Ruprecht-Karls-Universität Heidelberg
Institut für Informatik
Lehrstuhl für Parallele und Verteilte Systeme

Bachelorarbeit

Data Efficient Semantic Parsing for Spreadsheet Queries using Decomposition

Name:	David Schwenke
Matrikelnummer:	3395053
Betreuer:	Prof. Dr. Artur Andrzejak
Datum der Abgabe:	31.08.2022

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit ist in gleicher oder vergleichbarer Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

[illegible]

Heidelberg, 31.08.2022

Zusammenfassung

Wir schlagen eine Methode zur Implementierung einer natürlichsprachlichen Schnittstelle für Tabellenkalkulationsprogramme mit Hilfe von maschinellem Lernen vor. Zu diesem Zweck werden wir ein Proof-of-Concept-System vorstellen, das auf minimalen Daten aus vier verschiedenen Domänen trainiert wurde. Angepasst an ein System für die Roboter-Mensch-Interaktion [1], kann das System neue Abfragen, durch Zerlegung in bekannte Unterabfragen, erlernen. Wir diskutieren mehrere Herausforderungen, die sich bei der Herangehensweise an dieses Problem ergeben, sowie Lösungsansätze. Anschließend geben wir eine quantitative und qualitative Bewertung unseres Systems. Auf dieser Grundlage geben wir ein Urteil über die allgemeine Praktikabilität und Leistungsfähigkeit eines solchen Systems sowie unserer Lösungsansätze ab.

Abstract

We propose a method for implementing a natural language interface for spreadsheet programs using machine learning. To this end, we will present a proof-of-concept system trained on minimal data from four different domains. Adapted from a system for robot-human interaction [1], our system can learn new queries by decomposition into known subqueries. We discuss multiple challenges which arise in approaching this problem as well as solution approaches. Subsequently, we will give a quantitative and qualitative evaluation of our system. From this, we provide a verdict on the general practicality and performance of such a system along with our solution approaches.

Contents

1. Introduction

1.1. Motivation

Spreadsheet applications are some of the most powerful tools available in standard office suites. They handle computations and data for a myriad of problems. From private festivity planning to prototyping of complex financial products, millions use them daily. Unfortunately, their broad capabilities make it hard to equip such programs with an intuitive user interface (UI). This is made worse because spreadsheet programs are usually secondary tools used by people with little to no experience in programming.

Multiple tools to create string wrangling programs from input-output pairs have been developed [2, 3]. While these make working with spreadsheet applications easier, they cannot always recognize the desired pattern. In addition, they require the user to find sufficient unambiguous examples. Furthermore, these approaches cannot be easily modified to tailor to many applications. Thus, a more general approach to ease the complexity is needed.

A natural language interface can be more intuitive while being more general. Users could interact using their domain knowledge to provide sufficiently concrete utterances to interpret into machine-understandable code. Interpreting natural language into machine-understandable code is called "Semantic Parsing" [4].

In recent years statistical parsers have shown the best performance in generating database queries from questions [5, 6, 7]. Inspired by the success of contextualized embeddings of natural language words for downstream tasks such as translation [8, 9]. The aim is to create natural language-aligned embeddings of tables and use them for semantic parsing. The issue is to train language models to generate the desired embeddings [7] because there is a lack of high-quality data in this field.

Therefore, creating a cost-effective statistical semantic parser for tabular data requires a solution that can use the power of contextualized word embeddings without the need for copious amounts of domain-specific data.

1.2. Goals

The main goal of this work is to adapt a continually learning semantic parser for tabular data from a system for human-robot interaction [1]. We will explore the viability of this approach for tabular data in office automation.

The parser will initially be trained on a limited set of atomic operations and multiple corresponding natural language utterances. For this, we will define a pseudo-domain-specific language (DSL) containing these atomic actions. The main feature of the parser will be the ability to train new data points from user-system interaction by dividing more complex operations into pre-trained atomic examples. We will measure its performance on hand-crafted data from four different domains. We divide our data such that we can measure the performance with increasing difficulty.

1.3. Contributions

To achieve the goals set out, we will make the following contributions:

1. A functional implementation of the semantic parser outlined in Section 1.2.

2. A rule-based method for identifying objects in natural language sentences using dependency grammar and a qualitative evaluation of it.

3. A method to improve data separation of word embeddings using metric learning and a quantitative evaluation of its impact.

4. A data-efficient method for using contextualized table embeddings as part of a continually learning semantic parser.

5. A method for using the output of another pre-trained semantic parser to inform the output of our system.

6. A hand-crafted data set containing natural language utterance code pairs of three different degrees of difficulty.

7. A comprehensive quantitative and qualitative evaluation of the semantic parser and the solution approaches outlined here.

To create a functional implementation of the continual learning system as described by Karamcheti et al. [1] on spreadsheet operations, we are changing the original system in multiple ways. For one, we are modifying the entity abstraction method away from a lexical search more complex grammar-based system. Furthermore, we change the embedding of utterances away from statistical methods based on co-occurrence toward deep contextualized embeddings. We also change the way we postprocess these embeddings compared to the original system. In addition to these changes in the way we create text embeddings, we also introduce metric learning to cluster our utterance vector representations in groups corresponding to the same output program.

Moreover, we introduce a new strategy for capturing the context of the parser such that tabular data can be used. This is not part of the original system. Here we will also present a method for using a second parser to hint the correct program for the system, which is another addition to the original system.

For a more in-depth comparison of our implementation and the system proposed by Karamcheti et al., please refer to Section 4.2.

1.4. Structure

Chapter 2 provides a comprehensive description and explanation of the theoretical concepts around semantic parsing. It describes in detail all statistical tools used in the semantic parser.

In Chapter 3 related work which is either used in the making of the semantic parser or inspire aspects of its structure are described. Chapters 2 and 3 provide a context for this work in recent research as well as the topic of semantic parsing and, beyond that, natural language processing as a whole.

Chapter 4 describes how we use the tools and knowledge and build our semantic parser. It will motivate the decisions made over the course of this work and provide details on how to replicate the results. The chapter follows the structure of the semantic parser from natural language input to DSL output.

Chapter 5 provides in-depth results of different tests made on the system and its parts. First quantitative measures of the accuracy of our semantic parser are presented. Here we will measure both the performance on the pre-trained data set as well as its one-shot generalization performance from user-machine interaction in multiple configurations. The result is an ablation study measuring the performance of each component and solution approach. Our testing methodology for anyone who wants to replicate these results is explained. Consequently, we look at qualitative examples from the data set to motivate some of the shortcomings seen in the quantitative analysis.

Finally, in Chapter 6 we will give a verdict on the feasibility of our approach and provide avenues in which our system can be improved.

2. Theoretical Background

2.1. Semantic Parsing

Semantic parsing is the exact translation of natural language utterances into a machine understandable code or logic [4] called meaning representations.

The field is a mainstay of natural language processing as it has many applications, such as question answering [4, 10], automated reasoning [11] and code generation [6, 7].

2.1.1. History

One of the early examples of semantic parsing is AMR-parsing. Here a program translates natural language utterances into the Abstract Meaning Representation (AMR) [12]. AMRs aim to provide a machine-understandable representation for every sentence where semantically similar sentences are assigned to the same AMR. These representations are used for downstream tasks like text summarization [13] or text generation [14].

```
(w / want-01
  :ARG0 (b / boy)
  :ARG1 (g / go-01
          :ARG0 b))
```

Figure 2.1.: AMR example for "The boy wants to go." from Banarescu et. al. [15]

Early AMR-parsers were symbolic parsers containing a vocabulary of labeled words and rules or code templates for translation. Later, formal grammars (see Section 2.2) in conjunction with chart parsers are used [16]. The advantages of symbolic and grammar-based parsers are that they do not necessarily require labeled training data while still being computationally inexpensive. On the other hand, they require extensive linguistic knowledge. Furthermore, these systems could not create correct outputs from utterances containing out-of-vocabulary words.

Due to the increasing amount of data and computing power, statistical parsers trained on labeled data sets were conceived. These mitigate the issue of out-of-vocabulary words and do not need much linguistic knowledge to create. These statistical models grew into the state-of-the-art deep recurrent neural networks and transformers we know today. Lately, intermediary meaning representations, mostly vector-based embeddings, are used as an abstraction to enable multiple downstream tasks, including semantic parsing [8, 17, 9, 6, 7].

2.1.2. Meaning Representations

Meaning representations are the machine-understandable translations of a semantic parser. This broad definition enables a meaning representation to be a vector [9, 17], a formal system like lambda calculus [18] and AMR or code from a domain-specific language like SQL [7, 6, 5] as well as of general-purpose programming languages like Java.

Domain-Specific Languages

Domain-specific languages (DSLs) are computer programming languages geared toward use in a specific domain. They are limited in their expressive capabilities, making them easier to learn [19]. Some well-known examples are the HyperText Markup Language (HTML) for structuring webpages or the aforementioned Structured Query Language (SQL) for databases.

One can see the benefits of DSLs when looking at SQL compared to a general-purpose language like Java. The implementation of the tables in SQL are hidden. In fact, there are no complex data structures available in SQL, only tables. Furthermore, the details of core functions and algorithms are mostly fixed and cannot be changed. In contrast, general-purpose languages expose all details and leave it to the user to provide sufficiently well-posed interfaces to their system. This can lead to verbose expressions requiring extensive knowledge in programming to verify correctness. As a consequence, one can create natural language DSL utterance pairs for a supervised learning task comparatively easily.

