An efficient method for Natural Language Querying on Structured Data

Hanoz Bhathena[¶], Aviral Joshi[1¶], Prateek Singh[1¶]

[¶]Machine Learning Center of Excellence, JPMorgan Chase & Co.
{hanoz.bhathena,aviral.joshi,prateek.x.singh}@jpmchase.com

Abstract

We present an efficient and reliable approach to Natural Language Querying (NLQ) on databases (DB) which is not based on text-to-SQL type semantic parsing. Our approach simplifies the NLQ on structured data problem to the following "bread and butter" NLP tasks: (a) Domain classification, for choosing which DB table to query, whether the question is out-of-scope (b) Multi-head slot/entity extraction (SE) to extract the field criteria and other attributes such as its role (filter, sort etc) from the raw text and (c) Slot value disambiguation (SVD) to resolve/normalize raw spans from SE to format suitable to query a DB. This is a general purpose, DB language agnostic approach and the output can be used to query any DB and return results to the user. Also each of these tasks is extremely well studied, mature, easier to collect data for and enables better error analysis by tracing problems to specific components when something goes wrong.

1 Introduction

With the recent revolution in information retrieval and question answering, powered by deep learning models, asking queries in a more natural question format e.g. *who scored the most points in the NBA back in 2018?* have become commonplace instead of keyword based searches. More recently models like ChatGPT [1] directly generate responses instead of just highlighting text in webpages.

A large majority of work in large scale QA systems has been on documentQA (Chen et al., 2017; Karpukhin et al., 2020), wherein both the query and the retrieval unit is of text modality. Here, both query and documents are generally embedded into a vector representation (generally in the same D-dimensional space) and fast maximum inner product search is used to retrieve the top documents. Similar approaches have been used for image search (Dubey, 2021).

However, when the information to be retrieved is in structured form i.e. table or group of tables in a database; dual embedding approaches are less common. Here, semantic parsing i.e. translating the natural language query into a formal meaning representation e.g. SQL are more common. This has inspired several text-to-SQL approaches Zhong et al. (2017); Yu et al. (2018); Finegan-Dollak et al. (2018); Iyer et al. (2017). Yu et al. (2018) introduced *Spider*, a large-scale complex and cross-domain semantic parsing and text-to-SQL dataset. Models are evaluated on (clause level) exact match between the gold SQL query and the generated one, and the execution accuracy.

However, building such models have many practical constraints, perhaps the most important one being collection of domain specific annotated data. Annotators not only need to be well versed in the query language but also have detailed knowledge of the comprehensive database schema, table structure etc. It is one thing to know a unique list of all the tables and/or columns in a database, but it is even more demanding to remember which schema or table they belong to and have to refer to this everytime to manually write an output query (even with templates provided). Additionally, if the schema/table structure changes with columns added/dropped/moved from one table, then some queries might become invalid. Furthermore, the models themselves can suffer from some of the common issues associated with auto-regressive generative models such as repetition, hallucination etc. Although these can to some extent be mitigated via decoding constraints, the process is still cumbersome and the gains in most practical use cases might not be worth it. Finally, if the DB type is changed and uses some other query language then annotating newer data might require re-training the annotation team and even existing queries would need to be translated into the second language,

[1]Equal contribution.

[1]https://openai.com/blog/chatgpt/

Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics
Volume 5: Industry Track, pages 322–331

which might be difficult for certain language pairs. An alternative end-to-end approach first introduced in Herzig et al. (2020) has also been explored in the literature where the question and flattened table are jointly embedded into a transformer model and trained using masked language modeling (MLM) to table cells. While this approach can work well on smaller tables and documents with text and embedded tables, it would be difficult to scale to large tables and have even bigger controllable generation issues than semantic parsing approaches.

