy(%)			
	NALIR+	Pipeline+	Ours	
			Single Table	Multiple Table
imdb	50.0	64.8	**88.9(9)**	**68.6(86)**
scholar	40.2	76.3	**100(2)**	**87.0(69)**
yelp	52.8	**85.0**	60.0(5)	83.6(61)

Table 5.3: Query Translation Accuracies of NALIR+, Pipeline+ and Ours

	Aggregate Queries	
	Aggregate Queries with Group By	Aggregate Queries without Group By
imdb	1.00 (6)	0.88 (16)
scholar	0.89 (18)	0.90 (21)
yelp	0.64 (11)	0.80 (44)

Table 5.4: Generated Query Structure Results for Our Approach

contain aggregates, we accepted a query as correct if the FROM and WHERE clauses are constructed correctly so that only aggregate function and/or GROUP BY clause is missing.

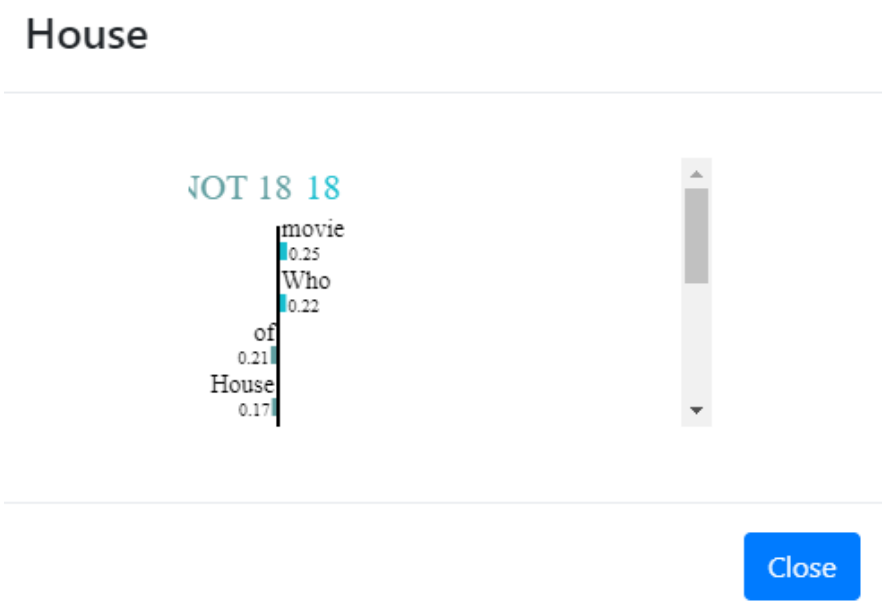
We compare our translations with reported accuracies of state-of-art systems NALIR+ and Pipeline+ [18]. Table 5.3 gives accuracy results of NALIR+, Pipeline+ and our SQL translation algorithm for simple SELECT-JOIN queries, while Table 5.4 shows the structure accuracy for queries that contain aggregates. The numbers in parentheses in Tables 5.3 and 5.4 correspond to the number of queries that fall into that category. We can see that our translation results surpass NALIR+ in all datasets and better than Pipeline+ in imdb and scholar datasets. Pipeline+ is slightly better than ours in yelp dataset.

5.1.3 Interpretability of DBTagger through LIME Explanations

Interpretability of a deep learning model is crucial for reasoning its prediction. In this section, we disscuss the interpretability of our DBTagger architecture by providing some explanation examples produced using the LIME [15] tool. For consistency we use the natural language query (NLQ) given in Figure 3.1 which is "Who is the director of the series 'House of Cards' produced by Netflix ?". Table 3.1 shows the predicted tag outputs of the DBTagger architecture for the query.