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM orders
ORDER BY OrderDate;
```

Figure 2.2.: An example of a SQL query selecting columns from a table and sorting them. Limiting the domain to database management and lacking implementation details provide more natural queries than general-purpose programming.

One recent data set in DSL-parsing is WikiSQL [20] which contains a hand-annotated set of question SQL query pairs. Another approach in DSL-parsing is to jump labeling altogether for semi-supervised or unsupervised training like in the seminal dataset made by Papsuat et. al. [10] for semi-structured tables from Wikipedia. SQL can be used here as an intermediary meaning representation to reach the desired goal [7].

Lambda Calculus

Lambda calculus is a logical system introduced by Alonzo Church [21] to formally define functions and provide semantics on their computability. In computer science, adapted versions of this formal system are used to provide anonymous functions in programming languages. Barendregt provides the following elegant inductive definition of lambda calculus in [22].

Definition (Lambda Calculus). *Given the symbols $(,), \lambda, .$, and an countably infinite set of variables $a, b, c, ...$, we define the set of all lambda expressions Λ as follows:*

1. *If x is a variable, then $x \in \Lambda$* (2.1)

2. *If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$* (2.2)

3. *If $M, N \in \Lambda$, then $(MN) \in \Lambda$* (2.3)

Instances the form in 2.2 are called abstractions where M can be some arbitrary function made up of variables, like $x^2 + y^2$. The variables between λ and . are the input variables of the function. These variables are also called bound, while all other variables in M are called free. In the example $\lambda x.M$, where M is defined as $x^2 + y^2$, x would be bound and y free. Rules of the form defined in 2.3 are called applications where a function is applied to the result of another.

2.2. Grammar Formalisms

Grammars are a set of rules and symbols describing syntactic properties of languages in an abstract way [23]. We will look in particular at two grammar formalisms that capture syntax in a computationally tractable manner: Combinartory Categorial Grammars (CCG) and Universal Dependencies (UD). Furthermore, we will define two overarching types of grammar called dependency and constituency grammars. This section is based on the pertinent chapters 12, and 14 in the book of Jurafsky and Martin [23].

2.2.1. Constituency Grammars

Constituency grammars are conceived around the idea that groups of words can behave as single units, called constituents [23]. A noun phrase is the most trivial example for a class of constituents. Looking at "the school bus" or "they" we can see that these phrases can be used in very similar positions as seen here: "The school bus picks up children." and here: "They play football.", where the noun phrases are put in front of a verb.

We use this notion of constituency to create grammars that contain rules grouping sequences of words to a limited set of abstractions. The most widely used system for modeling constituency are called context-free grammars (CFGs) or phase-structure grammars and are formally defined as follows [24, 23]:

Definition (Context-free grammar)**.** *A context-free grammar is defined by a 4-tuple N, Σ, R, S where:*

- *N is a set of non-terminal symbols*
- *Σ is a set of terminal symbols disjoint from N*
- *R is a set of rules or productions of the form $A \to \beta$, where $A \in N$ and β is a string of symbols from $(\Sigma \cup N)^*$*
- *S a designated start symbol in N*

In the case of natural language grammars, Σ is a set of words called the lexicon, N is the set of word and phrase types, and R is a set of rules. For examples of small context-free grammars and annotated sentences we refer to the book of Jurafsky and Martin on page 264 [23].

Combinatory Categorial Grammar

Looking at normal constituency grammars one can see that a large number of production rules are required to capture the syntax of a sentence satisfiably. Lexcalized grammars like the Combinatory Categorial Grammar (CCG) aim to resolve such issues by introducing word categories binding production rules to words [25].

Definition. *We define the set of all word categories $\mathcal{C}$ as follows:*

- *$\mathcal{A} \subseteq \mathcal{C}$, where $\mathcal{A}$ is the given set of atomic elements*
- *$(X/Y),\ (X\backslash Y) \in \mathcal{C}$, if $X,\ Y \in \mathcal{C}$*

We note that (X/Y) is a function looking for a constituency Y to its left and returning X while $(X\backslash Y)$ does the same but seeks Y to the right.

While including rules in word categories reduce the set of production rules, they also yield many words with multiple possible categories in the lexicon. The assignment of categories to words is called super tagging and is an active field of research [26].

Rules define functions on word categories. In the following, we look at the two rule templates which form the basis for all rules in a CCG.

$$X/Y \; Y \Rightarrow X \tag{2.4a}$$

$$Y \; X\backslash Y \Rightarrow X \tag{2.4b}$$

The first rule applies the function to its argument on the right, while the second applies to the left. These directions are referred to as forward and backward function applications, respectively [23].

In addition to direction, we introduce two function types. Composition is the combination of neighboring functions:

$$X/Y \; Y/Z \Rightarrow X/Z \tag{2.5a}$$

$$Y\backslash Z X\backslash Y \Rightarrow X\backslash Z \tag{2.5b}$$

While type raising substitutes a simple category with a function taking the original category as an input:

$$X \Rightarrow T/(T\backslash X) \tag{2.6a}$$

$$X \Rightarrow T\backslash(T/X)) \tag{2.6b}$$

Here T can be any word category, including functional categories.

Now we can look at the CCG derivation in Figure 2.3. The directions of applications are shown by $>$ and $<$, while the function types are labeled with the capital letters B for composition and T for type raising. CCG parse trees can be created relatively easily and efficiently by the use of an A^* parser from a tagged lexicon of words [18].

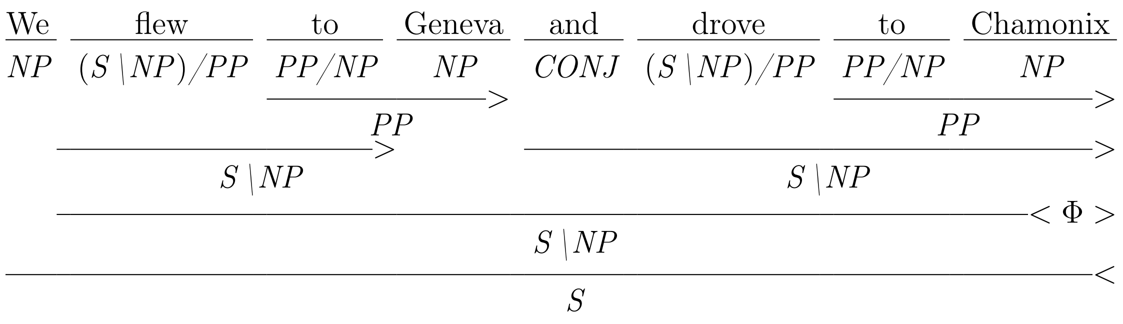

We flew to Geneva and drove to Chamonix
NP (S\NP)/PP PP/NP NP CONJ (S\NP)/PP PP/NP NP
PP >
PP >
S\NP >
S\NP >
S\NP < Φ >
S <

Figure 2.3.: The CCG derivation for "We flew to Geneva and drove to Chamonix" from the book of Jurafsky and Martin [23]. The Φ marks a conjunction operation between two-word categories.

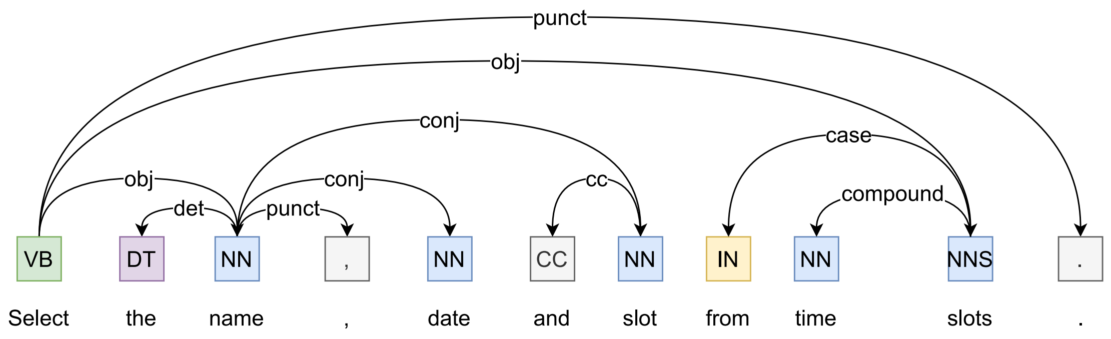

punct
obj
conj
case
obj
det
conj
punct
cc
compound
VB DT NN , NN CC NN IN NN NNS .
Select the name , date and slot from time slots .

Figure 2.4.: A dependency tree of the sentence "Select the name, date, and slot from time slots." made by the Stanford CoreNLP parser [27].

2.2.2. Dependency Grammars

Dependency grammars, in contrast to constituency grammars, use the notion of semantic dependency to capture a sentence. A semantic dependency is a hierarchical relation between words consisting of a head and dependents describing their semantic connection. This works exceptionally well for subject and object dependencies making dependency parsing well-suited for information extraction.

Universal Dependencies (UD) is currently the most popular dependency grammar as it is efficiently parsable and applicable to multiple languages [28]. As describing all dependencies in UD is beyond the scope of this work, we provided a list of all dependencies of (UD) in Appendix B with links to the corresponding help pages in the electronic version of this work.

2.3. Machine Learning Methods

2.3.1. Nearest Neighbor Classifier

The nearest neighbor classifier is one of the simplest supervised machine learning algorithms. It assigns the label of the trained feature vector closest to our input.

Let Y be the set of all labels, $\hat{X}$ the set of trained feature vectors, and $X \subseteq \mathbb{R}^n$ the set of all possible features vectors. For a training set of feature label pairs $(X_i, Y_i) \in \hat{X} \times Y,\ i \in \mathbb{N}$, we want to find $y^* \in Y$, the true label for an unknown feature vector $x \in X \setminus \hat{X}$ with the following function:

$$\hat{f}(x) = Y_k,\ k = \arg\min_k(d(x, X_k)), \tag{2.7}$$

where d is some distance function on $\mathbb{R}^n$.

As one can see from the definition, we assume that all possible labels have at least one feature vector in the training set. Otherwise, the algorithm can't make a reliable prediction for some labels since there is no feature vector we could measure the distance to.

The neighborhoods around the trained feature points in $\hat{X}$ defined as

$$\mathit{neighborhood}(X_k) = \{x \in X \mid d(x, X_k) < d(x, X_j),\ \forall X_j \in \hat{X},\ j \neq k\ \in \mathbb{N}\} \tag{2.8}$$

are called Voronoi regions. In these regions, all points are classified as the corresponding Y_k in Y belonging to X_k.

2.3.2. Logistic Regression

Logistic regression is a statistical method modelling the probability $p(X)$ of one feature vector X belonging to one of two categories $Y = \{0, 1\}$ [29]. We will look at its simplest form where $X \in \mathbb{R}$. One can motivate logistic regression with trainable parameters $w,\ b \in \mathbb{R}$ as a normalization of the linear regression

$$p(X) = wX + b \tag{2.9}$$

to the interval $[0, 1]$ by using the following sigmoid function:

$$\sigma(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}},\ x \in \mathbb{R} \tag{2.10}$$

called the logistic function [29]. This results in the following form:

$$p(X) = \frac{1}{1 + e^{-(wX+b)}} \tag{2.11}$$

As one can see in Figure 2.5 small and large values are assigned probability values close to zero or one, respectively. The steepness of the function is defined by w while the center is moved using the parameter b.

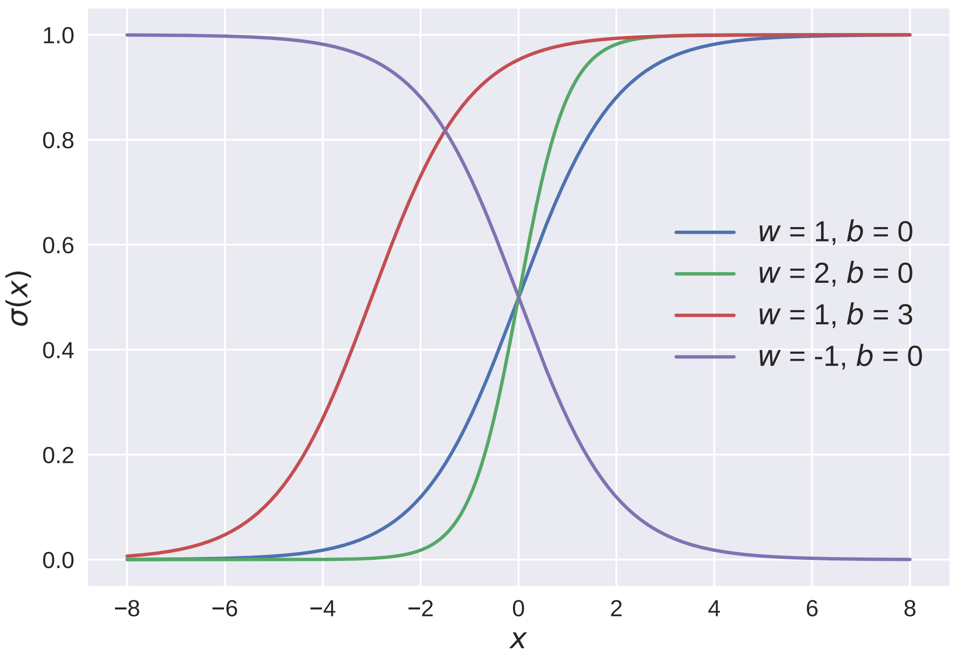


Figure 2.5.: The logistic function for different choices of parameters w and b.

Maximum Likelihood Estimation

To find parameters w, b which best fit to the training data $(X_i, Y_i) \in \hat{X} \times Y$, one can use maximum likelihood estimation. For N data points, let us define the likelihood function $\mathcal{L}(w, b)$ for two possible outcomes:

$$\mathcal{L}(w, b|\hat{X}) = \prod_{i:Y_i=1} p(X_i) \prod_{i:Y_i=0} (1 - p(X_i)) \tag{2.12a}$$

$$= p(Y = 1, (w, b))^{Np^*(Y=1)} p(Y = 1, (w, b))^{Np^*(Y=0)}, \tag{2.12b}$$

where p^* is the real observed probability and p is the model probability dependent on w and b. The likelihood function measures the probability that Equation (2.11) obtains the set of feature vectors in $\hat{X}$. Thus maximizing $\mathcal{L}$ yields the best fit of Equation (2.11) on the training data.
The maximum can be estimated using the gradient ascent method:

$$\mathcal{L}(w_{n+1}, b_{n+1}) = \begin{bmatrix} w_n \\ b_n \end{bmatrix} + \gamma \nabla \mathcal{L}(w_n, b_n), \tag{2.13}$$

where $\gamma \in [0, 1]$ is called the learning rate.

2.3.3. Perceptron

The perceptron is a type of artificial neuron for binary classification tasks introduced by Warren McCulloch and Walter Pitts in 1943 [30]. The inspiration were real neurons that fire when a specific electrical current is reached. Instead of current, the artificial neuron uses a weighted sum as its activation. Then, like a real neuron, it would fire when a certain threshold has been reached.

Using this thought process McCulloch and Pitts devised the following classifier function for labels $Y = \{0, 1\}$:

$$f(x) = \begin{cases} 1 & \text{, if } \sum_{i=1}^{n} w_i x_i + b > 0 \\ 0 & \text{, if } \sum_{i=1}^{n} w_i x_i + b \leq 0 \end{cases}, \tag{2.14}$$

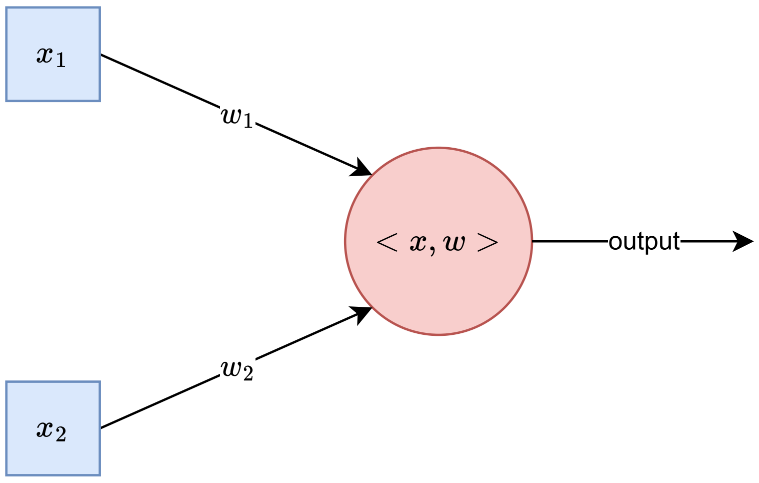


Figure 2.6.: Graphical representation of a perceptron with 2 inputs.

where $x \in \mathbb{R}^n$ is the input, $w \in \mathbb{R}^n$ the weight vector and $b \in \mathbb{R}$ the threshold parameter or bias. If the output $\hat{y}$ of input x with given weights w was false, the weights are updated using this formula:

$$w_{n+1} = w_n - \gamma(\hat{y} - y^*)x, \tag{2.15}$$

with $y^* = 1 - \hat{y}$ being the real output. $(\hat{y} - y^*)$ above can be interpreted as the error or loss of our perceptron.

As one might have realized the activation in Equation (2.14) is nothing else than a linear regression (see 2.9). We can use the the logistic function 2.11 to have a smoothed output. In this case, the logistic function σ is called the activation function. Though instead of taking Equation (2.15) to update our weights, we will need to define a loss function for logistic regression and minimize it.

Logistic Regression with a Single Neuron Perceptron

As we have already seen, one can create a perceptron such that it is equivalent to logistic regression. The issue is to reformulate the maximum likelihood estimation into something describing a loss. Cross entropy can be very broadly interpreted as the "distance" between two distributions p and q defined on Y:

$$H(p,q) = -\sum_{y \in Y} p(y) \log q(y) \tag{2.16}$$

We can then assign q the estimated distribution of the perceptron and p the real distribution on our data set to have the "distance" between the estimate and the real observation. It turns out, that minimizing cross entropy is equivalent to maximizing the likelihood. We see that by generalizing Equation (2.12b) to Y, taking the logarithm and dividing by the number of measurements N:

$$\frac{1}{N}\log(\mathcal{L}(w,b)) = \frac{1}{N}\log \prod_{y \in Y} q(y,(w,b))^{Np(y)} = \sum_{y \in Y} p(y) log(q(y,(w,b))) = -H(p,q) \tag{2.17}$$

In the example of our perceptron, computing the cross entropy is easy. Because we have measurements with two outcomes, we can safely assume that our data is Bernoulli distributed. As such, we know that $p \in \{y^*, 1-y^*\}$ for the real output $y*$ and $q \in \{\hat{y}, 1-\hat{y}\}$ for our approximated output. Consequently, this yields the loss function $C(\hat{y}, y^*)$:

$$C(\hat{y}, y^*) = -y^* \log \hat{y} - (1-y^*)\log(1-\hat{y}) \tag{2.18}$$

As a result, we update our perceptron weights using gradient descent as follows:

$$w_{n+1} = w_n - \gamma \nabla C(\hat{y}, y^*), \tag{2.19}$$

2.3.4. Feedforward Neural Networks

Feedforward neural networks are a set of artificial neurons described in Section 2.3.3. They consist of layers of which each can have multiple neurons. We will look at fully-connected neural networks (see Figure 2.7) where each neuron of a layer is connected to every neuron of the previous layer and the layer after it.

The layers can be interpreted as vectors i, h, and o for the input layer, hidden layers, and output layer or a when meaning the layers in general. The

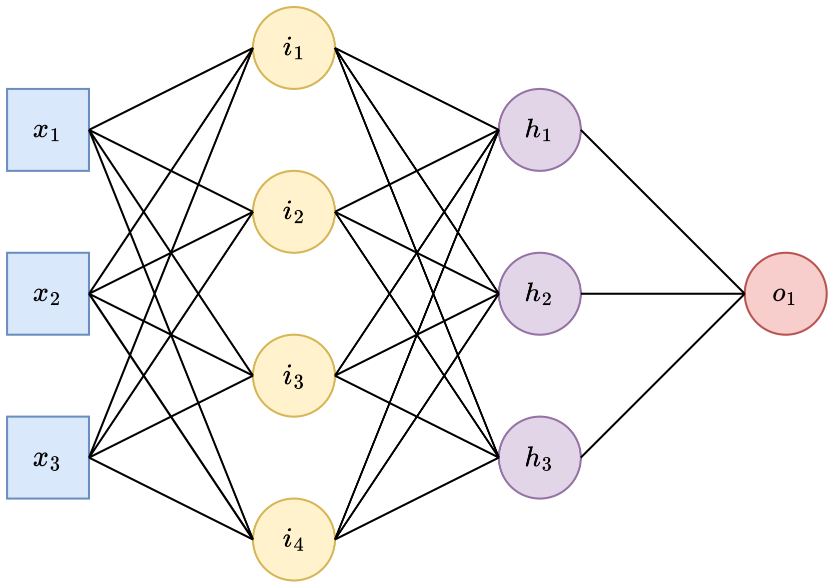


Figure 2.7.: Graphical representation of a 3 layer fully-connected feedforward neural network with a single output neuron

activation of each layer will be fed to the next layer, thus the name feedforward network. In contrast to Section 2.3.3, the weights are matrices of shape (previous layer size) $\times$ (current layer size) and the activation function is applied to the vector of the current layer's activations. Therefore, there can be multiple activation functions for each layer in the network.

Backpropagation

Backpropagation is a training method for minimizing the cost function C using both the hidden as well as the input and output activations [31]. The problem is to compute the gradient of the loss $\nabla C(o, y^*)$ with respect to the weights w^l_{jk} and biases b^l_j where l denotes the layer. This is done using the chain rule:

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \tag{2.20}$$

$$\frac{\partial C}{\partial b^l_j} = \frac{\partial C}{\partial a^l_j} \frac{\partial a^l_j}{\partial b^l_j} \tag{2.21}$$

with a_j^l the neuron activation defined as

$$a_j^l = \sum_{k=1}^{m} w_{jk}^l a_{jk}^{l-1}. \tag{2.22}$$

with m the length of the layer.
This yields the partial derivatives for the weights and biases:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial a_j^l} a_k^{l-1} \tag{2.23}$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial a_j^l} \tag{2.24}$$

Since each derivative requires the activation of the previous layer, one has to propagate backward through the network (e.g. Figure 2.7), hence the name.

2.3.5. Metric Learning

Metric learning aims to create a real-valued metric that fits a particular problem's needs. Such a metric can be for example the Mahalanobis distance $d_M(x, x') = \sqrt{(x - x')^T M (x - x')}$ which is used by most methods [32]. In particular, the positive semi-definite (p.s.d.) matrix M of d_M is being learned. Consequently, the p.s.d. M can be Cholesky decomposed into $L^T L$ [33].

We will look in particular at two methods, neighborhood component analysis (NCA) [34] and large margin nearest neighbors (LMNN) [35] which both use d_M to fit a problem search space and are both related to the nearest neighbor classification problem discussed in Section 2.3.1. For this section, we will follow the survey by Bellet et al. [32].

Neighborhood Component Analysis

The neighborhood component analysis seeks to minimize a cost function based on a stochastic nearest neighbor method's leave-one-out (LOO) error. LOO is computed from a data set of N points by creating all possible subsets containing $N-1$, training a model for each of these subsets, and summing all squared distances between the predicted value and real value. In our case, the distances would be computed using d_M:

$$C_{\text{LOO}}(x) = \sum_{i=1}^{N} d_M(\hat{f}_i(x) - y)^2 \tag{2.25}$$

As Goldberger et al. [34] point out, C_{LOO} is discontinuous and thus unsuited as an error function. Instead they applied the softmax function, a generalization of the logistic function 2.11 for multinomial problems

$$\sigma(y)_i = \frac{e^{\beta y_i}}{\sum_{j=1}^{n} e^{\beta y_j}}, y \in \mathbb{R}^n \tag{2.26}$$

to C_{LOO}. This yields a probability p_{ij}:

$$p_{ij} = \frac{exp(-\|Lx_i - Lx_j\|^2)}{\sum_{k \neq i} exp(-\|Lx_i - Lx_k\|^2)} \tag{2.27}$$

modelling if points x_i, x_j are neighbors. They then use a maximum likelihood estimator (see Section 2.3.2) to fit $M = L^T L$ to the training data. Unfortunately, a major drawback of NCA is that it not always finds the global maximum. This is due to the maximum likelihood estimation of p_{ij} being nonconvex [32].

Large Margin Nearest Neighbor

The large margin nearest neighbor (LMNN) [35] method is a very natural interpretation of learning a metric. Here a k-nearest neighbor classifier with euclidean distance is used to maximize the distance of points of different labels while minimizing it for points with the same label.

The difference between the nearest neighbor and the k-nearest neighbor algorithm is that k-nn takes a vote between the k nearest points to the input. Therefore, Voronoi regions are here defined as the regions where the vote always ends with the same label being selected. For LMNN, this means that points in the k-Voronoi regions with a different label than the result of the vote get pushed away while feature vectors with the same label are pulled closer. Bellet et al. [32] formally define LMNN as follows:

$$\mathcal{S} = \{(x_i, x_j) : y_i = y_j, \text{ and } x_j \text{belongs to the k-voronoi region of } x_i\}, \tag{2.28}$$

$$\mathcal{R} = \{(x_i, x_j, x_k) : (x_i, x_j) \in \mathcal{S}, y_i \neq y_k\} \tag{2.29}$$

with the following convex program:

$$\min_{M \in S_+^d} \quad (1-\mu) \sum_{(x_i,x_j)\in\mathcal{S}} d_M(x_i, x_j)^2 + \mu \sum_{i,j,k} \xi_{ijk} \tag{2.30}$$

$$\text{s.t.} \quad d_M(x_i, x_k)^2 - d_M(x_i, x_j)^2 \geq 1 - \xi_{ijk} \qquad \forall (x_i, x_j, x_k) \in \mathcal{R} \tag{2.31}$$

where $\mu \in [0, 1]$ is the push/pull tradeoff [32]. In contrast to NCA, LMNN is convex and always finds the optimal solution but can struggle with overfitting due to a lack of regularization [32].

2.3.6. Attention

Language has many unique properties that must be considered when making a machine learning algorithm. For one, inputs (e.g. sentences) are of variable length. Furthermore, these inputs contain semantic dependencies between parts of the input: "amazingly bad" means the polar opposite of "amazingly good" even though both sequences contain "amazingly" which itself is a very positive adverb. Only the combination of both words to an adverbial phrase locks its meaning for the reader.

While LSTMs (see appendix A) are able to preserve such dependencies through the whole depth of the sequence, they still lose part of the information due to the computations made inside the LSTM cell. Attention solves this issue completely by looking at the whole sequence jointly and producing a probability distribution that measures the alignment of two inputs in the sequence of inputs.

Initially envisioned as a method to enhance encoder-decoder networks [36, 37], attention cells are now known to be the heart of transformer networks [38]. We will describe transformers by following the pertinent sections in chapter 9, "Deep

Learning Architectures for Sequence Processing" of the book by Jurafsky and Martin [23] in the following two sections.

Let x_i, x_j be the input vectors for which we want to compute the proportional relevance α_{ij}:

$$\alpha_{ij} = softmax(x_i \cdot x_j), \tag{2.32}$$

called attention. Using this attention measure, we can return an output value y_i by computing a weighted sum over all $x_i, x_j : \; j \neq i$. Another form of attention is introduced by Bahadanau et al. [36] called additive attention that uses a sum instead of a vector product.

Transformers use a generalized version of the relevance measures above [38]. Here all inputs x_i are projected into three different roles [23]:

- As the query, x_i is the current focus of attention compared to all other inputs $q_i = W^Q x_i$.
- As a key vector, x_i is to be compared against the current query vector $k_i = W^K x_i$.
- As a value vector, x_i is used to compute the output for the current focus of attention $v_i = W^V x_i$.

To be more precise, one must mention that using the same input x_i for all attention inputs is called self-attention. The query and key vectors have length d_k while the value vector has length d_v. In its simplest form, the dimensionality of all vectors is the same. Using these projections, we get the following equations for α and y_i:

$$\alpha_{ij} = \text{softmax}\left(\frac{q_i \cdot k_j}{\sqrt{d_k}}\right) \tag{2.33a}$$

$$y_i = \sum_{j \leq i} \alpha_{ij} v_j \tag{2.33b}$$

Because the dot product in Equation (2.33a) can be arbitrarily large, it is scaled by the square root of the length of the key and value vectors. This is called scaled dot attention and ensures minimal gradient loss. Transformers stack multiple self-attention models and concatenate the outputs, which then get projected linearly down to the output dimension of one transformer model. Each model in this stack is now called attention head, and the conglomerate is called multi-head attention.

One of the advantages of this method beyond solving the vanishing gradient problem better than LSTMs is the ease of computation for multi-head attention layers compared to encoder-decoder networks. The query key and value vectors can be stacked, and the computation then simplifies to two matrix products and the application of the softmax function, both of which can be done natively on GPUs and TPUs [38].

2.3.7. Transformers

Transformers are ensemble decoder encoder models consisting of multiple transformer cells introduced by Vaswani et al. [38]. Transformer cells are made up of feedforward networks stacked on top of a multi-head self-attention model with residual connections and layer normalizations [38] (see Figure 2.8).
Residual connections jump layers and thus feed information directly to the next layer, which could otherwise get lost by computing the gradient in training. Layer normalizations can be functions like the ones defined by Ba et al. [39] and bound the output of the layers in the cell. We will look at a variation of the z-score defined as follows:

$$\text{layernorm}(x) = \gamma\left(\frac{x-\mu(x)}{\sigma(x)}\right) + \beta \tag{2.34a}$$

$$\sigma(x) = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i-\mu)^2} \tag{2.34b}$$

$$\mu(x) = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \tag{2.34c}$$

μ is the mean, σ the standard deviation, γ the trainable gain, and β the trainable bias.

Beyond the introduction of the transformer architecture, Vaswani et al. [38] define the following bounded positional encoding:

$$PE_{(\mathrm{pos},2i)} = \sin(pos/10000^{2i/d_{\mathrm{model}}}) \tag{2.35a}$$

$$PE_{(\mathrm{pos},2i+1)} = \sin(pos/10000^{2i/d_{\mathrm{model}}}), \tag{2.35b}$$

where pos is the position of the input and d_{model} the length of the model output vectors. This function is used to enhance the sequential transformer inputs by including positional information.

2.3.8. Text Embeddings

Text embeddings have become one of the most used tools in natural language processing having improved the state-of-the-art on multiple occasions [8, 17, 9]. Here, words are transformed into vector meaning representations for a multitude of downstream tasks. We will describe two of the most important text embeddings, "GloVe" [8] and "BERT" [9]. Furthermore, we will cover an evolution of BERT, which could provide a useful alley for improving our model on a limited data set called "ELECTRA" [40].