In this paper, our primary contribution is proposing a simple yet powerful and configurable framework for a reliable question answering system on structured data by converting the multi-domain NLQ on DB problem to (1) domain prediction (2) multi-head slot tagging (3) raw slot value resolution or disambiguation and (4) deterministic algorithm to convert the outputs of the above three into a query for given database. For (1) and (2) we use elemental NLP models for text classification and token classification (like NER), respectively. For slot value resolution we propose a suite of methods depending on the data type of the slot extracted and intrinsic nature of the problem (lexical vs semantic similarity; contextual vs non-contextual resolutions). This approach allows us to annotate data much faster as annotators only need to know the names of the tables and unique set of columns across all tables. Furthermore, we can also individually test, improve and troubleshoot potential defects to particular components of the pipeline as required; something critical in real world production systems. We posit that our approach can be applicable to a large majority of business use cases where the natural language queries being asked do not need overly complex sub-querying and joins. Our method is capable of scaling to when the number of total *unique* columns (which is equivalent to slot types) is of the order of tens to low hundreds and same for number of tables/domains. In terms of queries we are able to theoretically support selection, filtering, sorting and aggregation and joins (in a limited capacity as described below).

2 Methodology

2.1 Problem Setting

Our goal is to retrieve data from a structured database using natural language questions. Given a user question $\boldsymbol{X}$ and a database with tables $\boldsymbol{T}$ and set of all unique columns $\boldsymbol{C}$, our framework must be capable of converting it to a database query $\boldsymbol{Q}$. There must be no dependence on the type of DB (SQL or NoSQL) and we must support select, filter, sort and aggregate clauses and do simple joins. Furthermore, our framework must be reliable enough to apply in a real world commercial setting; scalable to be trained on large quantities of annotated data; easy to gather annotated data without annotators needing to know the query language or full schema details; and engineers and scientists who build the E2E system must be able to troubleshoot issues quickly. Finally, while adding unique new tables and columns can necessitate need for re-training, editing the schema via dropping, renaming or moving tables or columns should require no re-training of the NLU model components (this can be handled in the query formulation function).

In the rest of this paper we demonstrate our approach via an example use case, which is currently deployed in production, of querying two customer transactions tables with eight unique columns. For our particular use case only filtering of data is needed and joins are not needed. However, we will also explain how the framework can be generalized and adapted to other settings.

2.2 Slot Domain Model

The first two steps of our four step framework consists of a domain classification and slot extraction model, which corresponds to (multi-class or multi-label) text classification and multi-head token classification task, respectively. Depending on the use case these can be separate models or they can be done together using multi-task learning (MTL). There is typically positive transfer between these tasks and so unless there is good reason otherwise, we propose using a model trained with multi-task learning (MTL) for these tasks.

Functionally, the domain model would be used to (a) select the appropriate table/collection relevant to the user query and (b) detect when a given query is unanswerable from the available structured data. Having this module is advisable as even in the trivial case of just one table, deciding whether the query is answerable or not is practically crucial in a real world setting where users are free to type in anything. For joining multiple tables, the simpler multi-class text classification problem would become a multi-label classification problem with two heads: one for the table name and another for the type of join (inner, outer, full, or none), with the