Figure 5.1 gives the explanation of the word 'House' for schema tag prediction of DBTagger. In the figure, there are two labels named 'NOT 28' and '28' which correspond to the class label of 'tv_series.title'. Below both labels, on each side there are words with a value from the NLQ. Every word and value under label '28' represent a positive contribution and every word and value under label 'NOT 28' represent a negative contribution to the classification of the word. A positive contribution means that the word strengthens the prediction probability of the explained token and a negative contribution weakens that probability. In our example, we can see that the word 'series' in the NLQ has a very high positive contribution. This means that the word 'series' strengthens the prediction probability of the word 'House' for the class 'tv_series.title'. This explanation is reasonable when we think of the fact that the query asks for a tv series titled 'House of Cards', therefore in the database we must look for a record in tv_series table where the title attribute of the record should be 'House of Cards'. Furthermore, Figure 5.1b shows the explanation of the same word if the query was asking for a movie named 'House of Cards' instead of a tv series (NLQ: "Who is the director of the movie 'House of Cards' produced by Netflix ?"). When we modify the NLQ like that, the schema tag prediction of DBTagger for the word 'House' becomes 'movie.title' (class 18), and it is observed that the highest positive contribution for this classification comes from the word 'movie'. This example shows that the classification of the word 'House' is highly dependent on neighbouring words, in other words DBTagger searches clues for the classification

(a) Explanation for word 'House' for given NLQ



(b) Explanation for word 'House' if query was asking for movie instead of series

Figure 5.1: Explanation for word 'House'

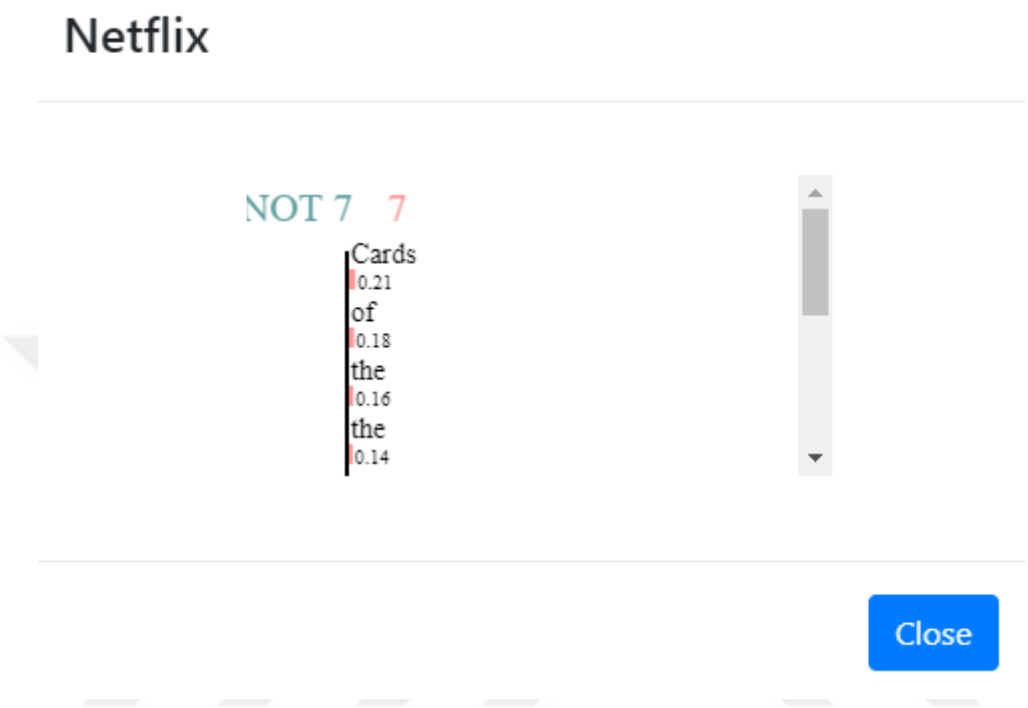


Figure 5.2: Explanation for word 'Netflix'

of the word 'House'. The explanation is reasonable and human readable since it is logical to look for a movie record titled 'House of Cards' if the query is asking for a movie instead of a tv series. Therefore, we can safely state that DBTagger learns the database schema and doesn't memorize the given dataset.

Figure 5.2 gives the explanation for the word 'Netflix' in the given NLQ. The explanation shows that the predicted tag for the word Netflix depends on the neighbouring words. The word 'Cards' have the highest positive contribution which is expected since the entity 'House of Cards' should infer to the Netflix company. The second highest contribution comes from the word 'of' which is also a part of the tv series title and is expected to have a positive contribution. This explanation of the word 'Netflix' is reasonable and can be interpreted by people, thus supports the claim that DBTagger doesn't memorize the given examples.