GloVe

Global Vectors for Word Representation or GloVe is a log-bilinear model capturing word co-occurrences in a given word corpus [8]. To be more specific, GloVe uses co-occurrence ratios as a semantic measure of similarity. This enables GloVe to embed words as long as they are in its 42B token corpus.

The paper uses the concept of thermodynamic phase to motivate why to use ratios instead of simple co-occurrence. Suppose we have the words "ice" and "steam". If one were to use probe words like "solid" which have much more to do with ice rather than steam, the ratio $\frac{P(solid|ice)}{P(solid|steam)}$ should be high. Conversely, for "gaseous" it should be small. For words that are related to both or neither (the paper uses "water" and "fashion"), the ratio should be close to one.

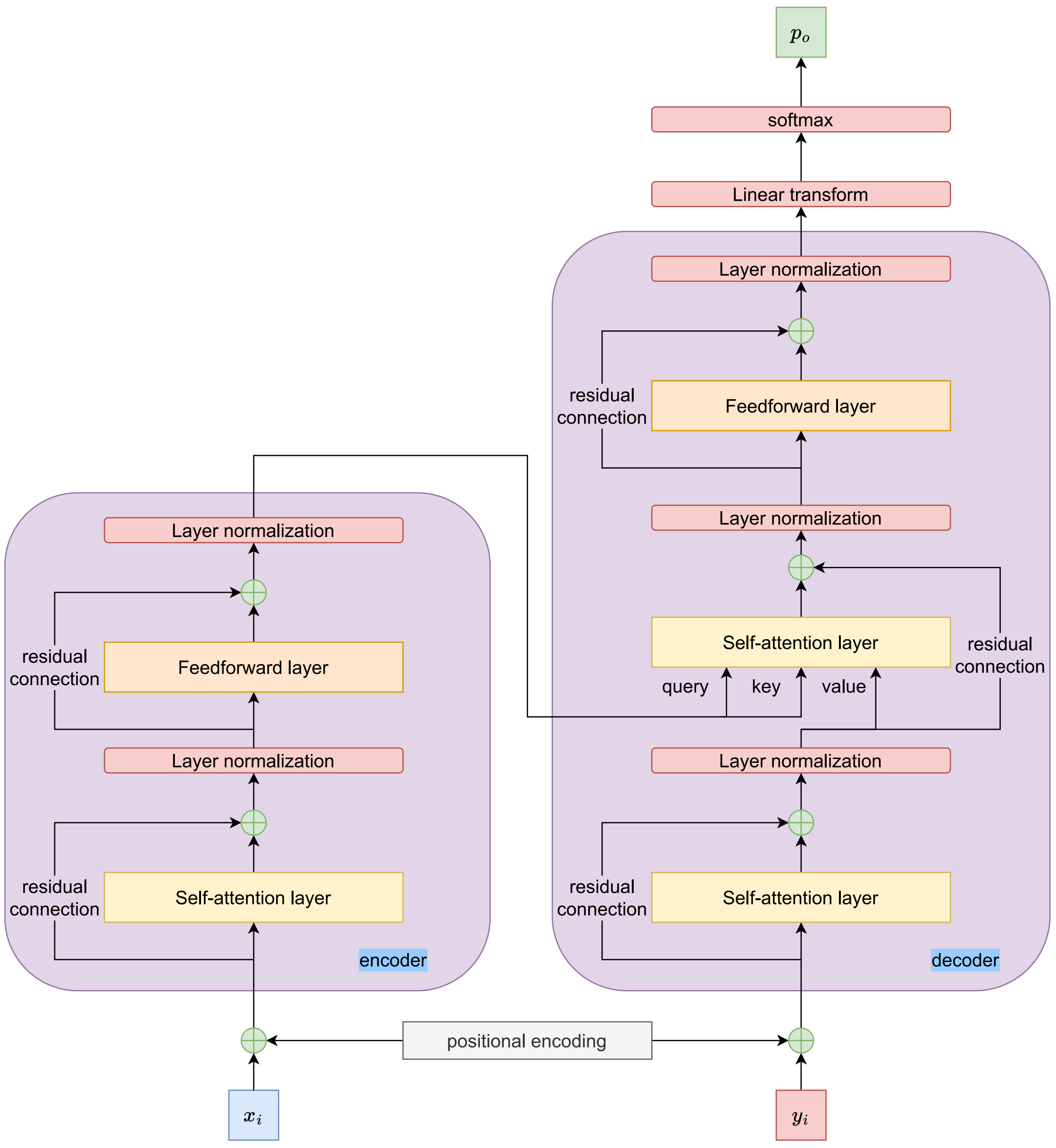


Figure 2.8.: Overview of a transformer. p_o is the output probability. The inputs x_i and y_i are word embeddings. Adopted from [38].

This is indeed a desired outcome since "water" might have something to do with both words but is not pertinent in the context of phase. Thus this method correctly evaluates the low value 1 instead of having some high value when proper co-occurrence is used. Therefore the paper reasons that ratios capture meaning in relation to some context better than simple co-occurrence.
For model training, the cost function is the following weighted least squares model:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b_j} - \log X_{ij})^2, \tag{2.36}$$

where V is the vocabulary size, X is a matrix counting word-word co-occurrences, and X_{ij} is the count for words i and j. w_i is the word context vector looking at co-occurrences to the left and $\tilde{w}$ looking to the right with b_i and $\tilde{b_j}$ the respective biases. If the windows for counting co-occurrences have the same size for both w and $\tilde{w}$, X is symmetric, and both word vectors are equivalent. The paper showed the best performance using equal context windows and summing both w and $\tilde{w}$.
The weighting function f is defined as:

$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & , ifx < x_{\max} \\ 1 & , \text{otherwise} \end{cases}, \tag{2.37}$$

$x_{\max}$ is a cutoff parameter, and α is used to introduce nonlinearity. The paper measured that performance for $\alpha = 3/4$ is best and used the cutoff $x_{\max}$ for all models. The function f was chosen to fulfill some properties set by the authors:

1. $f(0) = 0$ and $\lim_{x \to 0}(f(x) \log^2 x)$ is bounded for continuous f.
2. f is monotonically nondecreasing s.t. rare co-occurrences are not overweighted.
3. f is relatively small for large x so as not to overweigh frequent co-occurrences

GloVe improved the state-of-the-art in multiple tasks such as word analogy and word similarity. In word analogy, a classifier fills sentences "a is to b like c is to _". Here GloVe reached an accuracy of 81.9% beating the next best classifier by 13 percent.

BERT - Bidirectional Encoder Representations from Transformers

BERT is the seminal word embedding model by Devlin et al. [9]. Inspired by the success of other contextualized word embeddings from RNNs by Peters et al. [41, 17], the idea is to generate contextualized word embeddings using transformers. BERT improved the state-of-the-art in 11 natural language processing tasks, such as the F1-measure on the Stanford Question Answering Dataset version 1.1 [42], and version 2 [43] by 1.5 points absolute and 5.1 points absolute respectively.

The success of BERT using relatively simple models combining pre-trained or fine-tuned embeddings with feedforward neural networks beating much more complex encoder-decoder architectures inspired a multitude of papers [44, 40, 6, 5, 7] and is still an active area of research today.

At inference, the input document words get split into tokens, and two special tokens are introduced: `[CLS]` signals the start of the document while `[SEP]` is the sentence separator and document end token. The subword tokens get embedded by the word piece embedder of Wu et al. [45] and summed with the transformer positional encodings before being fed to a network of stacked transformer embedder cells.

In the original paper, the model had two configurations, where L is the number of transformer encoder cells, H is the size of the hidden vectors, and A is the number of self-attention blocks:

- BERT$_{\text{BASE}}$: $L = 12,\ H = 768,\ A = 12$
- BERT$_{\text{LARGE}}$: $L = 24,\ H = 1024,\ A = 16$

Furthermore, we define a smaller version referenced by [46] BERT$_{\text{SMALL}}$ with $L = 12,\ H = 512,\ A = 8$.

Beyond the use of transformers, training is another departure from previous text embedding systems. Rather than learning the probability that a sequence is being predicted by the text embedder [17] or the co-occurrence task defined in

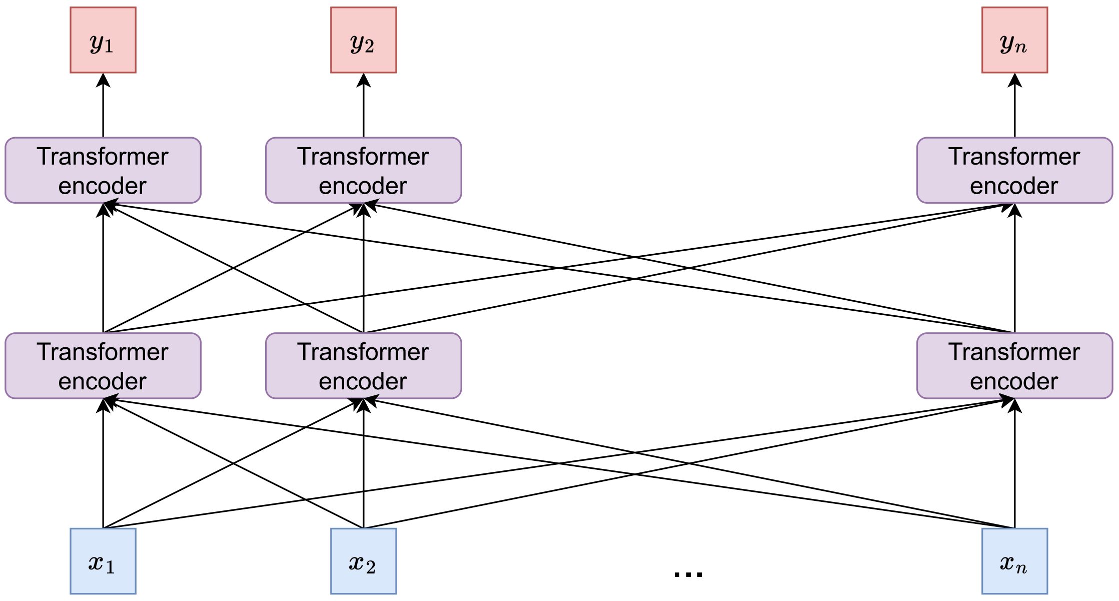


Figure 2.9.: A conceptual overview of BERT. One can see that the model feeds inputs in both sequence directions, making it bidirectional in contrast to [47]. Adopted from [9].

Section 2.3.8, BERT has two training objectives. The Masked Language Model (MLM) task is to find an obfuscated token in a sentence (see Figure 2.10). In Next Sentence Prediction (NSP) training, the model should find the next sentence sequence in a given document of at least two sentences.
In addition to the novel training methods, BERT adopts a system of pre-training and fine-tuning like [47]. In fine-tuning, the MLM and NSP training tasks get adopted to something fitting the target problem. An example of this is to predict the answer to a question instead of the following sentence in the NSP task for question answering. This enables BERT to be used in many domains and downstream tasks effectively since the text embedding can be fit to the problem at hand.

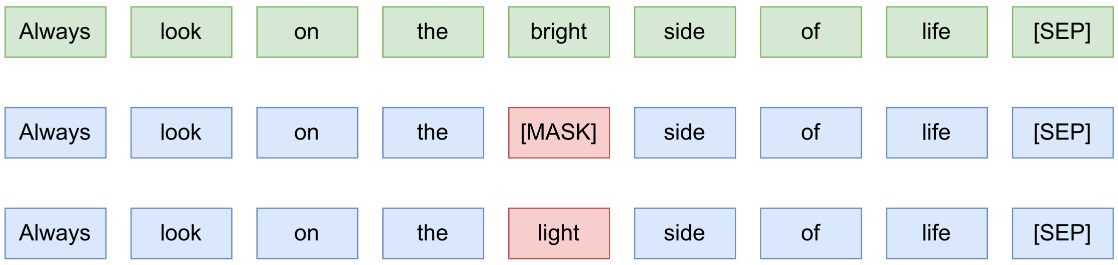


Figure 2.10.: The example sentence from the data set (green), the BERT masked language model task where the `[MASK]` token is to be predicted, and an example of a BERT prediction for the adversarial training of ELECTRA.

ELECTRA

Electra replaces the MLM training task of BERT with an adversarial approach [40]. At training, two systems - the generator, and the discriminator, are used.

The generator is a small BERT embedder which, instead of the `[MASK]` tokens, predicts a word like in the MLM task (see Figure 2.10). The discriminator is the actual ELECTRA embedder which replaces the word of the generator if it predicts the word is false. This approach leads to substantial savings in training data and computing requirements while retaining or even improving the embedding quality when compared to BERT and most of its other evolutions with the same amount of data and compute [40].

3. Related Work

As mentioned, semantic parsing is the translation of natural language to some chosen meaning representation. Since we want to create an interactive and continually learning semantic parser for spreadsheet applications, we will look at two topics: table embedding and interactive semantic parsing.

3.1. Table Search Using a Deep Contextualized Language Model

Chen et al. [5] devised a method for using BERT language models to embed tabular data, which can then be used for natural language table search. The approach is made up of 3 parts: The content selector selects the most pertinent table items. A feedforward neural network creates a vector presentation from hand-selected context information of the table. The BERT embedding language model takes a custom-made document from the selected items, the query, and some context information and creates an embedding from them. The final piece is a regression layer assigning each concatenation of the outputs from the BERT embedder and the feedforward neural network a probability of how much they agree with the original query.

The content selector embeds the items from the table (e.g., cells, columns, column names, etc.) and uses a salience measure on each of these items and the embedded input query to find the most pertinent table contents. Then the selected items are put into a custom document with the input query and jointly passed to the BERT embedder. We describe the structure of the document later in Section 4.7.2 as we will use this method as part of our system. Very broadly, the system uses the `[SEP]` separator token of BERT and appends a list of `[SEP]` separated selected items to the query token sequence.

The whole system is trained jointly, fine-tuning the BERT embedding mechanism, and uses example queries and their corresponding tables to fit the correct agreement probability for the final output.

3.2. TABERT: Pretraining for Joint Understanding of Textual and Tabular Data

TABERT is another system for embedding table information using BERT embedding language models. The model devised by Yin et al. [6] is very similar in style tot he model of Chen et al.. As part of the evaluation of this paper, a SQL parser was created using TABERT embeddings. The parser showed state-of-the-art performance on the SPIDER text-to-SQL dataset [48] and the WikiTableQuestions dataset [10].

In comparison to Chen et al., only the BERT embedding model is used to embed all the information from the table and table context. Furthermore, instead of pre-selecting cells, whole rows are selected. Still, here too, a salience score is used between the input query and selected rows to find the most pertinent rows, which then get joined into a custom document, almost the same as Chen et al. proposed and we explain in Section 4.7.2.

Where this work significantly differs from Chen et al. is in the training objective. Instead of jointly training the inputs with the downstream implementation, TABERT enables pre-training using an agnostic training method called masked column prediction. Analog to the masked language model pre-training task of BERT, the language model is tasked to fill a masked column name and datatype using the column contents.

Unfortunately, large datasets without much noise are needed for this training method to work. Therefore, Yin et al. had to use heuristics to cull noisy examples from their data. The following paper aims to improve these issues by artificially generating training examples.

3.3. TAPEX: Table Pre-training via Learning a Neural SQL Executor

TAPEX is another language model used to embed tabular data by Liu et al. [7]. It improved the score set by TABERT in the WikiTableQuestions benchmark.

TAPEX has some significant differences in how it encodes tables compared to TABERT and the approach by Chen et al.. First, it uses a different transformer-based encoder called BART [49] and does not use any feature selection beforehand, instead using the whole table as an input. Also, the structure of the table input is changed. Here the tokenizer of the system uses two new special tokens called `[ROW]` and `[HEAD]`, which mark the column name list and each row such that the result is the table head marked by the `[HEAD]` token followed by rows marked with the `[ROW]` token. No natural language input query is used for TAPEX.

In addition, the training target is changed. TAPEX learns encodings with the use of SQL templates which are executed on randomly sampled inputs from a given input table. The execution results are used as a label which the model aims to predict. This enables the system to constantly generate new instances for training, thus improving the data efficiency of the method.

3.4. An Imitation Game for Learning Semantic Parsers from User Interaction

In this work, a strategy for creating a continually learning semantic parser from user interaction is proposed by Yao et al. [50]. The authors introduce the notions of state and action to learn new instances from interaction. An action is a possible subaction a semantic parser performs on the way to translate the input utterance. This can be, for example, filling an empty slot in a target SQL code template. The state is defined as the progress of the translation process defined by all actions executed before it. The initial state corresponds to the untranslated natural language input.

Suppose we have a parser that can return a probability distribution for all trained actions for a given state. The paper states that then a probability for a set of actions can be defined as the product of all preceding subactions. If this probability falls below a certain predefined threshold, the system would ask the user for guidance according to the current state, e.g., the slot that in this step needs to be filled in in our example above. The system then provides the user with a selection of the most probable next actions as well as an answer telling the system no suggested action is correct.

To stay in our example, selecting one of the suggestions is interpreted as the user providing the correct value to be filled in the slot. Then the system saves this answer when the interaction is finished and retrains itself. When it reaches the same state as in the original interaction, it assigns the subaction chosen by the user a higher probability in retraining. If the user selects the answer, telling the system all suggested actions are incorrect, the system reduces the probability of all the suggested actions in this state. This results in the probability distribution of the parser being corrected towards the values given by the user for particular states.

The system achieves this by using a weighted negative log-likelihood of the function assigning the probability distributions for each state. If the action in the state was selected by the user, weight 1 is assigned. Conversely, the weight is 0 is assigned for all false suggestions made by the parser in the interaction phase when the user answers that no suggestion is correct.

3.5. Learning Adaptive Language Interfaces through Decomposition

The main inspiration of this work is the system for learning new instances continually from decomposition defined by Karamcheti et al. [1]. The paper discusses a problem setting where a house helper robot should be able to understand unknown tasks by asking the user to provide a description of each substep required to fulfill the task. The main idea is only to provide a very limited set of atomic actions from which the parser can generalize while in use. We will follow the general structure of this paper, adjusting it to our problem.

The authors have devised a system consisting of 4 substeps we call entity abstraction, candidate resolving, candidate combination, and candidate reranking. At the entity abstraction step, the system defines a dictionary of lifted objects it can interact with. Lifted objects are the main word stem of all composite words describing interactable objects in the context of the robot e.g., "sink" for the composite "kitchen sink". The abstraction mechanism then tries to match all occurrences of these words in the input utterance and replaces them with tags signaling that a context object was at this position. The result is a lifted utterance and a set of inputs that the parser will later use for the predicted program.

The program resolver translates the lifted utterance by embedding it using summed GloVe word vectors. Then an approximate nearest neighbor search on trained embedded lifted utterances is done using cosine similarity as a measure. Each trained utterance has a respective candidate program template assigned to it. The result is a list of candidate program templates and similarities within a predefined threshold. The similarities are given to a logistic regressor (see section 2.3.2), computing the probability for each similarity that it belongs to two embedded vectors translating to the same candidate program template. The candidate program templates then get filled with the extracted objects found by the entity abstraction mechanism in the combination stage. The resulting programs we call grounded.

Afterward, in the candidate reranking stage, the grounded programs are executed in a simulator removing all programs that fail to run. The remaining programs are then reranked using a two-layer feedforward network acting as a regressor. As an input to this regressor for each candidate program, the summed GloVe embeddings of the candidate program and initial utterance, as well as a bag-of-words vector from the context objects, are given. The final result is a list of candidates with respective probabilities assigned by the reranker. From this list, the most probable answer is given.

If no trained utterance is found within a threshold in the nearest neighbor search, the parser returns a question asking the user to decompose the task into substeps. Given this list of substeps, each step is parsed by the system. After the resolver stage, the program templates are combined and added as a new program assigned to the original lifted utterance corresponding to the composite task. Afterward, the resolver and reranker regressors are retrained using this new instance as the correct translation of the original composite utterance. We have provided a graphical overview of this model in Figure 3.1.

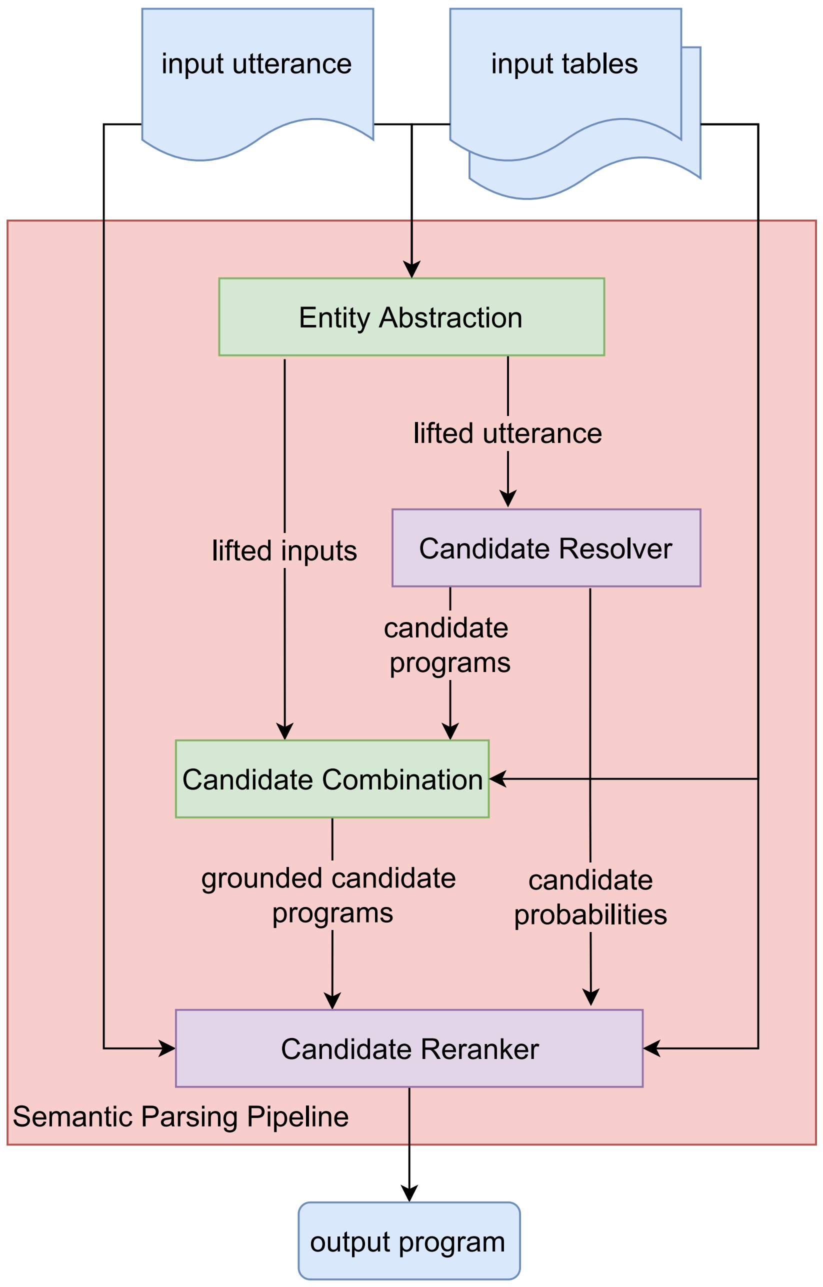


Figure 3.1.: Graphical overview of the semantic parser. Purple cells are machine learning models, while green cells are rule-based methods.

4. Approach

4.1. Challenges

While the structure of the parser pipeline as described in Section 3.5 is simple enough to understand, there are multiple challenges adapting [1] to our problem. For one, spreadsheet programs can be used in many different domains and have a broader scope than the house helper robot suggested by the original paper. Consequently, we have to use another approach for capturing the table state then provided by the paper. Furthermore, the table state has to be a vector such that we can use it for our machine learning reranker.

Secondly, many utterances for spreadsheet programs are similar in form - especially when removing all pertinent objects from the sentence. In practice, this means that operations like removing and selecting columns from a table only have a verb differentiating them. This similarity of instances leads to a situation where text embedders assign similar, harder to separate vectors making it hard for a nearest neighbor classifier to find the correct candidate programs.

Furthermore, limiting the amount of training data has multiple knock-on effects. For text embedding, fine-tuning is not feasible. In entity abstraction, we cannot do machine learning for lifting utterances and grounding programs without the need for a large set of annotated sentences from the domain. In reranking, we cannot use data-hungry processes like proposed in [7] and instead have to live with constrained performance.
Within these challenges, we realize that not every approach we choose provides benefits. Some might even make the systems worse. Consequently, we want to give reasons for making some decisions in the following chapter.

4.2. Customizations of the Decompositional Approach

In this section, we will summarize all changes compared to the system proposed by Karamcheti et al. in [1] to adopt this work to the domain of spreadsheet operations.

4.2.1. Entity Abstraction

In the work of Karamcheti et al., the lifted utterances are retrieved using a lexical search on the input sequences. For this, they defined a dictionary of lifted objects from the context by removing secondary word stems. When an input utterance is given, all compounds are lifted and then checked against the context dictionary. If a match is found, the word gets lifted and is replaced with an object marker.

Here our system is fundamentally different. We will provide a dependency-based entity abstractor converting dependency parse trees into annotated sentences. This system will not only be able to lift single objects from the parser's context but also conditions and object lists. Moreover, it will also be able to differentiate between different types of objects with regards to our DSL like columns, tables, or values. We will explain the system we use in more detail in Section 4.4.1.

4.2.2. Utterance Embedding

For the lifted utterance embeddings, the paper by Karamcheti et al. uses summed GloVe word vectors (see Section 2.3.8) of the input sequence.

We are using BERT and ELECTRA embeddings to encode our lifted input utterances. Additionally, we will define multiple postprocessing methods for our word vectors different from the original implementation. We will also define another adapted output from our BERT word vectors using positional encodings and padding. For more details on the utterance embeddings please refer to Section 4.5.1.

Beyond the changes to the word vectors themselves, we give a method for clustering our utterance embeddings using metric learning that is not part of the original system. Here we transform our vectors into a better-separated space and define a distance-sensitive similarity measure (see Section 4.5.3). Karamcheti et al., in contrast to us, used cosine similarity and no clustering method on their embedded vectors.

4.2.3. Combination

Instead of culling illegitimate candidates in the reranking stage, we already filter them at the combination stage by comparing the input types with the signatures of our candidate programs, removing all candidates that cannot accept an input type lifted.

4.2.4. Reranker Inputs

For reranker inputs, Karamcheti et al. use a bag-of-words representation from the list of grounded objects in the parser's context and summed GloVe word embeddings for both the original utterance and each candidate program. This results in an input for each program consisting of the bag-of-words representation of the state and the summed word embeddings of both the utterance and candidate program.

Our system uses the output probability of the candidate resolver, a bag-of-words representation of the lambda calculus output of another parser, and an aligned utterance table embedding as inputs for the reranker (see Section 4.7.1 and Section 4.7.3).

With the aligned utterance table embeddings, we seek to capture the table context in relation to the utterance as we feel a bag-of-words vector might not be sufficient to capture the meaning of a table properly. The method is based on the work of Chen et al. [5] creating a single embedding from a custom document containing the table column names, the input utterance, and further context information. The resulting output is then pooled. For more details on how we create this embedding, we refer to Section 4.7.2.

4.3. Domain Specific Language

We created a pseudo-domain-specific language containing a minimal set of operations required to reflect specific scenarios provided to us from the chair for parallel and distributed systems. We are using the denomination "pseudo" because the language has no implementation. Our target is to provide a formalized human-readable output of minimal complexity, which can be used to verify that the parser is working and is translating utterances as intended. To achieve this, we have made some decisions and assumptions we want to list below:

- All operations defined in the DSL either create a view or change the table data in place. A view refers to the underlying data constrained by all operations creating it. Thus, the view changes when the underlying table is manipulated. If the operation does not manipulate the data or a view, the main result is a side-effect.
- There will be 5 data types: `table`, `column`, `condition`, `value` and `group`. Depending on the function, the `table` data type can be a view or an actual table. An instance of `group` is a given access group.
- We assume that the program which uses our DSL has a GUI. We will, therefore, not provide functions for operations like naming input fields in web forms or renaming columns, as we feel GUIs better handle these operations.
- Nested function applications do not exist. All functions are called linearly and are divided by semicolons. A set of DSL functions define modifications on the same view, meaning that if a set of functions divided by semicolons is given, they all affect the same view/table in the order written in the program. This is done to keep the programs simple and easy to read.
- We will not include any implementation details like the type of algorithm used. We do this to keep the set of operations minimal.

4.3.1. Syntax

The DSL has a simplistic syntax consisting of functions applied to tables. Beyond function names, there are no keywords of any kind. A program is a set of predefined function calls separated by semicolons in our language.

Every function call consists of a function name and its inputs inside parentheses. One input must be the table on which the operation is done, creating a view or editing the table itself. If multiple inputs of the same type are required, an input list in brackets is given to the function.