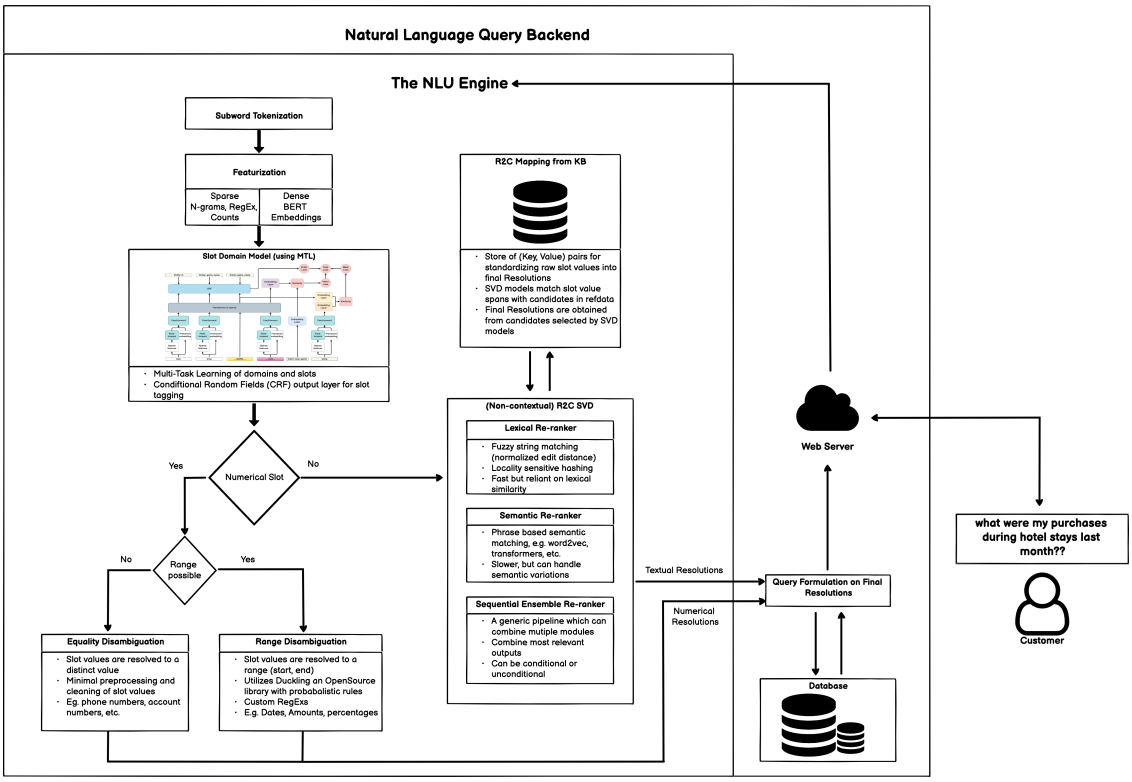

Figure 1: An overview of the proposed Natural Language Querying (NLQ) system. NLQ passes the user input to the NLU engine which is responsible for identifying the appropriate domain and slots in a user utterance, following which it disambiguates slot values using the different approaches mentioned in 2.3 and A.1. The disambiguated slot values are used by NLQ to formulate the database query and return the results to the user

join done on common column names.

The next component is multi-head slot/entity extraction which is framed as a multi head slot tagging task which must predict the *type, role and function* of a given span. The first head which we call the *type* head, predicts a label that corresponds to the column/field name. The second head predicts the *role* of that slot span e.g. *filter, sort_asc, sort_desc, aggregation, selection*. The *function* head predicts the aggregation function i.e. *count, sum, avg etc*. The slot type and aggregation function (which can be None for no aggregation) is always multi-class for a given span, but the role is multi-label as the same column can appear in multiple roles e.g. filter and sort or in select and group by.

For example, in the query "what were my purchases at Amazon in the last month"; *purchases, Amazon* and *last month* are all slots with role **filter** only. However, for "what were my *largest* purchases at Amazon in the last month", *Amazon* and *last month* are still only role of **filter**, while *purchases* has both **filter** and **sort_desc**.

2.2.1 Model Architecture

For our purposes we utilize the DIET model proposed by Bunk et al. (2020) whose implementation is available in the Rasa open source library [2]. Each head for the domain and slot prediction has a loss value associated with it. Softmax cross entropy loss is used for multi-class heads while sigmoid loss is used for multi-label heads. The total training loss is the sum of all of the individual head losses; with differentiated weighting possible if one or more heads requires it. We use balanced mini-batching to handle class imbalance and utilize data augmentation to mitigate most underrepresented classes.

In our use case we use both sparse and dense feature inputs to DIET. Sparse features include word and character n-gram counts while the dense features are sentence level ([CLS]) and token level features from a BERT (Devlin et al., 2018) pre-trained language model, with in-domain masked language modeling (MLM) pre-training. Since we use sub-word tokenization, we employ a general purpose custom token to span level label aggregation and conflict resolution also.

[2] https://github.com/RasaHQ/rasa

Furthermore, since we utilize MTL for our use case, it is possible that the slot type i.e. column and domain i.e. table might not match as our model does not have direct knowledge of the schema. One way to solve for this would be adding heuristic rules in the query formulation step to handle conflicts, say based on confidence, which is what we use. Another method is not to use a joint model, rather first predict the domain (which can map to a table) and then train separate slot models for each.