Figure 5.3 gives the explanation for the word 'director'. The predicted schema tag for the word is 'director' table (class 10) and unlike the other examples the

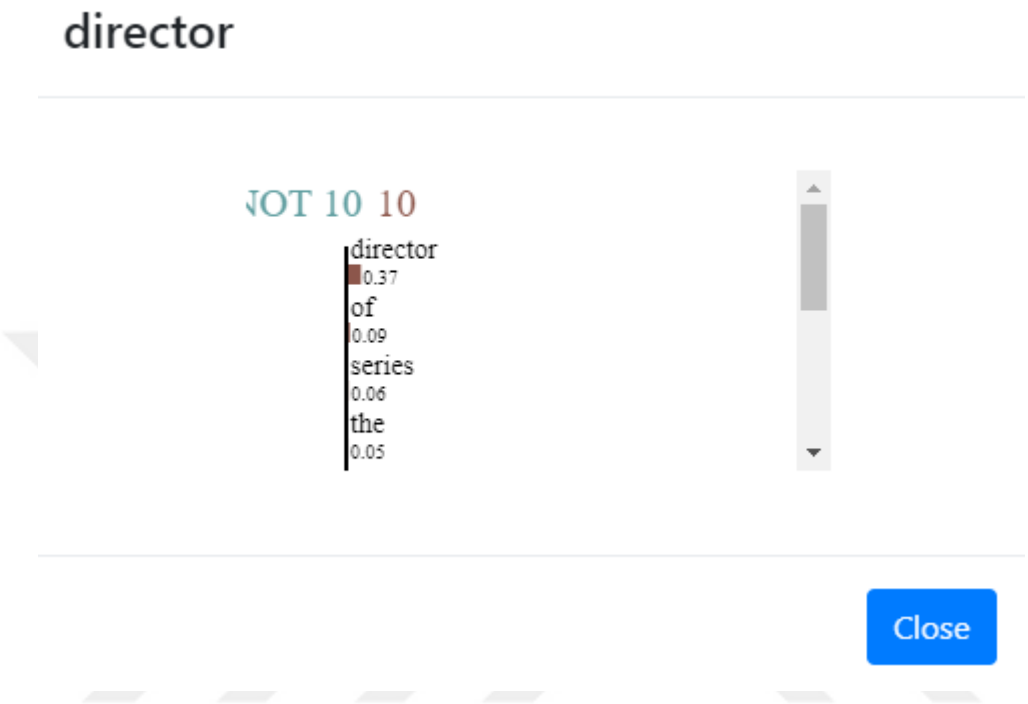


Figure 5.3: Explanation for word 'director'

highest positive contribution for this prediction comes from the word itself. Furthermore, we can observe that the other words do not really contribute that much either in positive or negative direction. In NLQs there are some words such as actor, director, movie, etc. which refer to a schema entity even it is the only word in the NLQ. The context represented by such words doesn't necessarily depend on the whole sentence, therefore their predictions should be the same most of the time. This explanation is an example of such a situation, the prediction of the word 'director' should stay the same in most of the queries.

5.2 Semantic Search

In this section we analyze the performance of our semantic search approach which is presented in Chapter 4. We performed our experiments on the college schema from the Spider [35] dataset collection, since imdb, yelp [5] and scholar [6] datasets contain over a million rows and it is not possible to process them as a graph due

dataset	# of total queries	# of evaluated queries	# of queries without representation	# of queries without result
college	170	28	12	4

Table 5.5: College Dataset Information

	Top 5 accuracy(%)	Top 5 accuracy after excluding queries without representation or result (%)
College	42.8	85.7

Table 5.6: Semantic Search Results

to technical limitations. Table 5.5 shows the statistics of our college dataset. For evaluation, we only used simple SELECT-JOIN queries since the semantic search approach aims only ranking existing database rows without further processing of them. The number of queries suitable for this evaluation is shown in '$\#ofevaluated\ queries$' column of Table 5.5. Among those queries, we could not produce a vector representation for 12 of them since we could not find exact matching nodes in database graph for the values detected in the queries. Furthermore, corresponding SQL statements of 4 queries did not return any rows as a result.