```
FILTER(name == 'Sophie Boehm', ['time_slots']);
COUNT(['time_slots'])
```

Figure 4.1.: A DSL program filtering the table `time_slots` such that all instances where the `name` field is equal to "Sophie Boehm" are inside and then count the rows of the resulting view.

One can define functions by giving a unique function name and all input types required in brackets. Input lists are marked using a comma after the first bracket. If marked, single or multiple inputs of the same type can be given to the function as a list. This does not imply that multiple inputs are required. Furthermore, the syntax does not allow for fixed-length input lists. While we realize that an actual implementation requires such constructions, we feel that incorporating syntax enabling these constraints is relatively simple.

A special class of input is the condition. We define a condition as a relation between a column and some value. Multiple relations can be combined using the logical and, and the logical or. In addition, there is a special condition construct `[column] ascending/descending`, which defines a sorting direction of a table. We define such constructs as conditions due to the implementation of our entity abstraction system rather than for syntactical or usability reasons (see Section 4.4).

The following subsections will motivate a minimal set of operations for our DSL from the application examples. We do not claim that this list is complete, nor do we imply that another set of actions cannot be valid. Since we only want to have

```
SELECT([table], [,column])
JOIN([,table], [column])
SORT([[table], [condition])
```

Figure 4.2.: Definition of some functions in the DSL. One can see the comma denominating a parameter list of the same type. For the `JOIN` statement, there is no constraint for the parameter list length, even though it is desirable to limit the number of tables to exactly two.

a reasonable representation that can be used to measure the system's performance we keep the set of actions as small as possible to reduce complexity and increase clarity.

4.3.2. Calendar Slots

The setting is to provide a system for managing meetings. A table should be bound to a web form where users can fill in some information about them and select their desired meeting date and time. The table has two views: The web form, which can only access rows that have not yet been filled, and the creator of the table, who can see all table contents. For the DSL to capture this setting, it needs to provide the following atomic operations:

1. `CREATE([table], [,column])`, to create an empty table consisting of the desired columns.

2. `FILTER([table], [condition])`, filter table rows according to a condition.

3. `SORT([table], [condition])`, to sort a table according to a column and direction. The conditions used here are the special conditions mentioned in Section 4.3.1.

4. `SELECT([table], [,column])`, selects desired columns from a table.

5. `COUNT([table])`, counts the number of rows in the table view.

6. `BIND FORM([table])`, to bind a web view to the desired table.

4.3.3. Holiday Home Guests Management

Here the task is to create a cleaning schedule from a list of guest data for a holiday home. A web form should be created for contractors who can choose which rooms to clean based on the bookings. The view for the contractors should not include any sensitive data and is implemented by a second table which can be joined with the booking table for an overview. Furthermore, rents should be computed from the length of stay, nightly rent, and the possible cleaning fee.
To achieve this, we provide the following new atomic actions:

1. `SUM([table], [,column])`, component-wise summation of the columns in the input list.

2. `SUBTRACT([table], [,column])`, component-wise subtraction where the minuend is the first entry in the input list and all others are subtrahends.

3. `MULTIPLY([table], [,column])`, component-wise multiplication of the columns in the input list.

4. `DIVIDE([table], [,column])`, component-wise division of the input columns where the operation is applied in the order provided by the input.

5. `ASSIGN([table], [column])`, assigns the vector-valued side effect of the function above to the given column.

6. `ADD([table], [,column])`, add columns to a table.

7. `DELETE([table], [,column])`, delete columns from a table.

8. `JOIN([,table], [column])`, to join tables along a column.

We realize that mathematical operations should be constrained further, but we feel that creating a DSL covering all side effects is beyond the scope of this work, where the task is to create a semantic parser.

4.3.4. Faculty Database

This task involves managing a small database consisting of one table of employees for the faculty of math and computer science at Heidelberg University. Here the task is to manage different access groups, each with a bespoke view of the underlying database. Furthermore, faculty committee lists should be created, which may be accessed by the institute members.
This requires the following additional operation:

1. `BIND GROUP([table], [group])`, bind a table view to some group of users.

4.3.5. Medical Certificate Upload

This task involves a database table for medical certificates uploaded to some server. Students are entering their relevant information inside a web form bound to some predefined table. The certificates are only valid for a limited time. Thus out-of-date certificates should be deleted automatically.
These requirements motivate the following list of atomic operations:

1. `DELETE ROW([table], [condition])`, deletes rows according to a condition.
2. `LISTEN([table])`, listens to changes in the table and executes the program above it.

4.4. Entity Abstraction

As the first step in parsing a natural language utterance, the Karamcheti paper [1] uses a dictionary search to extract all objects related to the state around its house helper bot. While this works for simple input utterances, this type of entity abstraction quickly becomes ill-suited for more complex examples. Firstly, a dictionary search requires one to create an extensive list of stop words for pruning unnecessary information. But more importantly, more syntactically complex constructions like conditions cannot be lifted correctly.

Since sentences are very diverse, we would recommend a machine learning method such as a transformer to obtain lifted instances. This model would be trained on pairs of utterances and their lifted counterparts to create a model computing the probability of a word being lifted in a sentence. But since we did not find any data consistent with our task, we designed a rule-based method using dependency grammars (see Section 2.2.2).

We chose dependency grammars because they capture semantically relevant information on the relation of words in a sentence rather than syntactical relationships. While syntax can inform semantics, a purely syntactical approach would only work well on grammatical sentences. We feel that in a realistic conversation, utterances are frequently ungrammatical, and a system purely based on syntax would perform poorly in this environment.

4.4.1. Using Dependency Parses for Entity Abstraction

Our entity abstraction mechanism uses the Stanford CoreNLP parser[1] [27] to parse a given utterance into a universal dependencies parse [28]. We then transform the dependency tree into a tree consisting of nodes that relate to certain dependencies we want to capture. For this, we group the universal dependencies into 6 special node categories: sentence roots, objects, values, cases, compounds, and stopwords. The rest is assigned a generic parent type of these categories.

Sentence Roots

Sentence roots are the root of the dependency parse tree of a sentence and are the verb defining our action. They have the `ROOT` dependency. The words assigned to root nodes are always contained within the lifted instance and are the distinguishing word of a lifted utterance.

If a child of a root node has the conjunction dependency `conj`, it is the root of a main clause subsentence of the given utterance. We then split the trees of the original root and the subroot such that we can process the clauses separately.

[1]CoreNLP source code available at: `https://github.com/stanfordnlp/CoreNLP`

The approach is analog to the method of Karamcheti et al. [1] which separates utterances at the word "and". Notably, our process handles conjunctions more gracefully as it can differentiate between conjunctions of sentences and other conjunctions that one might not want to separate, like object conjunctions.

Compounds

Compound nodes are the secondary stems of compound words and have the universal dependency `compound`. Semantically they inherit their parent's role as they are the same entity. For example, the word "time" of the compound "time slots" is a secondary word stem and is assigned this node category. In our program, we remove these words from the lifted instance and prepend or append them to the respective parent node when we want to extract the inputs for our candidate programs.

Sentence Objects

Sentence objects can be columns, tables, or access groups for our DSL. They have universal dependencies `obj`, `appos`, `nmod` or the oblique dependency (`obl`) if a child node has the `case` dependency. If a child node of a sentence object node has the `conj` dependency, it too is an object node.

We are looking at two cases to resolve the DSL type of an object node. The first and crucially more trivial case is to use the context, e.g., the access group names, the names of the tables, and their respective column names accessible by the program.

In this case, we retrieve the object compounds by finding all compound node children and combining the words. Then for column names, we normalize the compound words by putting all words into lowercase and finding the singular form. We retrieve the singular form by using the `singularize` function from the python package `pattern`[1]. Then we combine the compounds such that the lower case compound, the capitalized compound, and the programming naming conventions

[1]`pattern` source code available at: `https://github.com/clips/pattern`

snake case and camel case may be matched. For the table name search, we do the same but do not singularize the words.

If a respective name is found, we insert the tag from the DSL (e.g., `[column]` if it is a column) at its place in the sentence. For conjunctions, we check all children of the object node and insert a list tag if multiple objects of the same type are found in the conjunction.

Conversely, when no context is given, the abstractor has to find the objects from the sentence itself, which is more complex. Here the user has to name the object types in the utterance. For example, in "Create the table time slots, with columns name, surname, date, and slot." the parser should realize that "time slots" is the name of the table and the object list given are the columns.

Indeed the system is capable of doing this by looking at the compositions "columns name" and "table time slots". If children of object nodes match one of the object types, the respective type is given to the object or object list. This yields, for our example, that "columns" and "table" are matched, and the respective type tag is used.

Values

Value nodes capture some value described in a sentence, such as a number or a date. They have the universal dependencies `nummod`, `amod` or `acl:relcl` and are one half of a condition with the other half being an object node.

Sentence Cases

These nodes either represent the head of a conditional construction or are the relational operator of a condition. Case nodes either have the dependency `case` if they are a child of an oblique dependency, or they have the `aux` dependency and are a child of a value node.

For a sentence case node to be a DSL condition, there has to be an object and a value node in the vicinity of the case node. This makes sense, as a condition is some relation of an object to a value. We define the vicinity as the children, neighbors, and the parent node. If the case construction passes this check, the `[condition]` tag will be inserted. Otherwise, the bound words of the case construction will be lifted as if there is no case found.

We also create a lifted instance for the case itself to parse conditions using the entity resolver. For this, the abstractor lifts the values and columns in the vicinity and returns them separately as a specific condition input.

Stopwords

Stopword nodes capture unnecessary information for our entity abstraction task. The words bound to them are never displayed in a lifted utterance. They have one of the following universal dependencies: `det`, `nsubj`, `punct`, `cc` or `acl`. Furthermore, they can also have the auxiliary dependency `aux` if their parent is not a value dependency.

4.4.2. Shortcomings

While we cover some of the underlying sentence structures from our data set, we want to clarify that the categories defined in Section 4.4.1 do not nearly cover all possible sentences which can arise in a spreadsheet context. To achieve reasonable correctness, further linguistic work finding rules and dependencies corresponding to one of the categories above is required. Moreover, the underlying dependency parser has to be improved to reduce the likelihood of erroneous input.

We also realize that the table-based input type deduction only works if column, table, and group names are unambiguous. Apart from this, we note that there is no implementation for user group resolving as of this date.
Because of these shortcomings, an abstraction implementation that uses our data set such that a perfect abstraction mechanism can be mocked is provided. This is done to measure the performance of the machine learning pipeline parts independently from the abstraction mechanism.

4.5. Program Resolving

Finding candidate programs for a given lifted output is the first machine learning subtask we will solve. We will divide this task into three stages. In the first stage, a lifted utterance is embedded. These embeddings are then pooled and transformed into a search space according to a learned metric as seen in Section 2.3.5. Then a nearest neighbor search on lifted utterance program pairs is done in this transformed search space. Finally, for each assigned distance, a probability is computed of how likely it is that the distance belongs to two instances with the same lifted DSL program. Both the probabilities and the respective candidates from the closest instances are returned.

4.5.1. Embedding

For our text embeddings we use pre-trained BERT and ELECTRA models described in Section 2.3.8. We use the small, base, and large versions to compare the performance. Depending on the configuration, the sequence embedding or the embedding of the special `[CLS]` token at the beginning of an utterance is returned. It has been shown by Devlin et al. that the embedding of this tag captures the semantic meaning of the following document well [9].

In addition, we provide the option to use positional encodings as proposed by Vaswani et al. [38] for our sequence output (see Section 2.3.7). For this method, we also cut the fixed length sequence output of BERT at the length of our input sequence and fill the removed part with zero-valued vectors. This way, we hope to better capture the length of our input utterance.

We then provide multiple postprocessing methods for our resulting BERT outputs. The simplest method involves summing the word vectors and squaring the result. Moreover, we provide maximum and average pooling of our outputs.

Pooling is a dimension reduction method primarily used in convolutional neural networks. A window is moved over the entries of a tensor such that all scalars inside it are covered. Inside this pooling window, a new scalar is computed at each step. We use two rules to retrieve the said value. In max pooling, the biggest scalar is taken and used as the result. In average pooling, the result is computed from the average of all values inside the pooling window. Pooling windows usually reduce overfitting and improve downstream models' inference and training times while not drastically affecting the performance. We want to look at possible performance differences by using pooling in our system.

4.5.2. Nearest Neighbor Search

The nearest neighbor search enables us to add new instances for one-shot generalization. In addition, it provides us with a confidence measure for selecting instances for decomposition. We provide 2 implementations of the nearest neighbor algorithm from Section 2.3.1. If the utterance text embedding rank is lower or equal to two, we use a kd-tree to find the nearest neighbor of a given input vector. Otherwise, we implement a naive approach to compute the euclidean distance between all trained instances and the input vector.
We then filter out all results outside of a manually defined threshold for our search. If no instance is found within the distance threshold, we conclude that the system is unsure, and the user can add a new training instance by providing natural language steps corresponding to known subactions.

KD-Trees

A kd-tree is a binary search tree for finding close points to a query vector in k dimensions. The idea is to use a heuristic to split the search space into smaller sections using a set of known vectors. For each dimension, the heuristic is applied to find the point for the best split. The selected point is then used as a node in the kd-tree. The algorithm runs as long as there are no points in the set of given vectors. There is only one node per vector, but a single node can split multiple dimensions if there are more dimensions than examples for the kd-tree. A suitable heuristic could be the distance to the center of a ball surrounding the given vectors for each dimension.

When we provide a query vector, the algorithm walks through the kd-tree. At every step, the query vector is compared with a node in the dimensions where the node splits the search space. If the value of the query vector is smaller than the respective value of the node, we take the child to the left; otherwise, we take the child to the right. This is done until a leaf is reached. Afterward, the algorithm propagates up through the tree and passes all nodes within a set threshold, computing the distance to them.

4.5.3. Feature Clustering

Karamcheti et al. proposed that after finding the nearest neighbors, cosine similarity between the input and all trained neighbors should be used as an input for the regression model [1] which is defined as follows:

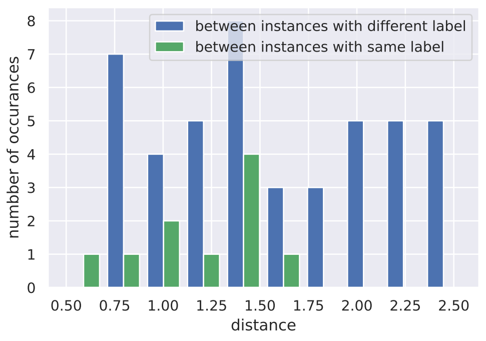
$$\text{similarity}(A, B) = \frac{A \cdot B}{\|A\|_2 \|B\|_2}, \tag{4.1}$$

where $\| \cdot \|_2$ is the euclidean distance.

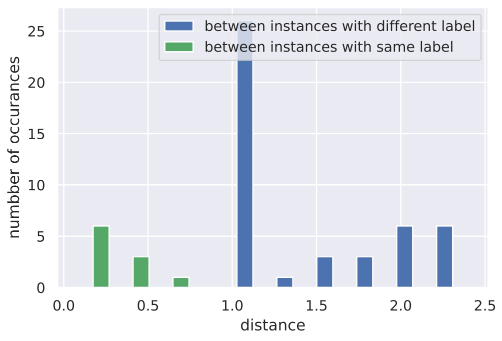
It turns out that the embeddings of our lifted instances are very similar. To achieve better performance for the nearest neighbor algorithm and the regression model, some domain-specific training needs to be done. Instead of fine-tuning BERT and ELECTRA, we chose a metric learning approach as it is data efficient and easier to implement thanks to the `metric-learn`[1] package in `sklearn` [51]. In particular, we use NCA and LMNN described in Section 2.3.5 in an attempt to transform our instances into a well-separated search space.

Metric learning scales distances between data points in a vector space. This means that this method does not change cosine similarities while the euclidean distance is changed. This is because cosine similarity is a normalized measure and does not change with increasing or decreasing distances. Therefore, we use the distances directly for our regressor.

[1]`metric-learn` source code available at: `https://github.com/scikit-learn-contrib/metric-learn`



(a) original



(b) using LMNN

Figure 4.3.: Two histograms taken from a subset of examples from the calendar slots domain. Here we can see the effect of metric learning on the separation of our instances.

4.5.4. Regression Model

The regressor is a single neuron perceptron with sigmoid activation and binary cross-entropy loss shown in Section 2.3.3 and is equivalent to a logistic regression. While we initialize the trainable weight randomly, the initial value of the bias is set to 5 such that the sigmoid function can assign all values between zero and one for the strictly positive distances from the start of training.
In training, we use a modification of gradient descent called adaptive moment estimation (ADAM) [52]. It decreases the sensitivity of gradient descent by using a running average of both the last gradients and first and second moments of the loss function over the last training steps.

4.5.5. Handling Conditions

To parse conditions, we use another program resolver the same way as when we would parse programs with the difference that this model is trained on lifted conditions and does not have a threshold bounding the nearest neighbor search. From the condition candidates found, we then take the most probable instance.

4.6. Combination

In the semantic parser pipeline combination step, we ground the lifted candidate programs given by the candidate resolver with the lifted inputs of the abstractor.

We take a candidate program and extract an input signature using the regular expression (regex) `\[(,)?(.+?)(\d)?]`, which matches the input tags in the function definitions of the DSL. The first capture group of the regex matches with the comma marker, signaling an input list. The next group matches the actual type string, while the last capture group finds a digit if the given program has multiple possible inputs of the same type which need to be enumerated.

Then the candidate input signatures are compared with the lifted inputs filtering candidates whose input types do not agree. The remaining programs are filled with the lifted inputs and returned. At this point, we remark that partial grounding of a candidate is possible. This is done to enable the addition of macros where some part of the underlying program has constant inputs.

4.7. Candidate Reranking

In this final step, we take the grounded candidates, their output probabilities, the tables involved, and the original utterance to embed them jointly and select the most probable output.

4.7.1. Informing the Output With Another Parser

The first half of the embedding process uses another semantic parser to parse the original utterance into lambda calculus (see Section 2.1.2). The parser we use is a CCG-parser by Yoshikawa et al.[1] [18].

We embed the lambda calculus outputs as bag-of-words vectors. For this, we parse all utterances to lambda calculus in the training set and create a vocabulary

[1]The source code of the parser can be found here: `https://github.com/masashi-y/depccg`

consisting of the normalized abstractions from these parses. These abstractions are verbs or conditions in the input sentence - basically, any function defined by a word. We call them normalized because they only contain the word stem and are not conjugated or otherwise changed. In doing this, we cover all possible representations of the abstractions provided. The normalized abstractions are then put into a dictionary with DSL programs as keys, and the respective abstractions from all lambda calculus parses in the training set as values. Using this dictionary, we can create bag-of-words vectors that have the same length as the number of possible lifted programs.

4.7.2. Table Embedding

The second half of the embedding process takes the original utterance, and all tables involved and creates an aligned embedding of both. We call it aligned because the utterance and the table contents are part of the same document to be embedded.

To create this embedding, we largely follow the work of Chen et al. [5] who have created aligned embeddings for a table search task using BERT. They aligned questions with column names and table contents of each known table by modifying the tokenization used for BERT. A very similar approach is also used by Yin et al. [6] to create table embeddings. The approach works as follows:

1. We start with the special document start token `[CLS]`.
2. We insert the natural language utterance and put the separator token `[SEP]` at the end.
3. The column names separated by the `[SEP]` token get appended.
4. We append a list of `[SEP]` separated context information. In our case, this is only a short task description like `meeting management`.
5. Optionally, we can choose the semantically closest rows from the table and append them. We do this the following way:

a) For each row, we create a sequence of strings containing the column name and respective value. These strings we call linearized entries akin to the definition by Yin et al. [6].

b) We concatenate all linearized entries and embed them using another text embedder like GloVe.

c) We find the semantically closest embeddings to our input query.

d) We append the list of rows as a list of linearized entries separated by the `[SEP]` token until we have reached the maximum sequence length of BERT, which usually is 128.

6. We embed the resulting document using a BERT language model.

As an example, consider the `time_slots` (see Table C.4) and the natural language query "Select slot, surname and private notes." taken from our dataset. Applying points 1 and 2 from above to our input, we get:

```
[CLS] Select slot, surname and private notes. [SEP]
```

The table contains the columns: `Date`, `Slot`, `Name`, `Surname`, `Email`, `Phone` and `Private Notes`. Thus applying 3 yields:

```
[CLS] Select slot, surname and private notes. [SEP]
Date [SEP] Name [SEP] Surname [SEP] Email [SEP]
Phone [SEP] Private Notes [SEP]
```

Finally, we add the context information in our case, this would be the short description `meeting management` of our task, yielding:

```
[CLS] Select slot, surname and private notes. [SEP]
Date [SEP] Name [SEP] Surname [SEP] Email [SEP]
Phone [SEP] Private Notes [SEP]
meeting management [SEP]
```

From this, we compute a sequence embedding using a BERT-type language model. The result then gets pooled. We will test two pooling methods, maximum and average pooling. The idea is that the pooled vector only contains relevant data, for example, very expressive data when applying max pooling. If multiple tables are involved, we sum the aligned embeddings.

We opt out of doing step number 4 as we want the model to be independent of the known tables in its training dataset and provide an ad-hoc way of embedding the table columns. Furthermore, we only use pre-trained language models in our reranker. Yin et al. [6] have shown promising performance pre-training their embedders, but unfortunately, the limited nature of our data does not allow for this. Therefore our research focuses more on the merit of this ad-hoc method based on the method of Chen et al. [5] as part of an interpretation of the reranker proposed by Karamcheti et al. in [1].

4.7.3. Reranker Model

The heart of the reranker is a two-layer feedforward neural network with a single output neuron describing the probability that our utterance combined with the output probability of the resolver model agrees with the table embedding. Analog to the regression model in the resolver described in Section 4.5.4 we use the binary cross-entropy loss and adaptive moment estimation (ADAM) to compute the optimal weights of the network.

The input to this network is the output probability of the reranker model, the table embedding, and the bag-of-words vector of the lambda calculus from the CCG parser if so configured. For training, the reranker model is trained jointly with the resolver model. Here the output probabilities of the resolver model and the correct probability for the example from the training set are given.

4.8. Retraining

To train new instances from decomposition, we need to provide a way for the resolver and reranker to retrain using the new instance generated from user interaction. A retraining sequence starts when the semantic parser pipeline returns the `NOT_SURE` token, and the user provides a list of atomic actions in natural language.

As an example of how the system creates a new training instance, let us look at the combined utterance "Create meeting web form from time slots." from the dataset and its respective decomposition: "Filter time slots where name is empty.", "Create web form from time slots.".

At first, the combined utterance is given to the system, and the normal prediction process is started. That means the utterance gets lifted by the entity abstractor and put into the candidate resolver. Assuming our abstractor and resolver are flawless, we would get "Create meeting web form [table]" as the lifted utterance, and the resolver would not find a neighbor inside its threshold. Therefore, the pipeline would return the `NOT_SURE` token. Now the user is prompted to provide the decomposition above.

Given that the user provides the list of utterances above, our problem setting is to provide a correct composite program $\mathcal{P}$ from the original query $\mathcal{O}$ and the list of atomic utterances $\mathcal{A}$. As one might realize, the program implied by $\mathcal{O}$ is as much a compositional function that can take different tables as inputs as it is a macro requiring the input table to have the column `name` in it. We can see this from the absence of the condition in the combined utterance implying that the condition input for filtering is constant. The combiner acts in the same way by comparing the lifted input of the original instance with the inputs of all atomic utterances. This way, the pipeline can return the two partially grounded subprograms, which are stitched together to become the new lifted DSL program to be learned by the system.

This means, in our example, that the output of our pipeline should be:

```
FILTER([table], name == '');
BIND FORM([table])
```

The tuple of $\mathcal{O}$ and the partially grounded DSL program are the inputs for our retraining process.

4.8.1. Candidate Resolver

The lifted original instance and the partially grounded program are passed to the candidate resolver. The lifted utterance is embedded and added to the nearest neighbor search space. If a metric learning algorithm is used, it is retrained with the new example, and the whole search space is retransformed by the newly learned metric.

The feature vectors from this retransformed space are then used to retrain the regressor model yielding a new weight and bias.

4.8.2. Candidate Reranker

$\mathcal{O}$ and the tables corresponding to $\mathcal{O}$ are given to the table embedder. If the reranker uses lambda calculus embeddings as well, the dictionary gets extended with the new grounded program. Therefore, the input dimensionality for the regression network has changed as the bag-of-words vectors from the lambda calculus embedding now have an additional entry. The reranker network has to be adapted for this new input size and is trained on the enhanced dataset.

4.9. Model Data

As part of this work, we release a hand-crafted tiered dataset containing queries from our use cases outlined in Section 4.3. It contains 144 atomic actions in total, of which we hold out 71 for testing. The dataset has three degrees of difficulty:

Difficulty 1 contains queries for the following DSL operations: `SELECT`, `COUNT`, `BIND FORM`, `SUM`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `DELETE`, `JOIN`, `BIND GROUP` and `LISTEN`. These are the easiest queries to solve as they do not contain any condition. Here the entity abstractor has access to all tables and columns involved to lift the utterances. These make up the bulk of our dataset as 44 training, and 42 test instances are of this difficulty degree.

Difficulty 2 contains all queries of difficulty 1 and queries for all operations where the entity abstraction mechanism has to lift the utterances from the sentence itself rather than the context provided. These involve utterances describing the atomic actions `CREATE` and `ADD`. They make up the smallest tier consisting of only 10 instances in each training and test set.

Difficulty 3 contains all queries of difficulties 1 and 2 and the remaining queries. This difficulty is reserved for any utterance with a condition that needs to be parsed. These are the most complex utterances we provide as conditions are hard to lift as well as harder to parse because of their complex structure. The utterances describe `FILTER` and `DELETE ROW` operations and make up the remaining 20 instances in the training and test sets each.

While small, we feel this dataset is representative of measuring the performance of the machine learning algorithms as lifting utterances reduces the diversity of any dataset massively down to a few synonyms of the verbs used to describe our atomic actions. We also introduced multiple unseen synonyms into our test data to see how well the system generalizes to unseen but known instances. Furthermore, the instances provide enough grammatical variety to expose the limitations of our rule-based entity abstraction mechanism.
Moreover, we provide another 8 one-shot generalization examples to test the pipeline's ability to recognize composite queries and generalize from decomposing actions.

The tables in appendix C are the parser's context for each of the tasks given to us.

5. Evaluation

Using the implemented system described in Chapter 4 we want to define the following research questions to provide a structured and thorough evaluation of our system and answer whether our approach is a viable method of creating a data-efficient semantic parser for spreadsheet applications:

RQ1 How well can the semantic parser translate natural language utterances into atomic DSL operations on a limited dataset with increasing difficulty? Are there limiting factors bounding the performance of the parser?

RQ2 How much does each part of the semantic parser pipeline contribute to the overall accuracy? What is the best configuration of the machine learning subsystems, and why?

RQ3 How much does the rule-based entity abstraction method negatively impact the accuracy of the parser, and why?

RQ4 Is the parser realistically able to learn from decomposition? Is continual learning from one-shot generalization possible using the method outlined?

The answers to these questions provided here will form the basis of our conclusion and discussion of this work. Furthermore, we hope to find new avenues of research along these questions to provide a basis for additional work in this field.

At first, we will do a quantitative analysis of the system on our dataset, which will inform examples for a qualitative analysis where we will illustrate some of the reasons why the semantic parser performs the way it does.

5.1. Quantitative Evaluation

5.1.1. Configurables

To answer the research questions, we have devised a set of measurements that measure the accuracy of our system on the three different tiers of the provided dataset. To this end, we will define some configurables for the different components of the semantic parser pipeline here.

Abstractor Configurations

For the abstraction mechanism, we provide two configurations: the dependency-based mechanism described in Section 4.4.1 and a mocked entity abstractor providing flawless abstractions using our dataset.

This enables us to precisely quantify the impact of the rule-based entity abstractor for **RQ3**. Additionally, the mocked abstraction mechanism is used to measure the performance of our system for **RQ1** and **RQ2** independently from the abstraction mechanism, as we have reason to believe that abstraction is a major bottleneck in the overall performance of the parser.

Candidate Resolver Configurations

To answer **RQ2** we provide multiple configurations for our candidate resolver, which we will compare in our measurements. We can configure the text embedding language model, embedded vector postprocessing, and the type of metric learning used for our search space.

For the lifted utterance embedding, we can use the following BERT-based language models:

- $\text{BERT}_{\text{SMALL}}$: $L = 12,\ H = 512,\ A = 8$
- $\text{BERT}_{\text{BASE}}$: $L = 12,\ H = 768,\ A = 12$
- $\text{BERT}_{\text{LARGE}}$: $L = 24,\ H = 1024,\ A = 16$

- ELECTRA$_{\text{SMALL}}$: $L = 12,\ H = 256,\ A = 4$
- ELECTRA$_{\text{BASE}}$: $L = 12,\ H = 768,\ A = 12$
- ELECTRA$_{\text{LARGE}}$: $L = 24,\ H = 1024,\ A = 16$

As a reminder from Section 2.3.8 L is the number of encoder cells, H the size of the hidden vectors and A the number of attention heads.

As word vector postprocessing options, we provide the following:

- The raw sequence output.
- The pooled output as described by Devlin et al. in [9] consisting of only the word vector of the document start token `[CLS]`. We call this BERT standard pooling.
- The sequence output cut to the input sequence length with the positional encodings of Vaswani et al. [38]. We implement the sequence cut, setting every word vector beyond the original sequence length to zero.
- The max pooled raw sequence output.
- The average pooled raw sequence output.
- Max pooling applied to the sequence output with positional encoding.
- Average pooling applied to the sequence output with positional encoding.
- The cubed summed raw sequence embeddings. This is the closest method to the original by Karamcheti et al., who suggested summed embeddings [1].
- The cubed summed BERT standard pooling vectors.
- The cubed summed sequence word vectors with positional encoding.

Finally, we use the following metric learning configurations for the resolver nearest neighbor search:

- A large margin nearest neighbor (LMNN) trained metric with a learning rate of 10^{-3}.
- A metric learned using the neighborhood component analysis (NCA) task.
- No metric learning, meaning standard euclidean distance.

Candidate Reranker Configurations

For the candidate reranker, we also provide multiple configurations as a means to answer **RQ2**. Analog to the resolver, we can change the table embedding language models to be any BERT-like model, as shown above. Furthermore, we provide average pooling and max pooling options for the table embeddings. It is also possible to use the bag-of-words lambda calculus embeddings from Section 4.7.1 or disable them.

Semantic Parser Pipeline Configurations

Beyond the configurations, for the pipeline parts shown above, we can also configure the larger structure of the semantic parser pipeline. Here we can create a pipeline with or without the reranker. Without a reranker, the grounded program corresponding to the highest resolver probability is selected by default.

5.1.2. Experimental Setup and Execution

In this work, we do 8 measurements. The first 7 tests are done on the atomic actions and provide the average accuracy and the accuracy variance of our system along a configurable axes of our system for all difficulty tiers in our dataset. For each test, we will fix the remaining configurables to restrict the number of tests as we cannot realistically provide data on all 17280 possible configurations. The last test will check how many of the 8 composites in the dataset can be recognized and generalized by the best system acquired through the atomic action tests.

Defaults

In every atomic action measurement, we compute the best, worst and average accuracy as well as the variance thereof by repeating the test for each configuration 20 times. The notable exception to this is the reranker lambda embedding measurement, where we only do 5 repetitions due to the longer inference times of these configurations.

Tn Table 5.1 we list the default hyperparameter values for our testing. We leave the subject of hyperparameter tuning in our system as a possible alley for future work, though we will change the not-sure threshold of our best classifier in the one-shot generalization and abstraction tests.

Hyperparameter	Default Value
Program Resolver Learning Rate	10^{-3}
Program Resolver Number of Epochs	1
Program Resolver Not Sure Threshold	Largest distance between two embedded utterances in the training set with the same program template / 2
Condition Resolver Learning Rate	10^{-3}
Condition Resolver Number of Epochs	1
Reranker Network Hidden Size	table embedding size / 2
Reranker Network Hidden Layer Dropout	0.4
Reranker Network Output Layer Dropout	0.4
Reranker Number of Epochs	2
Reranker Learning Rate	10^{-3}

Table 5.1.: The default hyperparameter values for the measurements.

Resolver Utterance Embedding Measurement

Here we measure the performance of the different utterance embedding models listed in Section 5.1.1. This test is done without the reranker to not introduce any errors which might occur by the interaction between both systems. You can see a table containing all configurations in Table 5.2.

Configurable	Possible Values
Abstraction Method	mocked abstraction
Resolver Text Embedding Language Model	BERT$_{\text{SMALL}}$, BERT$_{\text{BASE}}$, BERT$_{\text{LARGE}}$, ELECTRA$_{\text{SMALL}}$, ELECTRA$_{\text{BASE}}$, ELECTRA$_{\text{LARGE}}$
Resolver Text Embedding Postprocessing	max pooling
Resolver Metric Learning	none
Reranker included	no

Table 5.2.: The pipeline configurations for the resolver utterance embedding postprocessing measurement.

Resolver Metric Learning Measurement

Here, we measure the impact of metric learning on our overall parsing accuracy. The configurations tested can be found in Table 5.3.

Configurable	Possible Values
Abstraction Method	mocked abstraction
Resolver Text Embedding Language Model	BERT$_{\text{SMALL}}$
Resolver Text Embedding Postprocessing	max pooling
Resolver Metric Learning	NCA, LMNN, none
Reranker included	no

Table 5.3.: The pipeline configurations for the resolver metric learning measurement.

Resolver Embedding Postprocessing Measurement

This measurement studies the impact of different approaches to postprocessing utterance embeddings. For this test, we use the following configurations shown in Table 5.4.

Configurable	Possible Values
Abstraction Method	mocked abstraction
Resolver Text Embedding Language Model	$\text{BERT}_{\text{SMALL}}$
Resolver Text Embedding Postprocessing	sequence output, sequence output with positional encoding, BERT pooling, max pooling, average pooling, positional encoding with max pooling, positional encoding with average pooling, summed and squared sequence output, summed and squared positionally encoded sequence output, summed and squared BERT pooled output
Resolver Metric Learning	none
Reranker included	no

Table 5.4.: The pipeline configurations for the resolver utterance embedding post-process measurement.

Reranker Table Embedding Measurement

In this measurement, we compute the best, worst, average accuracy, and variance thereof to investigate the performance impact of the table embedding language models on the semantic parser pipeline. The configurations can be found in Table 5.5.

Reranker Table Embedding Pooling Measurement

Here we measure the performance of the semantic parser using max and avg pooled table embeddings at the reranker stage. See Table 5.6 for the configurations.

Reranker Lambda Embedding Measurement

Here, we look at how well the reranker works with the additional information provided by the CCG-Parser of Yoshikawa et al. [18]. For this, we look at the configuration found in Table 5.7.

Configurable	Possible Values
Abstraction Method	mocked abstraction
Resolver Text Embedding Language Model	$\text{BERT}_{\text{SMALL}}$
Resolver Text Embedding Postprocessing	max pooling
Resolver Metric Learning	none
Table Embedding Language Model	$\text{BERT}_{\text{SMALL}}$, $\text{BERT}_{\text{BASE}}$, $\text{BERT}_{\text{LARGE}}$, $\text{ELECTRA}_{\text{SMALL}}$, $\text{ELECTRA}_{\text{BASE}}$, $\text{ELECTRA}_{\text{LARGE}}$
Table Embedding Pooling Method	max pooling
Lambda Calculus Embeddings included	no

Table 5.5.: The pipeline configurations for the reranker table embedding measurement.

Configurable	Possible Values
Abstraction Method	mocked abstraction
Resolver Text Embedding Language Model	$\text{BERT}_{\text{SMALL}}$
Resolver Text Embedding Postprocessing	max pooling
Resolver Metric Learning	none
Table Embedding Language Model	$\text{BERT}_{\text{SMALL}}$
Table Embedding Pooling Method	max pooling, avg pooling
Lambda Calculus Embeddings included	no

Table 5.6.: The pipeline configurations for the reranker table embedding pooling measurement.

Abstraction Measurement

This measurement shows the difference in performance between perfect abstraction and our dependency-based abstraction mechanism. This test also doubles as an overall performance measurement of the best-performing system from the tests before using half the not sure threshold from the defaults. Therefore, some of the chosen values for our configurable in Table 5.8 point to the results in Section 5.1.3.

Configurable	Possible Values
Abstraction Method	mocked abstraction
Resolver Text Embedding Language Model	$\text{BERT}_{\text{SMALL}}$
Resolver Text Embedding Postprocessing	max pooling
Resolver Metric Learning	none
Table Embedding Language Model	$\text{BERT}_{\text{SMALL}}$
Table Embedding Pooling Method	max pooling
Lambda Calculus Embeddings included	yes

Table 5.7.: The pipeline configuration for the reranker lambda embedding measurement.

Configurable	Possible Values
Abstraction Method	dependency abstraction, mocked abstraction
Resolver Text Embedding Language Model	$\text{ELECTRA}_{\text{LARGE}}$
Resolver Text Embedding Postprocessing	positional encoding with average pooling
Resolver Metric Learning	LMNN
Reranker included	no

Table 5.8.: The pipeline configurations for the abstraction measurement.

One-shot Generalization Measurement

In this measurement, we check the one-shot generalization performance of 6 pipeline configurations for which each part has shown good performance on the atomic action set tests. We define these configurations in Table 5.9 and check their ability to generalize unknown instances by decomposition. For this, we use 2 measures: The first is the ratio of correctly recognized composite instances, and the second is the ratio of correctly translated instances that have been trained by the pipeline through decomposition.

This test will have no repetitions for each configuration and is the most qualitative measurement of the tests outlined here. We wanted to add it to the quantitative measurement since the results are numerical values rather than an analysis of each composite query in our dataset. Like the abstraction test, this test is also done

Configurable	Possible Values
Abstraction Method	mock abstraction
Resolver Text Embedding Language Model	BERT$_{\text{SMALL}}$, BERT$_{\text{LARGE}}$, ELECTRA$_{\text{LARGE}}$
Resolver Text Embedding Postprocessing	positional encoding with average pooling
Resolver Metric Learning	LMNN, none
Reranker included	no

Table 5.9.: The pipeline configurations for the one-shot-generalization measurement.

with half the not-sure threshold to find a balance between atomic and one-shot generalization performance.

5.1.3. Results

Resolver Measurement Results

Looking at the resolver utterance embedding measurement in Table 5.10 we already see good performance across the board without a reranker. This is shown by the worst performing language models BERT$_{\text{SMALL}}$ and ELECTRA$_{\text{SMALL}}$ already reaching an average accuracy of 76.4% on the full dataset. The best-performing language model on the full atomic dataset is ELECTRA$_{\text{LARGE}}$ achieving an accuracy of 79.1%.

We only measured small variances in the accuracy of the parsers suggesting each pipeline finds its optimal prediction reliably. Furthermore, this suggests good stability towards retraining, meaning that the performance of the parser is almost the same for every retraining operation. Although on occasion, the difference in performance can be multiple percentage points. The data also suggests that larger models are more prone to variance than smaller models.

In terms of our predefined difficulties, we measured that the impact of each higher difficulty setting is about 3 to 5 percentage points. We also see the benefits of

Difficulty	Text Embedding Model	Accuracy			
		Average	Worst	Best	Variance
1	$\text{BERT}_{\text{SMALL}}$	0.834	0.834	0.834	**0.0**
	$\text{BERT}_{\text{BASE}}$	0.881	0.881	0.881	**0.0**
	$\text{BERT}_{\text{LARGE}}$	**0.905**	**0.905**	**0.905**	**0.0**
	$\text{ELECTRA}_{\text{SMALL}}$	0.810	0.810	0.810	**0.0**
	$\text{ELECTRA}_{\text{BASE}}$	0.881	0.881	0.881	**0.0**
	$\text{ELECTRA}_{\text{LARGE}}$,	0.857	0.857	0.857	**0.0**
2	$\text{BERT}_{\text{SMALL}}$	0.808	0.808	0.808	**0.0**
	$\text{BERT}_{\text{BASE}}$	0.840	0.808	**0.865**	$1.516 \cdot 10^{-4}$
	$\text{BERT}_{\text{LARGE}}$	0.837	0.808	0.846	$1.294 \cdot 10^{-4}$
	$\text{ELECTRA}_{\text{SMALL}}$	0.808	0.808	0.808	**0.0**
	$\text{ELECTRA}_{\text{BASE}}$	**0.844**	0.827	**0.865**	$1.072 \cdot 10^{-4}$
	$\text{ELECTRA}_{\text{LARGE}}$	0.838	**0.833**	0.857	$9.070 \cdot 10^{-5}$
3	$\text{BERT}_{\text{SMALL}}$	0.764	0.764	0.764	**0.0**
	$\text{BERT}_{\text{BASE}}$	0.786	0.764	0.792	$1.042 \cdot 10^{-4}$
	$\text{BERT}_{\text{LARGE}}$	0.786	**0.778**	0.792	$4.630 \cdot 10^{-5}$
	$\text{ELECTRA}_{\text{SMALL}}$	0.764	0.764	0.764	**0.0**
	$\text{ELECTRA}_{\text{BASE}}$	0.774	0.750	0.792	$9.404 \cdot 10^{-5}$
	$\text{ELECTRA}_{\text{LARGE}}$	**0.791**	**0.778**	**0.806**	$6.703 \cdot 10^{-5}$

Table 5.10.: The measurements of the resolver text embedding test. The best values for each difficulty are bold.

using larger, more complex text embeddings decrease with increasing difficulty. This can be seen by taking the difference between the best and worst performing models for each difficulty setting. Here we see a reduction by 6.8 percentage points from 9.5 to 2.7 percentage points of accuracy.

When looking at the Table 5.11 we see that the postprocessing is more important to overall accuracy than the language model provided. Here, the best configuration uses the average pooled positionally encoded sequence embeddings from $\text{BERT}_{\text{SMALL}}$. This configuration is 5.6 percentage points better at 84.7% accuracy than $\text{ELECTRA}_{\text{LARGE}}$ from the measurement in Table 5.10.

We want to point out further that the parse performance of the best postprocessing method is sharply decreasing when comparing difficulties 2 and 3 in the data set. This suggests that average pooling positionally encoded sequences might not yield as good results on instances where conditions need to be parsed. Here a further investigation of other configurations as further work is recommended.

Besides using average pooling on positionally encoded outputs, we see that the standard sequential output is competitive, just being 1.3 percentage points less accurate. In addition, we measured quite a large gap between max pooling and average pooling of positionally encoded sequences. This is most probably due to the fact that max-pooling ignores the padded word vectors because they only contain zeroes. Therefore, we believe that average pooling in this situation captures the sequence length of the lifted utterance, which seems to help find the best candidate programs from the lifted utterances in a significant way.