2.3 Slot Value Disambiguation

Slot values extracted from the tagging model are not fit to be used directly for querying a database. A user might enter "last week" which needs to be resolved to an actual date range. They can enter free form text with spelling mistakes or abbreviations such as "amzn" instead of "Amazon.com, Inc"; or in more complex cases use totally different values than enumerated names or codes in the DB e.g. say "coffee" whereas DB only has a category called "Beverages" or say "hotel" while the DB has value of "Hospitality". Therefore, the free form slot value from a user input must be resolved to a compatible value (generally from some knowledge base) before querying the DB. This task is commonly called named entity disambiguation (NED) or entity linking when we are in the context of Named Entity Recognition (NER). However, since in our case this is not only needed for named entities but broader types and values e.g. dates, numbers, amounts, expense categories etc we call this more generic step Slot Value Disambiguation (SVD).
Our solution for SVD utilizes the prediction of the *type* head of the upstream slot tagger to determine the type of disambiguation treatment applied to the slot value. At a high level, we categorize the type of slot values into four distinct categories.

2.3.1 Numeric slot values queried with strict equality only

Numeric slots like account numbers, phone numbers, SSNs etc. which generally have distinct values which require an exact (or partial e.g. last 4 digits of credit card) lookup into the database, but are never queried as ranges. For example, phone numbers are not generally queried as a range and hence such slot types will be used with minimal post processing for lookup against the database, such as removing dashes, commas or other non-numeric characters as needed.

2.3.2 Numeric slot values queried with equality or ranges

Other numeric values e.g. date, amounts, percentages, area etc are potentially queryable using equality (e.g. "6/1") or ranges (e.g. "6/1-9/2") and they also need to be normalized e.g. a dollar amount could be represented as "$2,000" or "2000 dollars" or "two thousand" etc. To normalize above into a standard format we utilize Duckling [3, 4]. Duckling is an open-source probabilistic parser to detect slots like dates, times, amounts and durations. It then resolves these to standard values using rule based methods. We found the entity extraction quality of Duckling quite inferior to our in-domain trained DIET model. Therefore, unlike typical usage of duckling for both slot/entity extraction and disambiguation, we use it only for disambiguation and provide the type of slot derived from the upstream tagger (DIET). To handle potentially conflicting DIET and Duckling types we supplement our SVD with a set of curated rules, see Table 4 for full details. Additionally, our solution supplements Duckling's rules by accounting for many more variations which are seen in natural language utterances, see appendix A.1 for more details.

2.3.3 Non-contextual Textual SVD

For remaining types of slot values, SVD involves mapping the raw value in the slot to one from a provided knowledge base (KB). The mapping can be non-contextual (takes only slot value span as input) or contextual (needs the entire utterance context for the mapping).
For non-contextual SVD, our approach relies on the comparing the similarity between the raw span extracted from tagger against a finite set [5] of potential resolution candidates to determine the final normalized value which is used for querying the database. We create a Resolution to Candidates (R2C) mapping from the knowledge base containing final enumerated resolutions and a corresponding list of candidates. The raw text span from the tagger is compared against the candidate list and the top-N candidates are selected which are then mapped back to the final resolution using the inverse R2C mapping. The candidates are chosen via a semi-automated approach. If available a subject matter expert can provide a seed list of candidates,

[3] https://github.com/treble-ai/pyduckling
[4] https://github.com/facebook/duckling
[5] finite but need not be static i.e. the list of final resolutions can change without necessarily needing re-training

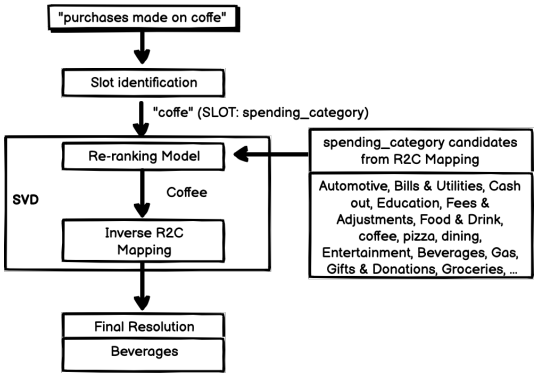

Figure 2: Overview of the candidate re-ranking and slot value disambiguation (SVD) process

however this is completely optional and might not be conducive in larger scale settings which is why we have a multi-stage automated approach to generate additional candidates as follows:

- Most frequent span values annotated to a particular resolution from training utterances are added as candidates to the R2C mapping.
- Sometimes, the same raw span might be mapped to more than one resolution. If we are in a multi-label setting, this is OK. However, for the multi-class case we tie break by choosing the resolution for which a particular span was most often assigned to by human annotators.
- We also generate synonyms for selected candidates by augmenting them with character perturbations and also derive candidate phrase synonyms by choosing top N most similar phrases from their phrase vector representations. Which phrase vector representation is a design choice; but we use Trask et al. (2015) in our experiments and its associated library [6]

We use lexical and semantic similarity between slot values and candidates from the R2C mapping. More technical details on these techniques are provided in appendix A.2.

2.3.4 Contextual Textual SVD

For certain types of slot disambiguation, the normalized value might be different according to the context e.g. "apple" in "dining table purchased at the apple store" should resolve to "Big Apple Antiques", a furniture store and not "Apple Inc". Here, the context is essential for the slot value disambiguation as only the word "apple" is not enough. Finally, contextual and non-contextual SVD methods can be ensembled using either unconditional (use all models) or conditional (only trigger other models if first one is less confident) as described in appendix A.3.

2.4 Query formulation

Given the table name and resolved slot values, types and roles; we can write a deterministic function to generate the query. The filtering clause can get slightly more complicated because filters can be connected by AND, OR conditions and can have ranges or equality conditions. We posit that AND conditions are generally be among inter-column filters and OR for intra-column filtering e.g. "how many stocks of Apple and Amazon did I purchase last week". Even though the user says "Apple *and* Amazon", the intent is to filter on both companies, the (SQL) query actually would be something like *where (company_name="Apple Inc" OR company_name=Amazon.com Inc) AND (date between <week_start> AND <week_end>)*. Additionally, notice how we separated filter conditions on equality from ranges and this (also needed for SVD) must be done by defining the type of the given column.

3 Experimental Details

3.1 Dataset

Statistics for the domain and slot/entity types on our internal dataset are provided in Table 1. We aim to support searches on two large database tables in the retail and investment space. We add a third domain *other* which serves as a background class that bypasses the remaining pipeline for unsupported queries. There are eight slot types in our dataset.

3.2 Data augmentation for Domain Slot Model

In order to mitigate high imbalance and improve robustness we used various data augmentations.

- Backtranslation: Generated semantically similar utterance reformulations by translating an utterance to another language and then translating it back to English. We then annotated these new utterances for missing slot tags using combination of exact match and certain user defined transformation functions for

[6] https://github.com/explosion/sense2vec

Domains	W/o Augmentation	Augmented
retail	9871	67548
investments	541	1807
other	3150	15763
Slots	**W/o Augmentation**	**Augmented**
acc num	313	1909
amounts	5800	56931
dates	4096	22914
merchant	2557	17147
prod. name	649	4448
prod. type	1482	7356
spend category	1463	6229
txn type	6564	35439

Table 1: Training Dataset statistics

Slots	P	R	F1
account number	86.09	100.00	92.52
amounts	98.13	99.25	98.68
dates	95.36	95.08	95.19
merchant	79.93	90.61	84.93
product name	90.29	95.64	92.89
product type	93.03	92.86	92.94
spending category	84.64	82.75	83.68
txn type	94.53	95.09	94.81
micro avg	92.53	95.17	93.83
macro avg	92.41	94.78	93.51
weighted avg	92.77	95.17	93.91

Table 2: Slot tagging results for different slot types

perturbed slot/entity spans e.g. "$50" would match against "50 dollars".