As our evaluation metric we used top-5 accuracy; we considered the result as correct if any of the top 5 rows returned from the semantic search exists in the result set returned from the actual SQL statement of the natural language query (NLQ). Table 5.6 shows the accuracy results of our proposed semantic search approach. The second column of the table presents the accuracy of our proposed method when we exclude the queries without representation or result from the evaluation set. Although excluding them from the evaluation set is not fair, the increase in accuracy shows that our approach is promising good results yet a better query representation methodology is needed. Currently, we are searching for exact matches of detected values from NLQ and average them. If we cannot find an exact match, we cannot produce a vector representation for NLQ. Therefore, to produce a vector representation for a given NLQ, the query

should contain values existing in the database schema.

Chapter 6

Conclusion

In this study we presented two different approaches that aims to produce an answer for a natural language query (NLQ) that is issued to a Relational Database Management System (RDBMS). First approach proposes a NLIDB pipeline and an explainable AI interface and the second approach utilizes the Information Retrieval based methodologies to construct a semantic search strategy.

The first work in the thesis describes a complete Natural Language Interfaces for Database (NLIDB) pipeline that utilizes a keyword mapper and SQL generation algorithms to output SQL queries. This NLIDB pipeline is supported with an explainable AI interface that presents user the outputs of the keyword mapper, provides explanations for those mappings, presents a graph representation of database schema along with the predicted join path of a given query, and displays the SQL output of NLIDB pipeline.

The NLIDB pipeline uses simple yet effective algorithms to generate SQL queries that contain simple SELECT-JOIN statements. These algorithms utilize the graph representation of database schema and output of keyword mapper for query generation. The NLIDB pipeline provides better results than the other state-of-art approaches on three datasets (imdb, scholar, yelp) that are widely used to evaluate such systems. This simple technique is also quite successful

at generating the structures of the SQL queries that require aggregate clauses. Most of the generated queries require only the aggregate clause (MIN, MAX, SUM, AVG, etc.) and/or GROUP BY clause, so that when it is issued with those additions it would yield the correct answer to the given NLQ.

The NLIDB pipeline is presented to the user with an explainable AI interface that reasons over the predicted tag outputs of keyword mapper, displays the SQL join path on the graph representation of database schema and generated SQL query. Most of the deep learning models do not provide a reasoning for their prediction and this creates a problem for interpreting the prediction of the models. We integrated LIME to our keyword mapper model so that the predictions of the model can be reasoned and interpreted by the user. Furthermore, we display the predicted join path on the graph so that the user can visually see what join path is used in generated SQL. We believe that our interface is novel as it displays multiple NLIDB pipeline related components and provides an interface for the user that does not just display the output of NLIDB pipeline but also tries to explain the resulting query with those components.

The second work in the thesis proposes a semantic search approach that uses Information Retrieval (IR) based methodologies to retrieve the most related database rows for a given query. Each row in the database is treated as a document of an IR system, and like every document and query they are associated with a vector representation. We used Graph Convolutional Networks (GCN) for generating vector representations and cosine similarity metric to score the relevance of database rows to the given query. We evaluated our approach with the college schema from the Spider dataset collection and observed that we struggle with query representation since we search for exact matching values from the query in graph nodes, and with a better query representation methodology this approach may promise good results.

As a future work, we are planning to focus on query vector representation so that we can have a vector representation for queries that do not contain exactly matching values. Furthermore, we aim to expand the scope of our NLIDB pipeline so that it can generate aggregate queries as well.

Bibliography

[1] S. Blank, F. Wilhelm, H.-P. Zorn, and A. Rettinger, "Querying nosql with deep learning to answer natural language questions," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 9416–9421, Jul. 2019.

[2] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger, "Soda: Generating sql for business users," *Proc. VLDB Endow.*, vol. 5, p. 932–943, June 2012.

[3] F. Li and H. V. Jagadish, "Constructing an interactive natural language interface for relational databases," *Proc. VLDB Endow.*, vol. 8, p. 73–84, Sept. 2014.

[4] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan, "Athena: An ontology-driven system for natural language querying over relational data stores," *Proc. VLDB Endow.*, vol. 9, p. 1209–1220, Aug. 2016.