Analyzing the metric learning measurement in Table 5.12 we can see a smaller advantage than measured by changing the postprocessing. The best model on the full dataset uses $\text{BERT}_{\text{SMALL}}$ and an LMNN learned metric achieving an accuracy of 79.2%. While this is still on par with the resolver using $\text{ELECTRA}_{\text{LARGE}}$ from Table 5.10, it is 5.5 percentage points short of the resolver using $\text{BERT}_{\text{SMALL}}$ and the best postprocessing method.

Reranker Measurement Results

While we have seen significant differences in parsing performance when changing parts of the resolver, we cannot make the same conclusions for the reranker. The issue seems to be that the reranker does not change the parsing performance of our pipeline whatsoever. This can be seen when we compare the measurements of all configurations of the resolver with our initial text embedding measurement in Table 5.13. Here we see that the accuracy for all our resolver configurations tested is equal to the $\text{BERT}_{\text{SMALL}}$ measurement. As an example, we provide the results of the table embedding test in Table 5.13. The remaining reranker measurements can be found in Chapter D.

Difficulty	Postprocessing Type	Accuracy			
		average	worst	best	variance
1	Sequence	**0.952**	**0.952**	**0.952**	**0.0**
	BERT pooled	0.810	0.810	0.810	**0.0**
	Sequence pos. encoded[1]	0.905	0.905	0.905	**0.0**
	Max pooled	0.834	0.834	0.834	**0.0**
	Average pooled	0.881	0.881	0.881	**0.0**
	Pos. encoded max pooled[2]	0.810	0.810	0.810	**0.0**
	Pos. encoded avg. pooled[3]	**0.952**	**0.952**	**0.952**	**0.0**
	Cubed summed sequence	0.667	0.667	0.667	**0.0**
	Cubed sum. BERT pooled[4]	0.834	0.834	0.834	**0.0**
	Cub. sum. pos. enc. seq.[5]	0.667	0.667	0.667	**0.0**
2	Sequence	0.885	0.885	0.885	**0.0**
	BERT pooled	0.827	0.827	0.827	**0.0**
	Sequence, pos. encoded[1]	0.865	0.865	0.865	**0.0**
	Max pooled	0.808	0.808	0.808	**0.0**
	Average pooled	0.846	0.846	0.846	**0.0**
	Pos. encoded max pooled[2]	0.808	0.808	0.808	**0.0**
	Pos. encoded avg. pooled[3]	**0.923**	**0.923**	**0.923**	**0.0**
	Cubed summed sequence	0.673	0.673	0.673	**0.0**
	Cubed sum. BERT pooled[4]	0.846	0.846	0.846	**0.0**
	Cub. sum. pos. enc. seq.[5]	0.673	0.673	0.673	**0.0**
3	Sequence	0.834	0.834	0.834	**0.0**
	BERT pooled	0.778	0.778	0.778	**0.0**
	Sequence, pos. encoded[1]	0.819	0.819	0.819	**0.0**
	Max pooled	0.764	0.764	0.764	**0.0**
	Average pooled	0.778	0.778	0.778	**0.0**
	Pos. encoded max pooled[2]	0.750	0.750	0.750	**0.0**
	Pos. encoded avg. pooled[3]	**0.847**	**0.847**	**0.847**	**0.0**
	Cubed summed sequence	0.653	0.653	0.653	**0.0**
	Cubed sum. BERT pooled[4]	0.778	0.778	0.778	**0.0**
	Cub. sum. pos. enc. seq.[5]	0.653	0.653	0.653	**0.0**

Table 5.11.: The measurements from the resolver text embedding postprocessing test. The best values for each difficulty are bold. [1]Sequence, positionally encoded; [2]Positionally encoded sequence, max pooled; [3]Positionally encoded sequence, average pooled; [4]Cubed summed BERT pooled vectors; [5]Cubed summed positionally encoded sequence

Difficulty	Metric Learner Type	Accuracy			
		Average	Worst	Best	Variance
1	none	0.834	0.834	0.834	**0.0**
	NCA	**0.857**	**0.857**	**0.857**	**0.0**
	LMNN	**0.857**	**0.857**	**0.857**	**0.0**
2	none	0.808	0.808	0.808	**0.0**
	NCA	**0.846**	**0.846**	**0.846**	**0.0**
	LMNN	0.827	0.827	0.827	**0.0**
3	none	0.764	0.764	0.764	**0.0**
	NCA	0.750	0.750	0.750	**0.0**
	LMNN	**0.792**	**0.792**	**0.792**	**0.0**

Table 5.12.: The measurements of the metric learning tests. The best accuracies and variances for each difficulty are bold.

We assume that the reranker feedforward network, because of the inclusion of the resolver output probability in its input vector, does fit exactly to the resolver output probability rather than looking at the input wholistically. We, therefore, do not recommend using the reranker in its current form. Rather, we would recommend changing the input vector for the reranker network closer to the original implementation of Karamcheti [1] where instead of the output probability, encoded candidate programs are added to the aligned representations of the reranker.

Difficulty	Pooling Type	Accuracy			
		Average	Worst	Best	Variance
1	average pooled	0.834	0.834	0.834	0.0
	max pooled	0.834	0.834	0.834	0.0
2	average pooled	0.808	0.808	0.808	0.0
	max pooled	0.808	0.808	0.808	0.0
3	average pooled	0.764	0.764	0.764	0.0
	max pooled	0.764	0.764	0.764	0.0

Table 5.13.: The reranker table embedding pooling type measurement results.

Abstractor Measurement Result

The abstractor measurement shows that using the dependency-based entity abstraction drastically reduces the system's accuracy to a point where it is no longer usable. As we already mentioned in Section 4.4.1 this points to the fact that significantly more linguistic work is required to cover all sentences in the data set. On the other hand, we can show that the simplistic rules used are able to abstract some instances correctly.

As for the combination of the best results from the resolver measurements, we can show a slight improvement compared to the standard ELECTRA$_{\text{LARGE}}$ model in Table 5.10 and the LMNN model in table 5.12 with an average accuracy of 80.3%. Yet this model is significantly less effective than the BERT$_{\text{SMALL}}$ model with average pooled positionally encoded sequence word vectors from Table 5.11 suggesting that some of the choices do not work together well.

Difficulty	Abstraction Type	Accuracy			
		Average	Worst	Best	Variance
1	dependency	0.167	0.167	0.167	**0.0**
	mock	**0.857**	**0.857**	**0.857**	**0.0**
2	dependency	0.077	0.077	0.077	**0.0**
	mock	**0.854**	**0.827**	**0.885**	$1.997 \cdot 10^{-4}$
3	dependency	0.061	0.056	0.069	$\mathbf{4.630 \cdot 10^{-5}}$
	mock	**0.803**	**0.778**	**0.819**	$1.018 \cdot 10^{-4}$

Table 5.14.: The abstraction measurement. As one can see, the dependency abstraction method severely impacts performance such that the parser is not usable using this abstraction method.

One-shot Generalization Measurement Result

Because of the results from the abstractor test above, we provide some more combinations which might yield good performances in this test. As we can see in Table 5.15 it seems that ELECTRA$_{\text{LARGE}}$ while being a decent choice for our atomic action set is not suited for use in one-shot generalization. Of the 8 composite instances, parsers using ELECTRA could not recognize 2 as unknown concepts. Additionally, with metric learning, the system was not able to correctly predict 2 of the 6 recognized utterances. The configuration not using metric learning fared better recognizing 5 out of 6 composites after retraining.

The remaining configurations performed well on the composite queries provided by us. Here we found that BERT$_{\text{LARGE}}$ without any metric learning performed best out of the rest with 7 correct predictions out of 8. Both the results for ELECTRA$_{\text{LARGE}}$ and BERT$_{\text{LARGE}}$ suggest that the LMNN metric learning method and the average pooled positionally encoded sequence embeddings did work well together despite them working well separately. Therefore we would suggest using the BERT$_{\text{LARGE}}$ or BERT$_{\text{SMALL}}$ models without metric learning and the average pooled positionally encoded postprocessing for the best results.

Pipeline	Composite Instances	
	Recognized as Unsure	Correctly Predicted
BERT$_{\text{LARGE}}$, LMNN	**8/8**	6/8
BERT$_{\text{LARGE}}$, no metric learning	**8/8**	**7/8**
ELECTRA$_{\text{LARGE}}$, LMNN	6/8	4/6
ELECTRA$_{\text{LARGE}}$, no metric learning	6/8	5/6
BERT$_{\text{SMALL}}$, LMNN	**8/8**	6/8
BERT$_{\text{SMALL}}$, no metric learning	**8/8**	6/8

Table 5.15.: The one-shot generalization measurement. The best values are marked using bold letters

5.2. Qualitative Evaluation

For our qualitative evaluations, we are looking at examples to explain why the parser performs as shown in the quantitative evaluation. Therefore this section will answer the "whys" of the above research questions and provide avenues to improve some aspects of the system.

We will give 2 examples in the coming sections. These will be the basis for some of our answers to the research questions **RQ1**, **RQ3**, and **RQ4**. In each section, we will look at misclassified data from the measurements above and explain the parser's challenges and limitations in certain aspects.

5.2.1. Ambiguity from Lifting Utterances

Lifting natural language utterances brings numerous advantages we can exploit to enable us to train a semantic parser from a very minimal dataset. The biggest of which is that we restrict the possible inputs for our reranker massively by removing sentence objects and replacing them with some more abstract representations like our DSL input types. Yet this also becomes a big disadvantage in certain situations as we remove crucial contextual information from our sentences.

To understand this, we look at the following example. Let us assume the user wanted to compute the total cost for a visitor renting one of their holiday homes. For this, they have merged the `cleaning` and `holiday_management` tables and added the column cost in which they entered the rent of the stay. Now, they want to add the cleaning fee to this rent to get the total cost of a visit. To do this, they use the system and provide the utterance: "Add fee to cost in holiday management.".

As you can see, we want to sum the entries in the column fee to the respective entries in the column cost. But unfortunately, the system does not have the context we have provided you in the paragraph above. In fact, because of entity abstraction, what is given to the reranker is the following: "Add [,column] [table]".

We can quickly see that this is a problem since there is another DSL operation that adds columns to a table called `ADD`. For the parser, without any semantic knowledge of each input object, this utterance is indistinguishable from a lifted utterance for adding columns.
Indeed, for the example "Add fee to cost in holiday management." most pipeline configurations will provide the DSL code:

`ADD(['holiday_management'], ['fee', 'cost'])` adding new columns rather than summing them.
In our implementation, the reranker should have extracted that both "fee" and "cost" are inside the table, and thus `SUM` is the more probable candidate out of it, and `ADD`. The implementation should technically be able to do this from the information given - but it is hard to capture in less than 100 training examples where only one example requires that the reranker moves the candidate for sum into the first place. Here more data is required to capture such situations creating a somewhat hard upper bound for atomic action performance for our system.

Implications for One-shot Generalization

This same issue is also the cause for a common mistraining of "Add fee to total rent where cleaning id is not empty." in our composition dataset. Here the decomposition of this query are the following 4 subqueries:

1. "Add the column total cost to holiday management.", which translates to `ADD(['holiday_management'], ['total cost'])`
2. "Filter holiday management where the cleaning id is not empty.", corresponding to `FILTER(['holiday_management'], cleaning id != '')`
3. "Add the column fee to total rent.", which matches `SUM(['holiday_management'], ['fee', 'total rent'])`
4. "Assign to the total cost.", which converts to `ASSIGN(['holiday_management'], ['total cost'])`

We can see that the third atomic utterance is analogous to our example above and does lift to the same lifted utterance even though no table is mentioned. This is due to the fact that the abstractor can add the table tag automatically from its context.

Consequently, the semantic parser pipelines cannot correctly translate the third utterance for the reasons above, resulting in the single composition misclassification of our best system.

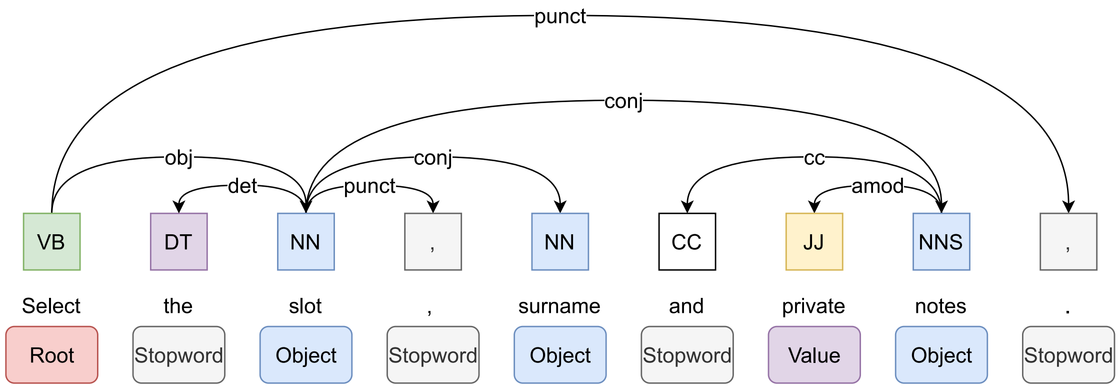


Figure 5.1.: The annotated dependency parse of "Select the slot, surname and private notes." where the annotations are the word categories assigned by the entity abstractor.

5.2.2. Missing Rules in Dependency-based Entity Abstraction

As already mentioned in Section 4.4.1 our approach to lifting objects and conditions from a sentence is incomplete. In the abstraction measurement, we have seen that this requires substantial linguistic work to provide a usable experience as the current implementation drastically reduces the parser's accuracy (see Table 5.14). Here we want to provide an example of a missing set of rules to correctly abstract the utterance "Select the slot, surname and private notes.". You can view an annotated dependency parse in Figure 5.1. The issue and the reason why the entity abstractor does abstract this incorrectly is the word "private" which gets assigned the value word category. Strictly speaking, this is correct as "private" could be a value of "notes", but we want to view this as part of the compound object "private notes" as this is a column in our table.

So for this sentence to be correctly lifted, we need a rule covering this case, where the `appos` universal dependency is assigned to the compound word category. Unfortunately is not enough to look if the parent in the dependency parse is an object because we need to be sure that the construct is not part of a condition where "private" is somehow related to "notes" as a value. Rather, we need to look in the vicinity of "notes" in the dependency parse to check whether there is no case dependency and then assign the word "private" the compound category.

5.3. Answering the Research Questions

5.3.1. How well can the semantic parser translate natural language utterances into atomic DSL operations on a limited dataset with increasing difficulty? Are there limiting factors bounding the performance of the parser?

As seen in the quantitative results in Section 5.1.3, we have measured a parsing accuracy of 84.7% with a not sure threshold of half the maximum distance between two instances of the same DSL program and 80.3% with a not sure threshold of a quarter of the maximum distance. This yields a usable parser, especially considering our training set only consists of 73 examples.

We also were able to show that the accuracy variance is low, suggesting that the method can reliably retrain and find a good fit for our dataset. Furthermore, we have shown that the parser can also parse more complex utterances containing conditions. This leads us to believe that given a perfect abstraction mechanism, the system can reliably extend to unseen atomic action data from a limited set of trained instances.
In the qualitative evaluation in Section 5.2.1, we found that one limiting factor of atomic action performance are ambiguities caused by the abstraction of natural language utterances removing context information for the resolver to use.

5.3.2. How much does each part of the semantic parser pipeline contribute to the overall accuracy? What is the best configuration of the machine learning subsystems, and why?

We found that all of the performance in our implementation comes from the resolver as the reranker did not add accuracy to the parser. This is partly due to the fact that the resolver cannot improve the parsing performance because it overfits to the resolver output probability and, as such, does not rerank any candidate programs. Also, the sparsity of data, compounds this issue as it is hard to sufficiently represent ambiguities in such a small training set resulting in the reranker not being able to intervene.

As for resolver configurations, we have shown in Table 5.11 that even smaller BERT language models can achieve good performance by choosing a suitable post-processing method. Here, we saw that the average pooling of positionally encoded sequences is the most effective representation of our text embeddings where in conjunction with the small BERT language model, we measured the highest atomic action accuracy at 84.7%.

We think this is due to the fact that we cut the BERT sequence embeddings at the input sequence length and use a zero-padded output. This means that instead of the fixed length outputs made by BERT, the padded vectors clearly show the length of the input, which propagates through the average pooling mechanism well to provide further information and thus separation for the resolver mechanism.

Another improvement has been shown using metric learning, which yields a more modest but substantial increase in performance by separating the embedded inputs (see Table 5.12). Here we found that a large margin nearest neighbor trained metric provides the best increase in performance at 79.2% accuracy. In addition, we have measured that the impact of larger language models is similar to introducing metric learning, with the large ELECTRA model being the most performant on our atomic test data achieving an accuracy of 79.1%.

We do not achieve a significant performance uplift by combining the postprocessing, metric learning, and large language model approaches. The combination achieved an 80.3% accuracy, which is 4.4 percentage points less than the best-performing parser. This suggests that some methods interfere with each other, yielding worse performances.

5.3.3. How much does the rule-based entity abstraction method negatively impact the accuracy of the parser, and why?

We found that our dependency-based entity abstraction method reduced the overall accuracy sharply by 74.2 percentage points on the whole dataset to only 6.1% percent on average and 6.9% at best making the parser unusable for any real-world applications.

The abstraction mechanism simply is too simplistic to capture all sentence structures in our dataset and needs further linguistic work introducing more rules to successfully abstract the inputs from our utterances. In Section 5.2.2 we provided one example where an adjectival modifier was incorrectly identified as a value rather than part of an object noun composition. This shows that even simple constructions need to be investigated further.

5.3.4. Is the parser realistically able to learn from decomposition? Is continual learning from one-shot generalization possible using the method outlined?

Given a perfect abstraction method, we can confirm that this method is able to reliably generalize new instances from decomposition. Our measurement has shown that in 4 different configurations, all unknown composition instances have been recognized. Of those 4 configurations, a semantic parser with a BERT$_{\text{LARGE}}$ text embedding model using average pooling on positionally encoded sequences performed best at generalizing correctly, translating 7 out of the 8 compositions given to it (see Table 5.15).

6. Conclusion

6.1. Strengths

As we have seen in Chapter 5, the semantic parser is able to reliably translate natural language utterances into a DSL that we defined. Indeed, given a suitable implementation for entity abstraction, we have shown that simple configurations of this system can achieve good performance only using minimal training data. In this constrained environment parser was able to translate complex sentences and conditions.

We did not need to train or fine-tune any language model. Moreover, we have shown that simple tools like pooling and padding can significantly boost the parser's performance. Consequently, we have proven that the system can continually learn by generalizing new composite operations from known subactions.

6.2. Weaknesses and Limitations

In its current state, there are two main factors that make the parser not usable in a real-world scenario: entity abstraction and candidate reranking.

Over the course of this work, it became clear that entity abstraction is a very complex part of this system. Unfortunately, here our implementation lacks as it only provides correct abstractions for a small subset of sentences. In fact, we believe that a rule-based entity abstraction method that can reliably extract the correct object and conditions from a sentence might not be achievable in a reasonable timeframe.

Likewise, the measurements of our reranker in Chapter 5 and Chapter D have shown that using output probabilities as part of the input for a reranker mechanism is not advisable. Our reranker implementation is not able to properly solve the problem it is designed to alleviate. Here an adaption of the reranker inputs is required.

The reranker issues highlight a fundamental problem of ambiguity when working on abstracted instances. Ambiguous utterances are only very hard to classify correctly, and a system like the one described here might not ever be able to solve this problem correctly using limited data.

6.3. Improvements and Outlook

To improve this parser, we want to focus on the entity abstraction and reranker mechanisms and provide possible solutions for them that could be implemented in future work.

We suggest two possible improvement alleys for the entity abstraction mechanism. The first would be to enhance our base program and add new rules and possibly categories to it. This way, one could minimize data usage, but it would come at a significant cost in implementation effort. Here probably more specialized linguistic knowledge is required as well to provide coverage for edge cases. Also, this would require improving the dependency parser such that the system can draw from solid input data.