- Paraphrase generation: Same idea as back-translation, we used the Pegasus model (Zhang et al., 2019), fine-tuned for paraphrasing, [7] to generate semantically similar utterances to a source utterance. We used similar post-processing described in back-translation.
- Keyboard perturbations: Introduce character errors in words based on the proximity of character keys on the keyboard, based on a probability.
- Swap perturbations: Characters within a word swapped, based on a probability.
- Deletion perturbations: Deleting randomly chosen characters from word(s) in an utterance, based on a probability.
- Short utterances: Generate short utterances, given longer ones using keyword models, slot span only utterances and extracting slot spans with minimally required context words around them as standalone utterances.
- Math operators for amount and date ranges: Utterances with just operators along with amount and date slots e.g. <$30.

3.3 Setup

We trained our model for 400 epochs, using a balanced mini-batching strategy with batch sizes increasing linearly from 32 to 64. We used an initial learning rater of 0.001 for our optimizer, and used cross-entropy loss during training. Our domain and slot tagging DIET model had 4 transformer layers and was also trained using MLM along with domain and slot prediction. Along with this we used a dropout rate of 0.2 for the encoder and applied separate dropout to the sparse input layers but none to the attention.

3.4 Results

In this section we present the results from the different parts of our pipeline. Table 3 reports the performance results of our approach for the components of domain classification, slot value tagging and disambiguation. The domain classification model achieves an F1 score of 89.64, with the maximum confusion occuring with the "other" class. At inference this is mitigated by ignoring "other" predictions if some slot span is predicted. Table 2 shows the performance of our approach on slot tagging. Because, dates and amount type slots are subdivided into equality, from (start of range) and to (end of range), their slot tagging results are an average of the three subtypes. Overall our pipeline is able to recognize relevant slots with avg F1 score of 93.91. Finally, our SVD results can be found in Table 3, as evident from the table, our model performs the best on product type SVD (99.27 F1 score) followed by amounts and transaction type-97% accuracy and 92.58 F1 score. For dates and amounts SVD, since we do not map them to classes i.e. the ground truths are point-in-time standardized dates and amounts values, we can only calculate accuracy to measure their performance for them.

4 Conclusion

We presented a simple, yet effective and highly configurable framework for natural language querying on structured database tables which circumvents

[7] https://huggingface.co/tuner007/pegasus_paraphrase

Task	P	R	F1	Acc.
Domain class.	89.6	90.0	89.6	
Slot tagging	92.8	95.2	93.9	
SVD (txn type)	93.0	94.8	92.6	
SVD (prod. type)	98.6	100.0	99.3	
SVD (prod. name)	90.8	90.1	88.9	
SVD (spend category)	93.8	87.5	89.2	
SVD (dates)				88
SVD (amounts)				97

Table 3: Performance on domain classification, slot tagging and SVD

some of the practical constraints of generative text-to-SQL approaches. While our approach might not be all-encompassing especially w.r.t. complex sub-query generation, we empirically see that these queries are often required only in limited type of applications. Furthermore, the performance of SOTA text-to-SQL approaches today is anyway quite far away from the performance expected for commercial applications and are therefore also effectively limited to simpler queries anyway. In this setting, we posit that our approach could provide a way to quickly collect labelled data and scale to multiple domains and/or database tables while also providing much more interpretability and controllability.

Limitations

As mentioned previously, the main limitation of our approach is that, very complex joins e.g. sequences of joins of different types and joining on columns which have different names in different tables is not straightforward in our approach. One extension to possibly handle this would be using a decoder to generate the complex sequence of joins and column relations. Note, however that this does not complete revert to the constrained sequence-to-sequence decoding as in semantic parsing, as its not for the entire query but only the table joins or the *from* section of a SQL statement. The *select, where, order by and group by* can still be done via our approach and we could also continue to use MTL.

The second limitation of our approach is sub-querying capability which currently we do not have a strategy to handle queries which would require them. However, this is notoriously hard even for existing SOTA semantic parsing algorithms e.g. the current leader on the Spider dataset Graphix-T5-3B Li et al. (2023) achieves only 50 Exact Match (EM) accuracy on the extra hard Spider data subset and 61.5 on the Hard subset. Overall this model has a 75.6 EM.