[5] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: Query synthesis from natural language," *Proc. ACM Program. Lang.*, vol. 1, pp. 63:1–63:26, Oct. 2017.

[6] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, "Learning a neural semantic parser from user feedback," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Vancouver, Canada), pp. 963–973, Association for Computational Linguistics, July 2017.

[7] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *arXiv preprint arXiv:1709.00103*, 2017.

[8] X. Xu, C. Liu, and D. Song, "Sqlnet: Generating structured queries from natural language without reinforcement learning," *arXiv preprint arXiv:1711.04436*, 2017.

[9] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. Radev, "SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, (Brussels, Belgium), pp. 1653–1663, Association for Computational Linguistics, Oct.-Nov. 2018.

[10] N. Weir, P. Utama, A. Galakatos, A. Crotty, A. Ilkhechi, S. Ramaswamy, R. Bhushan, N. Geisler, B. Hättasch, S. Eger, U. Cetintemel, and C. Binnig, "Dbpal: A fully pluggable nl2sql training pipeline," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, (New York, NY, USA), p. 2347–2361, Association for Computing Machinery, 2020.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[12] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7263–7271, 2017.

[13] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of machine learning research*, vol. 12, no. ARTICLE, pp. 2493–2537, 2011.

[14] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick,

O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[15] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pp. 1135–1144, 2016.

[16] N. Xie, G. Ras, M. V. Gerven, and D. Doran, "Explainable deep learning: A field guide for the uninitiated," *ArXiv*, vol. abs/2004.14545, 2020.

[17] S. Yavuz, I. Gur, Y. Su, and X. Yan, "What it takes to achieve 100% condition accuracy on WikiSQL," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, (Brussels, Belgium), pp. 1702–1711, Association for Computational Linguistics, Oct.-Nov. 2018.

[18] C. Baik, H. V. Jagadish, and Y. Li, "Bridging the semantic gap with sql query logs in natural language interfaces to databases," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 374–385, April 2019.

[19] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations*, 2014.

[20] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015.

[21] W. J. Murdoch and A. Szlam, "Automatic rule extraction from long short term memory networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, OpenReview.net, 2017.

[22] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," *ArXiv*, vol. abs/1612.08220, 2016.

[23] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 818–833, Springer International Publishing, 2014.

[24] P. Pasupat and P. Liang, "Compositional semantic parsing on semi-structured tables," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, (Beijing, China), pp. 1470–1480, Association for Computational Linguistics, July 2015.

[25] S. Zhang and K. Balog, "Ad hoc table retrieval using semantic similarity," *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*, 2018.

[26] J. Bao, D. Tang, N. Duan, Z. Yan, Y. Lv, M. Zhou, and T. Zhao, "Table-to-text: Describing table region with natural language," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[27] J. Herzig, P. K. Nowak, T. Müller, F. Piccinno, and J. M. Eisenschlos, "Tapas: Weakly supervised table parsing via pre-training," *arXiv preprint arXiv:2004.02349*, 2020.

[28] P. Yin, G. Neubig, W.-t. Yih, and S. Riedel, "Tabert: Pretraining for joint understanding of textual and tabular data," *arXiv preprint arXiv:2005.08314*, 2020.

[29] X. Deng, H. Sun, A. Lees, Y. Wu, and C. Yu, "Turl: Table understanding through representation learning," 2020.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[31] T. Yu, C.-S. Wu, X. V. Lin, B. Wang, Y. Tan, X. Yang, D. Radev, R. Socher, and C. Xiong, "Grappa: Grammar-augmented pre-training for table semantic parsing," *ArXiv*, vol. abs/2009.13845, 2020.

[32] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[33] A. Usta, A. Karakayali, and O. Ulusoy, "Dbtagger: Multi-task learning for keyword mapping in nlidbs using bi-directional recurrent neural networks," *Proc. VLDB Endow.*, vol. 14, p. 813–821, Jan. 2021.

[34] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, (USA), p. 173–180, Association for Computational Linguistics, 2003.

[35] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," (Brussels, Belgium), pp. 3911–3921, Association for Computational Linguistics, Oct.-Nov. 2018.