Another approach would be to repose this problem for machine learning and annotate sentences that are to be lifted. Creating a dataset sufficiently large dataset would be the most obvious problem. What could be done is to create a preliminary generative system for an adversarial training approach on a small set of hand-crafted data. The idea would be for the generator to create lifted sentences that a discriminative system modifies if it is only partially incorrect, training both the generator and discriminator. This might yield a system capable enough for this semantic parser to work. One could also combine both approaches, where the rule-based method provides "gold" labeled data for the training task of the adversarial approach for sentences where the rule-based method functions correctly.

For the reranker, we would recommend redoing some parts of the embedding such that the output probability of the reranker is not included as an input for the reranking feedforward network. A simple change would be to use the language model to embed the candidate programs for the reranker. A more complex approach could be to use a method akin to the TAPEX paper of Liu et al. [7] and get a candidate program embedding by executing it using sampled inputs from the input table and embedding the outputs.

Beyond entity abstraction and reranking changes, we would also suggest increasing the dataset size and using the extended set to fine-tune an ELECTRA language model. In contrast to BERT models, these do not need as much domain data to create custom embeddings and should improve performance and separation substantially. This could yield interesting performances in both the atomic and one-shot generalization tasks

6.4. Summary

Over the course of this work, we have adopted a continual learning strategy proposed by Karamcheti et al. [1] and tested it on a hand-crafted dataset with three increasing difficulties. To achieve this, we defined a small unimplemented domain-specific language for spreadsheet operations defined along 4 different task domains to check parsing accuracy of the system.
As part of our semantic parser, we also provided some solution approaches to issues arising from implementing the continual learning strategy for spreadsheet operations. In contrast to the original strategy proposed by Karamcheti et al., we use transformer-based word embeddings, a metric learning strategy, and different text embedding postprocessing methods to improve data separation. Furthermore, we added an ad-hoc method of capturing table meaning. Furthermore, as part of the system, we defined a dependency-based entity abstraction mechanism for the parser.

While we were not able to report accuracies beyond 6.9% with the dependency-based abstraction method on the full dataset, we could verify the validity of the continual learning concept by Karamcheti et al. [1] using a mocked abstraction method, achieving reliable one-shot generalization performance and a maximum parsing accuracy of 84.7%.

Appendices

A. Recurrent Neural Networks as a Motivation for Transformers

Unfortunately, feedforward neural networks are incapable of taking variable length inputs, nor do they have the ability to carry information through multiple layers. As a solution, recurrent networks like the Elman network [53] were devised (see figure A.1). They operate very much like feedforward networks at inference. The only difference is that a weighted sum from the last hidden layer activation of the previous input in the sequence (e.g., the previous word in the sentence) is added to the usual activation and then put into the activation function σ.

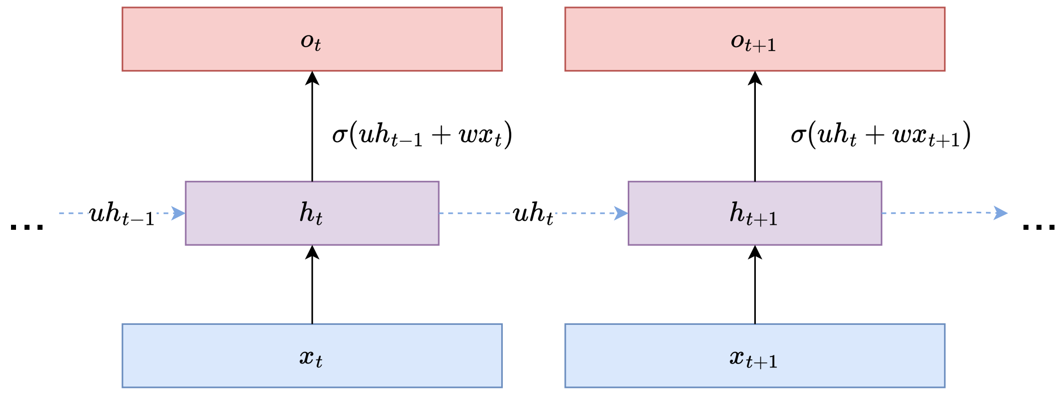


Figure A.1.: Overview of an Elman network. σ is the activation function x_t, h_t and o_t are the input, hidden activation and output at time t (e.g. the t-th input in the sequence). w and u are the input and recurrent hidden activation weights matrices.

Therefore at training, the gradient of the error is now also dependent on the hidden activation from the last input in sequence. Consequently, this is represented in ∇C. Because of this temporal dependency, the adapted backpropagation algorithm is called backpropagation through time. While we will not provide the adapted chain rule in this work, we want to mention that two passes are required now. The first pass is a forward pass of the network generating the errors for each input, while the second pass propagates back through the sequence output errors generating ∇C.

A.0.1. Long Short-Term Memory

While better in capturing dependencies in input sequences than feedforward neural networks, a recurrent network can only capture short dependencies. This is due to the fact that weights are changed proportionally to their partial derivatives. As seen in equation (2.23), the backpropagation algorithm multiplies every partial derivative with the previous activation. Since activations are usually in $[0, 1]$, the gradient becomes vanishingly small, and thus the first layers in a deep neural network are not trained at all. This is made worse for recurrent networks because input sequences can be arbitrarily large, and thus, the network is arbitrarily deep [54].

The long short-term memory model (LSTM) [55] aims to solve this problem by enhancing the way information is transferred in a recurrent neural network enabling the gradient to flow through the whole network if needed (see figure A.2).
Even though the cell is more complex than Elman networks, one can still use backpropagation through time to train this recurrent neural network. When using an additional LSTM model in the opposite sequence direction, we speak of a bidirectional LSTM where both temporal dependencies can be captured.

A.0.2. Encoder-Decoder Architecture

The encoder-decoder architecture is a very popular machine learning architecture in natural language processing. It consists of two RNNs. One encodes a word sequence into some intermediary vector representation while the second model decodes the vectors into the target representation. Up until recently, the strongest models in almost all natural language tasks were dominated by this architecture [37, 56, 57].

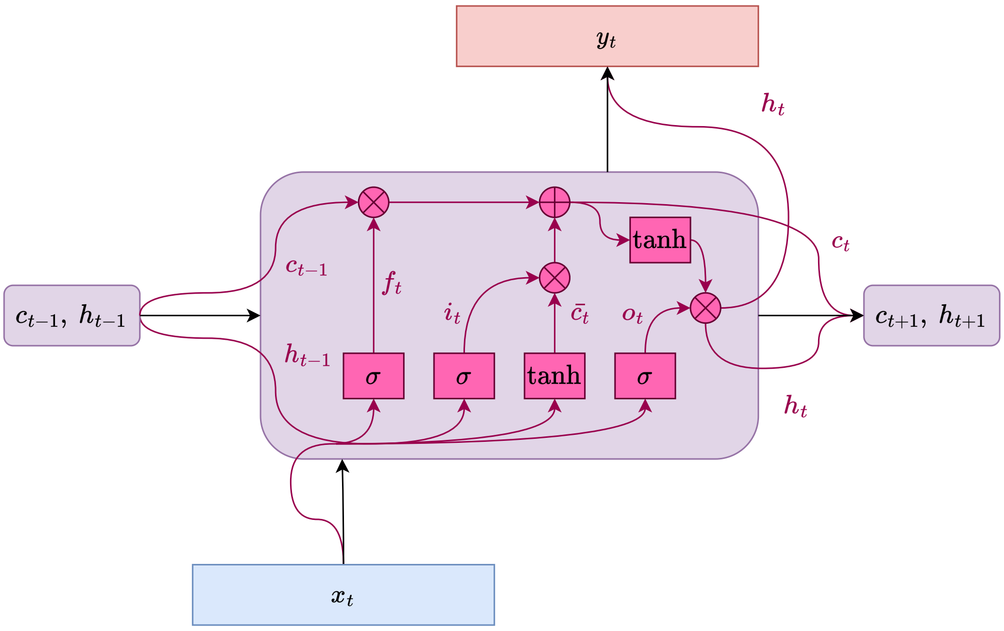


Figure A.2.: Overview of a LSTM cell. The $+$ and $\times$ signs represent vector addition and the element-wise product. c_t is the cell activation, f_t the forget gate activation, i_t the input gate activation, o_t the output gate activation and h_t the hidden state.

B. List of Universal Dependencies

This is a list of all dependencies defined by Nivre et al. [28].

- `acl`: clausal modifier of noun (adnominal clause)
- `acl:relcl`: relative clause modifier
- `advcl`: adverbial clause modifier
- `advmod`: adverbial modifier
- `advmod:emph`: emphasizing word, intensifier
- `advmod:lmod`: locative adverbial modifier
- `amod`: adjectival modifier
- `appos`: appositional modifier
- `aux`: auxiliary
- `aux:pass`: passive auxiliary
- `case`: case marking
- `cc`: coordinating conjunction
- `cc:preconj`: preconjunct
- `ccomp`: clausal complement
- `clf`: classifier
- `compound`: compound
- `compound:lvc`: light verb construction
- `compound:prt`: phrasal verb particle
- `compound:redup`: reduplicated compounds
- `compound:svc`: serial verb compounds
- `conj`: conjunct
- `cop`: copula
- `csubj`: clausal subject
- `csubj:outer`: outer clause clausal subject
- `csubj:pass`: clausal passive subject

- `dep`: unspecified dependency
- `det`: determiner
- `det:numgov`: pronominal quantifier governing the case of the noun
- `det:nummod`: pronominal quantifier agreeing in case with the noun
- `det:poss`: possessive determiner
- `discourse`: discourse element
- `dislocated`: dislocated elements
- `expl`: expletive
- `expl:impers`: impersonal expletive
- `expl:pass`: reflexive pronoun used in reflexive passive
- `expl:pv`: reflexive clitic with an inherently reflexive verb
- `fixed`: fixed multiword expression
- `flat`: flat multiword expression
- `flat:foreign`: foreign words
- `flat:name`: names
- `goeswith`: goes with
- `iobj`: indirect object
- `list`: list
- `mark`: marker
- `nmod`: nominal modifier
- `nmod:poss`: possessive nominal modifier
- `nmod:tmod`: temporal modifier
- `nsubj`: nominal subject
- `nsubj:outer`: outer clause nominal subject
- `nsubj:pass`: passive nominal subject
- `nummod`: numeric modifier
- `nummod:gov`: numeric modifier governing the case of the noun
- `obj`: object
- `obl`: oblique nominal
- `obl:agent`: agent modifier
- `obl:arg`: oblique argument
- `obl:lmod`: locative modifier
- `obl:tmod`: temporal modifier

- `orphan`: orphan
- `parataxis`: parataxis
- `punct`: punctuation
- `reparandum`: overridden disfluency
- `root`: root
- `vocative`: vocative
- `xcomp`: open clausal complement

C. List of Tables in the Dataset

Name	Surname	Student number	Attachment name	Date
Leo	Dorste	4052766	4052766/attest.pdf	13.08.44
Sabrina	Engler	4162332	4162332/attest.pdf	07.05.22
Alexandra	Schiffer	4008571	4008571/attest.pdf	03.12.22
Maike	Baum	4187296	4187296/attest.pdf	12.08.21
Andreas	Lautenbecher	4197295	4197295/attest.pdf	04.05.20
Nick	Freund	4193759	4193759/attest.pdf	27.03.22
Stefanie	Goldschmidt	4068273	4068273/attest.pdf	05.07.22
Tobias	Kuester	4099183	4099183/attest.pdf	25.09.21
Doreen	Bergmann	4182933	4182933/attest.pdf	29.10.22
Christin	Gersten	4057971	4057971/attest.pdf	18.07.21
Felix	Wulf	4095826	4095826/attest.pdf	12.02.23
Tanja	Engel	4112746	4112746/attest.pdf	19.11.22
Barbara	Reinhardt	4096823	4096823/attest.pdf	17.04.22
Sarah W.	Saenger	4175552	4175552/attest.pdf	01.09.22

Table C.1.: The `medical_certificates` table from the Medical Certificate Upload task in section 4.3.5.

Cleaning id	Cleaner	Fee
1	Martha Ludwig	30
2	Klaus Weber	15
3		
4	Martha Ludwig	20
5		
6		
7		
8	Jens Paukner	30
9	Tobias Knecht	25

Table C.2.: The `cleaning` table from the Holiday Home Guests Management task in section 4.3.3.

Name	Surname	Phone	Arrival	STA	Departure	STD	Cleaning id
Steffen	Berg	3435887465	07/12/2022	1:00pm	17/12/2022	12:30pm	1
Karolin	Schreiber	9874785342	12/06/2022	10:00am	22/06/2022	3:30pm	
Dirk	Schreiber	2324657732	12/06/2022	10:00am	22/06/2022	11:00pm	2
Leonie	Hoover	2891336492	13/08/2023	4:00pm	19/08/2023	11:00pm	3
Karin	Kaiser	9927645433	05/07/2022	2:00pm	08/07/2022	12:30pm	4
Monika	Roth	9231556577	01/03/2023	2:00pm	11/03/2023	1:00pm	
Frank	Diederich	1173544623	24/08/2022	1:00pm	29/08/2022	3:30pm	5
Melanie	Frueh	7836634776	05/09/2022	11:30am	13/09/2022	1:00pm	6
Niklas	Trommler	9818235537	17/04/2023	2:00pm	23/04/2023	1:00pm	
Johannes	Trommler	3134767892	17/04/2023	2:00pm	23/04/2023	2:00pm	
Leon	Ebersbacher	8834011254	02/02/2022	9:00am	08/02/2022	9:30am	7
Agnes	Kober	9948274925	09/01/2022	12:00am	21/01/2022	10:30am	8
Alexander	Hertz	5744938175	27/05/2023	11:30am	05/06/2023	3:00pm	9
Vanessa	Dreher	2956309572	10/09/2022	4:00pm	17/09/2022	12:30pm	

Table C.3.: The `holiday_management` table from the Holiday Home Guests Management task in section 4.3.3.

Date	Slot	Name	Surname	Email	Phone	Private Notes
28.03.22	1200-1300					office hours
01.04.22	1000-1015	Juergen	Bieber	JurgenBieber@rhyta.com	6138848100	first meeting
01.04.22	1015-1030	Ralph	Eberhardt	RalphEberhardt@jourrapide.com	2622449580	wants to know about programming basics
01.04.22	1045-1100	Dennis	Braun	DennisBraun@teleworm.us	6020797114	interesting project advice
01.04.22	1100-1145	Julia	Farber	JuliaFarber@teleworm.us	5323961496	kick off
01.04.22	1200-1215	Sophie	Boehm	SophieBoehm@teleworm.us	3940528208	regular meet
02.04.22	1200-1300					office hours
03.04.22	1200-1300					office hours
04.04.22	0900-0915	Maria	Schmitz	MariaSchmitz@armyspy.com	7252323962	questions about exams
04.04.22	1330-1400	Steffen	Schulze	SteffenSchulze@jourrapide.com	9383460916	
04.04.22	1415-1430	Katharina	Nacht	KatharinaNacht@jourrapide.com	7255610166	regular meet
05.04.22	1600-1615	Robert	Pfeiffer	RobertPfeiffer@armyspy.com	9519884353	questions about project
05.04.22	1615-1700	Laura	Freytag	LauraFreytag@rhyta.com	3417252072	kick off
05.04.22	1700-1715	Florian	Wagner	FlorianWagner@dayrep.com	6281589915	first meeting
06.04.22	1200-1300					office hours

Table C.4.: The `time_slots` table from the Calendar Slots task in section 4.3.2.

Name	Surname	Phone	Address	Title	Institute	Role	Staff ID	Committee
Alexander	Bachmeier	2162859856	Brandenburgische Str.16, 7460 Ilsfeld		IWR	PhD Student	rf554	
Johanna	Fischer	5536431826	Mollstrasse 14, 33189 Schlangen	Prof. Dr.	IWR	Teaching	dk813	exam, funding
Sabrina	Bosch	5286703944	Rohrdamm 42,32108 Bad Salzuflen Ehrsen-Breden	Assoc. Prof. Dr.	IWR	Teaching	bs006	exam
Wolfgang	Schuster	9116836528	Rathausstrasse 14, 90221 Nürnberg		IWR	Master Student	gh439	
Tanja	Eichel	9403255415	Genterstrasse 29, 24044 Kiel		IWR	Bachelor Student	aq223	
Manuela	Hoffmann	1848462231	Alt-Moabit 69, 04831 Eilenburg		IWR	RA	fn751	
Markus	Angelberg	3284677306	Kantstraße 27, 36422 Bad Salzungen		IWR	Administration	da466	personel
Vanessa	Hueber	7324994018	Eichendorffstr. 1, 88047 Friedrichshafen Raderach		IWR	Administration	cs321	personel
Irene	Graf	2546708838	Kantstrasse 40, 90459 Nürnberg		IWR	PhD Student	nb456	
Agnes	Steinmeier	4782934345	Koenigstrasse 36, 88250 Weingarten		IWR	RA	lk302	
Tim	Wolf	9151728837	Hardenbergstraße 59, 66887 Sankt Julian		IWR	RA	ja777	
Frederick	Hebler	1233487829	Spresstrasse 21, 24918 Flensburg		IWR	Administration	op987	personel
Elisabeth	Treutner	9845223784	Hallesches Ufer 36, 71157 Hildrizhausen		IWR	Bachelor Student	uu734	

Table C.5.: The head of the `faculty` table from the Faculty Database task in section 4.3.4.

D. Additional Reranker Measurements

Difficulty	Accuracy			
	Average	Worst	Best	Variance
1	0.834	0.834	0.834	0.0
2	0.808	0.808	0.808	0.0
3	0.764	0.764	0.764	0.0

Table D.1.: The results of the reranker lambda embedding measurement. Here, the semantic parser pipeline has a reranker attached that uses another parser to inform its output.

Difficulty	Table Embedding Language Model	Accuracy			
		Average	Worst	Best	Variance
1	$\text{BERT}_{\text{SMALL}}$	0.834	0.834	0.834	0.0
	$\text{BERT}_{\text{BASE}}$	0.834	0.834	0.834	0.0
	$\text{BERT}_{\text{LARGE}}$	0.834	0.834	0.834	0.0
	$\text{ELECTRA}_{\text{SMALL}}$	0.834	0.834	0.834	0.0
	$\text{ELECTRA}_{\text{BASE}}$	0.834	0.834	0.834	0.0
	$\text{ELECTRA}_{\text{LARGE}}$	0.834	0.834	0.834	0.0
2	$\text{BERT}_{\text{SMALL}}$	0.808	0.808	0.808	0.0
	$\text{BERT}_{\text{BASE}}$	0.808	0.808	0.808	0.0
	$\text{BERT}_{\text{LARGE}}$	0.808	0.808	0.808	0.0
	$\text{ELECTRA}_{\text{SMALL}}$	0.808	0.808	0.808	0.0
	$\text{ELECTRA}_{\text{BASE}}$	0.808	0.808	0.808	0.0
	$\text{ELECTRA}_{\text{LARGE}}$	0.808	0.808	0.808	0.0
3	$\text{BERT}_{\text{SMALL}}$	0.764	0.764	0.764	0.0
	$\text{BERT}_{\text{BASE}}$	0.764	0.764	0.764	0.0
	$\text{BERT}_{\text{LARGE}}$	0.764	0.764	0.764	0.0
	$\text{ELECTRA}_{\text{SMALL}}$	0.764	0.764	0.764	0.0
	$\text{ELECTRA}_{\text{BASE}}$	0.764	0.764	0.764	0.0
	$\text{ELECTRA}_{\text{LARGE}}$	0.764	0.764	0.764	0.0

Table D.2.: The results of the reranker table embedding measurement.

List of Figures

List of Tables

Bibliography

[1] Siddharth Karamcheti, Dorsa Sadigh, and Percy Liang. Learning adaptive language interfaces through decomposition. *CoRR*, abs/2010.05190, 2020.

[2] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, January 2011.

[3] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[4] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. *CoRR*, abs/1606.03622, 2016.

[5] Zhiyu Chen, Mohamed Trabelsi, Jeff Heflin, Yinan Xu, and Brian D. Davison. Table search using a deep contextualized language model. *CoRR*, abs/2005.09207, 2020.

[6] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. Tabert: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314*, 2020.

[7] Qian Liu, Bei Chen, Jiaqi Guo, Zeqi Lin, and Jian-Guang Lou. TAPEX: table pre-training via learning a neural SQL executor. *CoRR*, abs/2107.07653, 2021.

[8] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[10] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. *CoRR*, abs/1508.00305, 2015.

[11] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Automating formalization by statistical and semantic parsing of mathematics. pages 12–27, 08 2017.

[12] Robert T. Kasper. A flexible interface for linking applications to penman's sentence generator. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '89, page 153–158, USA, 1989. Association for Computational Linguistics.

[13] Shibhansh Dohare and Harish Karnick. Text summarization using abstract meaning representation. *CoRR*, abs/1706.01678, 2017.

[14] Linfeng Song, Xiaochang Peng, Yue Zhang, Zhiguo Wang, and Daniel Gildea. Amr-to-text generation with synchronous node replacement grammar. *CoRR*, abs/1702.00500, 2017.

[15] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[16] Yoav Artzi and Luke Zettlemoyer. UW SPF: the university of washington semantic parsing framework. *CoRR*, abs/1311.3011, 2013.

[17] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018.

[18] Masashi Yoshikawa, Hiroshi Noji, and Yuji Matsumoto. A* ccg parsing with a supertag and dependency factored model. 2017.

[19] Martin Fowler. *Domain-Specific Languages.* Addison-Wesley, 2010.

[20] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[21] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345, 1936.

[22] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[23] Dan Jurafsky and James H. Martin. *Speech and Language Processing.* draft 3rd edition, 2022.

[24] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[25] Mark Steedman. Combinatory grammars and parasitic gaps. *Natural Language & Linguistic Theory*, 5(3):403–439, 1987.

[26] Kevin Clark, Minh-Thang Luong, Christopher D Manning, and Quoc V Le. Semi-supervised sequence modeling with cross-view training. *arXiv preprint arXiv:1809.08370*, 2018.

[27] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *ACL*, 2014.

[28] Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajic, Christopher D. Manning, Sampo Pyysalo, Sebastian Schuster, Francis M. Tyers, and Daniel Zeman. Universal dependencies v2: An evergrowing multilingual treebank collection. *CoRR*, abs/2004.10643, 2020.

[29] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R (Springer Texts in Statistics).* Springer, 7 2021.

[30] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[31] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[32] Aurélien Bellet, Amaury Habrard, and Marc Sebban. A survey on metric learning for feature vectors and structured data. *CoRR*, abs/1306.6709, 2013.

[33] André-Louis Cholesky. Note sur une méthode de résolution des équations normales provenant de l'application de la méthode des moindres carrés a un système d'équations linéaires en nombre inférieur a celui des inconnues. — application de la méthode a la résolution d'un système defini d'équations linéaires. *Bulletin géodésique*, 2(1):67–77, Apr 1924.

[34] Jacob Goldberger, Sam T. Roweis, Geoffrey E. Hinton, and Ruslan Salakhutdinov. Neighbourhood components analysis. In *NIPS*, 2004.

[35] Kilian Q. Weinberger and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. *J. Mach. Learn. Res.*, 10:207–244, jun 2009.

[36] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

[37] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[39] Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *ArXiv*, abs/1607.06450, 2016.

[40] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. ELECTRA: pre-training text encoders as discriminators rather than generators. *CoRR*, abs/2003.10555, 2020.

[41] Matthew E. Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. Semi-supervised sequence tagging with bidirectional language models. *CoRR*, abs/1705.00108, 2017.

[42] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. *CoRR*, abs/1606.05250, 2016.

[43] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *CoRR*, abs/1806.03822, 2018.

[44] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[45] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[46] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: The impact of student initialization on knowledge distillation. *CoRR*, abs/1908.08962, 2019.

[47] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[48] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *CoRR*, abs/1809.08887, 2018.

[49] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461, 2019.

[50] Ziyu Yao, Yiqi Tang, Wen-tau Yih, Huan Sun, and Yu Su. An imitation game for learning semantic parsers from user interaction. *CoRR*, abs/2005.00689, 2020.

[51] William de Vazelhes, C. J. Carey, Yuan Tang, Nathalie Vauquier, and Aurélien Bellet. metric-learn: Metric learning algorithms in python. *CoRR*, abs/1908.04710, 2019.

[52] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[53] Jeffrey L. Elman. Finding structure in time. *Cogn. Sci.*, 14:179–211, 1990.

[54] Josef Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen, Jun 1991.

[55] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[56] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. *CoRR*, abs/1603.06042, 2016.

[57] Pengcheng Yin and Graham Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. *CoRR*, abs/1810.02720, 2018.