Finally, the last limitation is related to comparative evaluation. We did not benchmark our method directly against SOTA semantic parsing text-to-SQL methods on open-source datasets such as Spider. This was because to do this we would have needed to re-annotate Spider or any other dataset with domain, slot extraction and resolution labels and given the size of open source datasets this was infeasible given available annotation resources. However, we can say that on private datasets and use cases, this approach was tested against some existing text-to-SQL approaches and was very competitive especially as we could collect a lot more data for these simpler tasks and also were able to train, evaluate, troubleshoot and improve different components individually.

Ethics Statement

All the work done and discussed in this paper meets and upholds the ACL Code of Ethics.

References

Tanja Bunk, Daksh Varshneya, Vladimir Vlasov, and Alan Nichol. 2020. Diet: Lightweight language understanding for dialogue systems. *arXiv preprint arXiv:2004.09936*.

Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading wikipedia to answer open-domain questions. *arXiv preprint arXiv:1704.00051*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Shiv Ram Dubey. 2021. A decade survey of content based image retrieval using deep learning. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(5):2687–2704.

Catherine Finegan-Dollak, Jonathan K Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. *arXiv preprint arXiv:1806.09029*.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*.

Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. Tapas: Weakly supervised table parsing via pre-training. *arXiv preprint arXiv:2004.02349*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. *arXiv preprint arXiv:1704.08760*.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023. Graphix-t5: Mixing pre-trained transformers with graph-aware layers for text-to-sql parsing. *arXiv preprint arXiv:2301.07507*.

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

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013b. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Andrew Trask, Phil Michalak, and John Liu. 2015. sense2vec-a fast and accurate method for word sense disambiguation in neural word embeddings. *arXiv preprint arXiv:1511.06388*.

Wenpeng Yin, Jamaal Hay, and Dan Roth. 2019. Benchmarking zero-shot text classification: Datasets, evaluation and entailment approach. *arXiv preprint arXiv:1909.00161*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. 2019. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

A Appendix

A.1 Supplemental rules for numerical range SVD using duckling

Since we use duckling just for disambiguation of already extracted slots and not how its typically used i.e. extraction and disambiguation with the entire utterance provided, we need to (1) resolve for certain potential conflicts between slot type from our trained DIET slot tagger and duckling's internal slot type, see Table 4 which contains one such example for date ranges; (2) help improve duckling's resolution performance using certain pre and post-processing rules as below:

- Merging separate slot spans split by the slot tagger. Example: "August 2nd to 10th" are tagged by the slot tagging model as [August 2nd](date_from) to [10th](date_to). In this case duckling would fail to resolve "10th" as "10th of August". Hence we merge two otherwise independent spans to [August 2nd to 10th] from which duckling is capable of resolving the range correctly.
- Use regex patterns to map diverse date formats e.g. "mm/ddyy", "mmdd/yy", "mmddyy", "mm/ddyyyy", "mmdd/yyyy", "mmddyyyy" to standardized "mm/dd/yyyy" to make SVD process output more consistent and reliable.
- Combining multiple duckling outputs during post-processing e.g. for "May 2021", duckling does not understand it is May in the year 2021 but splits "May" and "2021" and detects it as May of the current year (say 2022) and the entire year of 2021, which is incorrect. Our custom post-processing corrects the resolution.
- For dates add "st", "nd", "rd", "th" etc. as applicable e.g. "1" -> "1st" to help duckling resolution.
- Prepend extracted amount span values without $ sign with a dollar sign, Ex. "$30 to 200" or "$30-200" can only be interpreted correctly when $ is prepended to 200.

A.2 Non-contextual SVD methods

A.2.1 Lexical ranking

This ranking approach relies on the lexical similarity between the raw slot value in the tagged span

Duckling	Slot	from_date	to_date
value	date_eq	from_date	to_date
value	date_from	from_date	None
value	date_to	None	to_date
from	date_eq	from_date	None
from	date_from	from_date	None
from	date_to	from_date	None
to	date_eq	None	to_date
to	date_from	None	to_date
to	date_to	None	to_date
from+to	date_eq	from_date	to_date
from+to	date_from	from_date	to_date
from+to	date_to	from_date	to_date

Table 4: Dataset statistics pre and post augmentation

and the candidates of the slot type R2C mapping to identify the most relevant candidate. The top candidate(s) are used to lookup the inverse R2C mapping to obtain the final resolutions. For example, consider the phrase "starbks crd" in a user utterance of the form, "my purchases using my starbks crd", needs to be resolved to the name "Starbucks Card". We might not be able to come up with all possible variations, mis-spellings, abbreviations, or synonyms of *Starbucks* and so we collect these from our training data (SVD ground truth labels tagged by human annotators) to improve our R2C mapping for recall. Then we use fuzzy string matching, specifically a length normalized Levenshtein distance, but other string similarity metrics could also work.
The advantages of this Fuzzy string match approach are:

- Relatively quick to execute, especially when the candidate list is small.
- Needed when slot values and resolutions are more lexically than semantically similar e.g. people names, company names etc.

The disadvantages are as follows:

- If words have similar meaning but widely differ in characters (e.g. beverages and coffee) then simple string similarity is insufficient. While this can be somewhat mitigated by R2C augmentations from labelled data, one needs large enough dataset for this.
- Does not use context surrounding the word hence cannot be utilized for contextual methods.

A.2.2 Semantic similarity

With certain spans it might be preferable to use semantic information for disambiguation. For the coffee and beverage example above, using lexical similarity would lead to poor results if "coffee" was not a candidate in the R2C mapping for "beverage". The most intuitive method to do this is by training a phrase embedding model using classical techniques like word2vec (Mikolov et al., 2013b,a) or GloVe (Pennington et al., 2014) and calculating a phrase similarity of the raw slot value embedding against the candidates in the inverse R2C mapping. For very large number of candidates where pairwise similarity is impractical, approximate nearest neighbor (ANN) algorithms like FAISS (Johnson et al., 2019) or ScaNN (Guo et al., 2020) could be used.
Finally, we also experimented with an alternative zero-shot approach which can be used in certain cases. Based on the method put forth in (Yin et al., 2019) we formulate our task into one of textual entailment, where spans and the candidates are converted into (premise, hypothesis) pairs using a predefined template, with high entailment score signifying semantic similarity.

The advantages of using semantic similarity:

- Works even when the candidate is not lexically similar to the span value mentioned in the customer utterance.
- Can be used in unsupervised way but can also be fine-tuned to specific domain if training data is available.
- These methods can be used for contextual SVD as well, if in-domain data is available.

The disadvantages are as follows:

- Might need in domain training data especially for specialized domains where unsupervised or self-supervised learning is insufficient.
- On average are slower than a string based algorithms, although this is less of a problem in recent times due to availability of fast ANN algorithms as mentioned earlier.

A.3 Ensemble SVD Re-ranking

We can chain multiple SVD components for the same slot value resolution, choosing the best resolution using pre-defined criteria such as majority voting. The decision of whether to execute all SVD

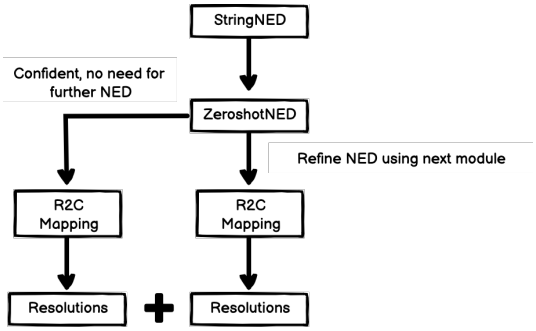

Figure 3: Ensemble SVD Re-ranking

components in the sequential chain can be based on confidence (conditional chaining) or not (unconditional chaining).
Conditional chaining works as follows and is highlighted in figure 3:

- The first SVD module returns the top candidates using a threshold.
- If the confidence score exceeds a set value and/or the first N values are within a given ambiguity threshold we directly return the SVD outputs.
- Else, we proceed to the next SVD module to help improve the final disambiguation, repeating this until the last available SVD module or until a high confidence prediction can be made.